

Performance Modeling and Prediction for Dense Linear Algebra

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Elmar Peise, Master of Science
aus Aachen, Deutschland.

Berichter: Universitätsprofessor Paolo Bientinesi, Ph.D.
Universitätsprofessor Dr. rer. nat. Matthias Müller
John Gunnels, Ph.D.

Tag der mündlichen Prüfung: 30. November 2017

Diese Dissertation ist auf den Internetseiten
der Universitätsbibliothek online verfügbar.

Abstract

This dissertation introduces measurement-based performance modeling and prediction techniques for dense linear algebra algorithms. As a core principle, these techniques avoid executions of such algorithms entirely, and instead predict their performance through runtime estimates for the underlying compute kernels. For a variety of operations, these predictions allow to quickly select the fastest algorithm configurations from available alternatives. We consider two scenarios that cover a wide range of computations:

To predict the performance of blocked algorithms, we design algorithm-independent performance models for kernel operations that are generated automatically once per platform. For various matrix operations, instantaneous predictions based on such models both accurately identify the fastest algorithm, and select a near-optimal block size.

For performance predictions of BLAS-based tensor contractions, we propose cache-aware micro-benchmarks that take advantage of the highly regular structure inherent to contraction algorithms. At merely a fraction of a contraction’s runtime, predictions based on such micro-benchmarks identify the fastest combination of tensor traversal and compute kernel.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Paolo Bientinesi. While guiding me through my studies, he always embraced my own ideas and helped me shape and develop them in countless discussions. While he granted me freedom in many aspects of my work, he always had time for anything between a quick exchange of thoughts and extensive brainstorming sessions. Beyond our professional relationship, we enjoyed twisty puzzles and board games in breaks from work, long game nights, and annual trips to SPIEL. I consider myself lucky to have spent my time as a doctoral student with him and his research group.

The HPAC group proved to be much more than a collection of researchers working on remotely associated projects; my colleagues were not only a source of incredibly valuable discussions and feedback regarding my work, we also indulged in various unrelated arguments and exchanges over lunch and at many other occasions. My thanks go to Edoardo Di Napoli, Diego Fabregat-Traver, Paul Springer, Jan Winkelmann, Henrik Barthels, Markus Höhnerbach, Sebastian Achilles, William McDoniel, and Caterina Fenu, as well as our former group members Matthias Petschow, Roman Iakymchuk, Daniel Taming, and Lucas Beyer.

I am grateful for financial support from the DEUTSCHE FORSCHUNGSGEMEINSCHAFT (DFG) through grant GSC 111 (the graduate school AICES) and the DEUTSCHE TELEKOMSTIFTUNG. Their programs not only funded my work, but opened further opportunities in the form of seminars and workshops, and connected me with like-minded students from various disciplines.

The RWTH IT CENTER provided and maintained an extremely reliable infrastructure central to my work: the RWTH COMPUTE CLUSTER. I thank its

staff not only for ensuring smooth operations but also for their competent and detailed responses to my many inquiries and requests regarding our institute's cluster partition.

The AICES service team did their best to shield me from the bureaucracy of contracts, stipends, and reimbursements. I am grateful they allowed me to focus solely on my research.

Even more important than a gratifying work environment is forgetting about it every once in a while. My friends played a bigger role in this effort than probably most of them know, whether we were simply spending time hanging out or playing games, went swimming, climbing or playing badminton, or taught swimming and worked as lifeguards. You are too many to enumerate, but you know who you are.

Finally, but most importantly, none of this would have been possible without the endless and uncompromising support of my parents. You are the reason I grew into the person I am today. Danke!

Contents

1	Introduction	1
1.1	Performance Modeling for Blocked Algorithms	3
1.1.1	Motivation: Blocked Algorithms	4
1.1.2	Prediction through Performance Models	9
1.2	Micro-Benchmarks for Tensor Contractions	10
1.2.1	Motivation: Tensor Contraction Algorithms	10
1.2.2	Prediction through Micro-Benchmarks	13
1.3	Related Work	14
1.3.1	Dense Linear Algebra Libraries and Algorithms	15
1.3.1.1	BLAS and LAPACK	15
1.3.1.2	Blocked Algorithms	16
1.3.1.3	Alternatives to Blocked Algorithms	17
1.3.1.4	Distributed-Memory and Accelerators	18
1.3.2	Performance Measurements and Profiling	19
1.3.3	Performance Modeling and Predictions	19
1.3.4	Tensor Contractions	21
2	Performance Effects and Measurements	23
2.1	Performance Effects for Dense Linear Algebra Kernels	24
2.1.1	Library Initialization Overhead	24
2.1.2	Fluctuations	25
2.1.2.1	Background and System Noise	25
2.1.2.2	INTEL TURBO BOOST	27
2.1.2.3	Distinct Long-Term Performance Levels	29
2.1.3	Thread Pinning	30

Contents

2.1.4	NUMA Effects	31
2.1.5	Caching	33
2.1.6	Summary	34
2.2	Measurements and Experiments: ELAPS	34
2.2.1	The SAMPLER	35
2.2.2	The ELAPS PYTHON Framework	38
2.3	Summary	41
3	Performance Modeling	43
3.1	Kernel Argument Analysis	44
3.1.1	Flag Arguments	47
3.1.2	Scalar Arguments	49
3.1.3	Leading Dimension Arguments	51
3.1.3.1	Alignment to Cache-Lines	51
3.1.3.2	Set-Associative Cache Conflicts	52
3.1.4	Increment Arguments	55
3.1.5	Size Arguments	57
3.1.5.1	Smalls Scale Behavior	58
3.1.5.2	Piecewise Polynomial Behavior	59
3.1.6	Data Arguments	61
3.1.7	Summary	63
3.2	Model Generation	64
3.2.1	Model Structure	65
3.2.2	Sample Distribution	67
3.2.3	Repeated Measurements and Summary Statistics	69
3.2.4	Relative Least-Squares Polynomial Fitting	70
3.2.5	Adaptive Refinement	72
3.3	Model Generator Configuration	75
3.3.1	Configuration Parameters	75
3.3.2	Trade-Off and Configuration Selection	77
3.3.3	Variations of the Default Configuration	83
3.4	Summary	83

4	Model-Based Predictions for Blocked Algorithms	85
4.1	Performance Prediction	86
4.2	Accuracy Quantification	89
4.3	Accuracy Case Study: Cholesky Decomposition	91
4.3.1	Varying Problem Size	92
4.3.2	Varying Block Size	96
4.3.3	Varying Problem Size and Block Size	97
4.3.4	Other Data-Types	99
4.3.5	Multi-Threaded BLAS	100
4.3.6	Summary	102
4.4	Accuracy Study: Blocked LAPACK Algorithms	102
4.4.1	Single-Threaded BLAS	105
4.4.2	Multi-Threaded BLAS	109
4.4.3	Summary	112
4.5	Algorithm Selection	112
4.5.1	Cholesky Decomposition	112
4.5.2	Triangular Inversion	114
4.5.3	Sylvester Equation Solver	117
4.5.3.1	Algorithms	118
4.5.3.2	Algorithm Selection	121
4.5.4	Summary	123
4.6	Block Size Optimization	124
4.6.1	Cholesky Decomposition	125
4.6.2	Triangular Inversion	128
4.6.3	LAPACK Algorithms	129
4.7	Summary	133
5	Cache Modeling and Prediction	135
5.1	Case Study: QR Decomposition on a HARPertown E5450	136
5.1.1	Timing Kernels in LAPACK's dgeqrf	136
5.1.2	Cache-Aware Timings	138
5.1.3	Modeling the Cache	141

5.1.4	Varying the Setup	144
5.2	Application to Other Algorithms	146
5.2.1	Cholesky Decomposition: <code>dpotrf_U</code>	147
5.2.2	Inversion of a Triangular Matrix: <code>dtrtri_{LN}</code>	148
5.2.3	Summary	149
5.3	Feasibility on Modern Hardware	150
5.3.1	In- and Out-of-Cache Timings	151
5.3.2	Algorithm-Aware Timings	155
5.4	Summary	157
6	Micro-Benchmarks for Tensor Contractions	159
6.1	Algorithm Generation	160
6.2	Runtime Prediction	164
6.2.1	Example Contraction: $C_{abc} := A_{ai}B_{ibc}$	165
6.2.2	Repeated Execution	167
6.2.3	Operand Access Distance	169
6.2.4	Cache Prefetching	174
6.2.5	Prefetching Failures	176
6.2.6	First Loop Iterations	177
6.3	Results	179
6.3.1	Changing the Setup for $C_{abc} := A_{ai}B_{ibc}$	180
6.3.2	Vector Contraction: $C_a := A_{iaj}B_{ji}$	181
6.3.3	Challenging Contraction: $C_{abc} := A_{ija}B_{jbic}$	182
6.3.4	Efficiency Study	186
6.4	Summary	187
7	Conclusion	189
7.1	Outlook	190
A	Terminology: Performance and Efficiency	191
A.1	Workload	192
A.1.1	Floating-Point Operations	192
A.1.2	Data Volume and Movement	194

A.1.3	Arithmetic Intensity	195
A.2	Runtime	197
A.3	Performance and Attained Bandwidth	198
A.4	Hardware Constraints	199
A.5	Efficiency	202
A.5.1	Compute-Bound Efficiency	202
A.5.2	Bandwidth-Bound Efficiency	204
A.5.3	The Roofline Model	205
A.6	Other Metrics	208
B	Dense Linear Algebra Routines and Libraries	209
B.1	Storage Format	209
B.1.1	Scalars	210
B.1.2	Vectors	210
B.1.3	Matrices	210
B.2	BASIC LINEAR ALGEBRA SUBPROGRAMS	211
B.2.1	BLAS Level 1	212
B.2.2	BLAS Level 2	213
B.2.3	BLAS Level 3	214
B.3	LINEAR ALGEBRA PACKAGE	218
B.4	Implementations	225
C	Hardware	229
C.1	HARPERTOWN E5450	229
C.2	SANDY BRIDGE-EP E5-2670	230
C.3	IVY BRIDGE-EP E5-2680 v2	230
C.4	HASWELL-EP E5-2680 v3	231
C.5	BROADWELL I7-5557U	232
	List of Examples	233
	List of Figures	237

Contents

List of Tables	243
Bibliography	245

1 Introduction

Software developers in scientific computing are often faced with performance-critical decisions such as the choice of algorithms, configuration parameters, hardware platforms, and software libraries. This dissertation presents novel techniques and tools to guide such decisions for dense linear algebra computations with accurate yet fast performance predictions. These predictions avoid the otherwise common exhaustive execution and timing of all potential alternatives, and thereby shorten the decision-making process both in compute time and developer effort.

The task of accurately predicting the performance of dense linear algebra algorithms is particularly challenging due to the complexity of the performance-related factors: The runtime of compute-kernels is not only non-linear in the problem size due to multi-threading and kernel-internal caching effects, but is also influenced by data locality and caching in sequences of such kernels. As a result, analytical performance predictions are either extremely rough and complex, or hardware-dependent; in contrast, this work investigates measurement-based techniques that are tailored to represent the kernel-specific performance effects.

The goal of measurement-based predictions is to estimate the performance of an algorithm both accurately and notably faster than the algorithm execution itself. These requirements lead to two practical alternatives as the basis for performance predictions: an algorithm-independent database of performance models for the building blocks that are automatically generated once per platform, or micro-benchmarks that execute a fraction of the algorithm's building blocks and extrapolate their runtime. Neither of these alternatives is applicable in all situations, and which one is more suitable depends on the type algorithm. By addressing two different types of operations that are at

1 Introduction

the core of many dense computations, this work investigates both alternatives: Blocked algorithms are predicted through algorithm-independent performance models, and tensor contraction algorithms are predicted through cache-aware micro-benchmarks.

Contributions

The main contributions of this work are the following:

- ELAPS, a lightweight yet portable and universal performance measurement framework for dense linear algebra routines and algorithms,
- Methods and tools for the automated generation of highly accurate performance models for compute kernels,
- Model-based performance predictions of blocked algorithms for optimal algorithm selection and configuration,
- A study on the influence of caching on kernel invocations within blocked algorithms, and
- Cache-aware micro-benchmarks to predict BLAS-based tensor contractions for optimal algorithm selection.

Outline

The remainder of this dissertation is structured as follows:

- [Chapter 1](#) proceeds to introduce blocked algorithms and tensor contractions, and motivates our performance prediction goals in [Sections 1.1](#) and [1.2](#). It concludes with an overview of related work in [Section 1.3](#).
- [Chapter 2](#) addresses common performance characteristics of compute kernels, and introduces ELAPS, a novel framework for performance measurements that serves as the basis for the following Chapters.
- [Chapter 3](#) presents the design and automatic generation of performance models, and analyzes their accuracy.

- [Chapter 4](#) uses such models to predict the runtime and performance of blocked algorithms and subsequently select platform-specific optimal algorithm configurations.
- [Chapter 5](#) studies the influence of caching on the runtime of compute kernels within blocked algorithms and the feasibility of accounting for caching effects in predictions.
- [Chapter 6](#) is devoted to predicting the performance of BLAS-based tensor contractions. It describes the creation of cache-aware micro-benchmarks that, for a given contraction, allow to identify the fastest algorithm(s).
- [Chapter 7](#) concludes this dissertation, summarizes the presented techniques and results, and gives an overview of potential extensions of this work.

The main chapters are supplemented by three appendices:

- [Appendix A](#) introduces readers new to high-performance computing to performance-related terminology and concepts.
- [Appendix B](#) gives an overview of the BLAS and LAPACK interfaces, their kernels used in this work, and relevant implementations.
- [Appendix C](#) details the hardware used throughout this work.

1.1 Performance Modeling for Blocked Algorithms

We aim to predict the performance of blocked algorithms with the goals of 1) selecting the fastest algorithm from a set of mathematically equivalent alternatives, and 2) tuning their algorithmic block size. In the following, [Section 1.1.1](#) introduces the concept of blocked algorithms, and exposes their inherent optimization challenges, and [Section 1.1.2](#) gives a brief overview of our approach to address these challenges using on performance models.

Readers familiar with blocked algorithms and the influence of block sizes may skip the introduction to these concepts in [Section 1.1.1](#), and focus on our prediction approach in [Section 1.1.2](#) on [Page 9](#).

1.1.1 Motivation: Blocked Algorithms

Blocked algorithms are commonly used to exploit the performance of optimized BLAS Level 3 kernels¹ in other matrix operations, such as decompositions, inversions, and reductions. Every blocked algorithm traverses its input matrix (or matrices) in steps of a fixed *block size*; in each step of this traversal, it exposes a set of *sub-matrices* to which it applies a series of *updates*. Through these updates, it progresses with the computation and obtains a portion of the operation’s result; once the matrix traversal completes, the entire result is computed.

Example 1.1: Blocked algorithms for the Cholesky decomposition

[Figure 1.1](#) illustrates blocked algorithms for a simple yet representative operation: the lower-triangular Cholesky decomposition

$$\begin{matrix} \diagdown & \diagup \\ L & L^T \end{matrix} := \begin{matrix} \square \\ A \end{matrix}$$

of a symmetric positive definite (SPD) matrix $\begin{matrix} \square \\ A \end{matrix} \in \mathbb{R}^{n \times n}$ in lower-triangular storage (LAPACK: `dpotrfL`²). For this operation there exist three different blocked algorithms. Each algorithm traverses $\begin{matrix} \square \\ A \end{matrix}$ diagonally from the top-left to the bottom-right \diagdown and computes the Cholesky factor $\begin{matrix} \diagdown \\ L \end{matrix}$ in place. At each step of the traversal, the algorithm exposes the sub-matrices shown in [Figure 1.1a](#) and makes progress by applying the algorithm-dependent updates in [Figures 1.1b](#) to [1.1d](#). Before these updates, the sub-matrix A_{00} ,

¹ The BASIC LINEAR ALGEBRA SUBPROGRAMS (BLAS) form the basis for high-performance in dense linear algebra. See [Appendices A](#) and [B](#).

² [Appendix B](#) gives an overview of the BLAS and LAPACK routines used throughout this work. When specified, the subscripts indicate the values of the flag arguments, which identify the variant of the operation; e.g., in `dpotrfL` the `L` corresponds to the argument `uplo` indicating a lower-triangular decomposition.

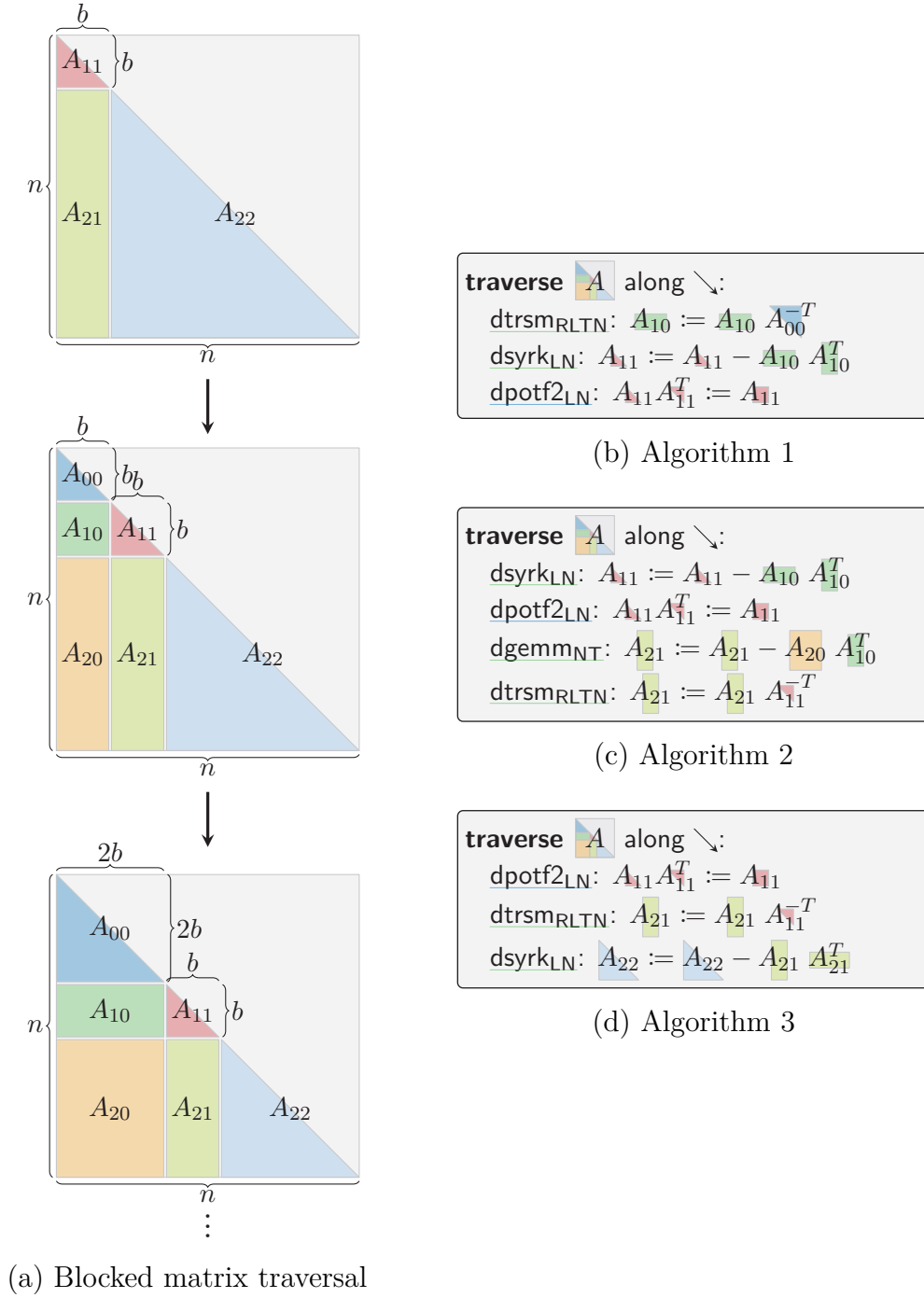


Figure 1.1: Blocked algorithms for the lower-triangular Cholesky decomposition.

1 Introduction

which in the first step is of size 0×0 , already contains a portion of the Cholesky factor L ; after the updates, the sub-matrices A_{10} and A_{11} also contain their portions of L , and in the next step become part of A_{00} . Once the traversal reaches the bottom-right corner (i.e., A_{00} is now of size $n \times n$), the entire matrix is factorized.

Blocked algorithms pose two *optimization challenges*:

- For each operation there typically exist several *alternative algorithms*, which are mathematically equivalent in exact arithmetic; however, even if such algorithms perform the same number of floating point operations, they may differ significantly in performance.
- For each algorithm, the *block size* influences the number of traversal steps and the sizes and shapes of the exposed sub-matrices, and thus the performance of the kernels applied to them.

What makes matters more complicated is that the optimal choice depends on various factors, such as the hardware, the number of threads, the kernel implementations, and the problem size.

Example 1.2: Performance of alternative algorithms

Figure 1.2 shows the performance of the three blocked Cholesky decompositions from Figure 1.1 with block size $b = 128$ and increasing problem size n on a 12-core HASWELL-EP E5-2680 v3³ with single- and multi-threaded OPENBLAS.

In both the single- and multi-threaded scenarios, algorithm 3 (—) is the fastest among the three alternatives for all problem sizes. On a single core and for problem size $n = 4152$, it is 27.40% and 12.89% faster than, respectively, algorithms 1 (—) and 2 (—), and it reaches up to 91.01% of the processor's theoretical peak performance (red line — at the top of the plot). On all 12 of the processor's cores, algorithm 3 (—) still reaches an efficiency of 69.70%, and outperforms algorithms 1 (—) and 2 (—) by, respectively, $5.21\times$ and $1.92\times$.

³ Appendix C provides an overview of the processors used throughout this work.

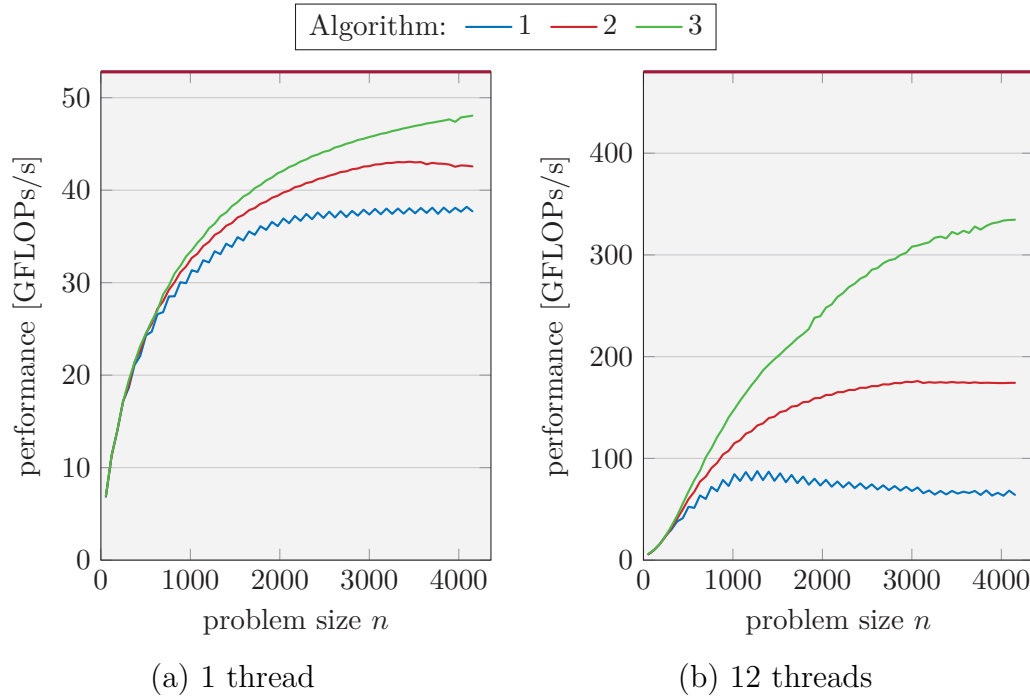


Figure 1.2: Performance of the three blocked Cholesky decomposition algorithms.

($b = 128$, HASWELL-EP E5-2680 v3, OPENBLAS, median of 10 repetitions)

Although algorithm 3 (—) is clearly the fastest in this and many other scenarios, LAPACK’s `dpotrfL` implements algorithm 2 (—).

For other operations, the choice becomes more complicated, since no single algorithm is the fastest for all problem sizes and scenarios. For instance, for the single-threaded inversion of a lower-triangular matrix $\mathbf{A} := \mathbf{A}^{-1}$, two different algorithms are the fastest for small and large matrices; with the performance differing by up to 13% in either direction (Section 4.5.2).

Example 1.3: Influence of the block size on performance

Let us consider the blocked Cholesky decomposition algorithm 3 (— in Figure 1.2) with fixed problem sizes $n = 1000, 2000, 3000$, and 4000 and varying block size b . Figure 1.3 presents the performance of these algorithm executions for 1 and 12 threads on the HASWELL-EP E5-2680 v3 using OPENBLAS: Single-threaded, the optimal block size increases from $b = 96$

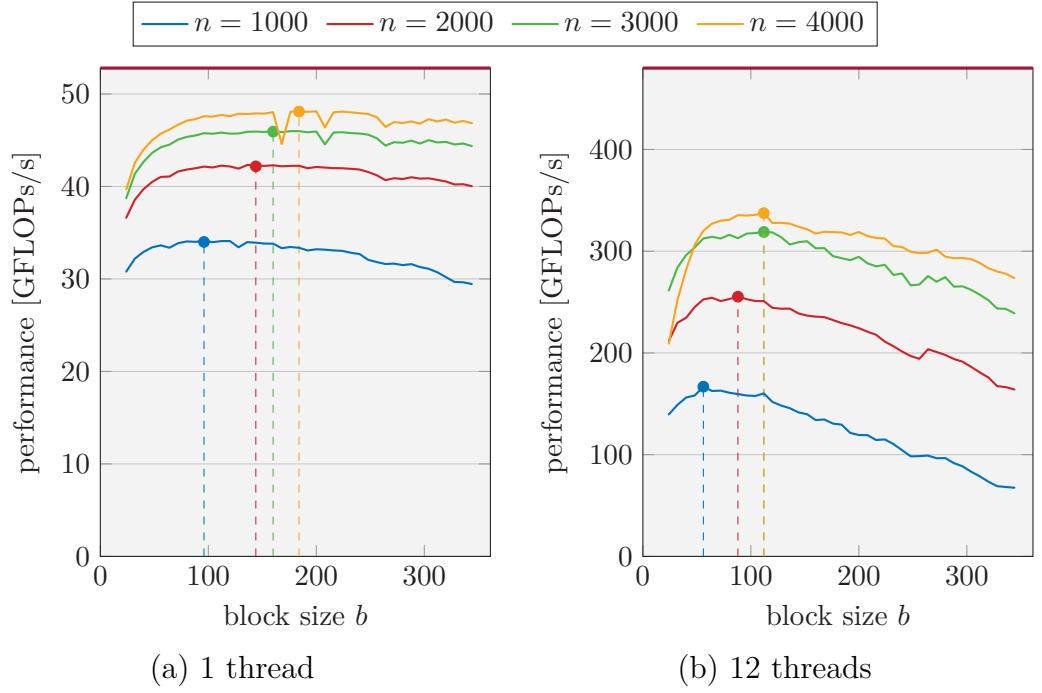


Figure 1.3: Performance of the blocked Cholesky decompositions algorithm 3 for varying block sizes.

(HASWELL-EP E5-2680 v3, OPENBLAS, median of 10 repetitions)

for $n = 1000$ to $b = 184$ for $n = 4000$. On 12 cores, on the other hand, the performance is less smooth and the optimal choices for b are between 56 and 112.

Figure 1.3 demonstrates the importance of selecting the block size dynamically: If we use $b = 184$, which is optimal for $n = 4000$ on one core, for $n = 1000$ on 12 cores we only reach 77.62% of the algorithm's optimal performance. On the other hand, LAPACK's default block size $b = 64$ (which is close to the optimal $b = 56$ for $n = 1000$ on 12 cores) would reach 95.95% of the optimal single-threaded performance for $n = 4000$.

1.1.2 Prediction through Performance Models

Naturally, both the best algorithm and its optimal block size for a given scenario (operation, problem size, hardware, kernel library, multi-threading) can be determined through exhaustive performance measurements; however, this is extremely time consuming and thus often impractical. Instead we aim to determine the optimal configuration *without executing* any of the alternative algorithms. For this purpose, we use the hierarchical structure of blocked algorithms: Their entire computation is performed in a series of calls to a few kernel routines; hence, by accurately estimating the runtime of these kernels, we can predict an entire algorithm’s runtime and performance.

In order to estimate the kernel runtimes, let us study how these kernels are used: In each algorithm execution, the same set of kernels is invoked repeatedly—once for each step of the blocked matrix traversal. Each invocation, however, works on operands of different size depending on the progress of the algorithms’ traversal, the input problem size, and the block size. In short, we need to estimate the performance of only a few kernels, yet with potentially wide ranges of operand sizes.

Our solution is *performance modeling*, as detailed in [Chapter 3](#): Based on a detailed study of how a kernel’s arguments (i.e., flags, operand sizes, etc.) affect its performance, we design performance models in the form of piecewise multivariate polynomials. These models are generated automatically once for each hardware and software setup and subsequently provide accurate performance estimates at a tiny fraction of the kernel’s runtime.

Using such estimates, we *predict* the *performance* of blocked algorithms, as presented in [Chapter 4](#). These fast predictions prove to be highly accurate, and allow us to both rank the blocked algorithms for a given operation according to their performance, and find near-optimal values for the algorithmic block sizes.

While our models yield accurate performance estimates for individual kernel executions, they do not capture the performance influence of *caching* between kernels. Prior to the invocation of each compute kernel in an algorithm, typically only a portion of its operands are in cache, and loading operands from main

memory increases the kernel runtime. [Chapter 5](#) investigates how caching effects can be accounted for in blocked algorithms, and attempts to combine pure in- and out-of-cache estimates into more accurate prediction. However, while the results look promising on a rather old HARPertown E5450, the analysis reveals that on modern processors the effect caching on kernel performance is so complex that accounting for it in algorithm-independent performance models to further improve our prediction accuracy is infeasible.

1.2 Micro-Benchmarks for Tensor Contractions

Tensor contractions play an increasingly important role in various scientific computations, such as machine learning [\[13\]](#), general relativity [\[62, 64\]](#), and quantum chemistry [\[21, 34\]](#). Following a brief introduction to BLAS-based tensor contraction algorithms and their performance in [Section 1.2.1](#), [Section 1.2.2](#) gives an overview of how predictions based on micro-benchmarks are used to rank alternative algorithms for a given contraction.

1.2.1 Motivation: Tensor Contraction Algorithms

Computationally, tensor contractions are generalizations of matrix-vector and matrix-matrix products to operands of higher dimensionality. While BLAS covers contractions of up to two-dimensional operands (i.e., matrices), there are no equivalently established and standardized high-performance libraries for general tensor contractions. Fortunately, just as a matrix-matrix products can be decomposed into sequences of matrix-vector products, higher dimensional tensor contractions can be cast in terms of matrix-matrix or matrix-vector kernels. (A broader overview of alternative approaches is given in [Section 1.3.4](#).)

Example 1.4: Tensor contraction algorithms

Let us consider the contraction $C_{abc} := A_{ai}B_{ibc}$ (in Einstein notation), which

1.2 Micro-Benchmarks for Tensor Contractions

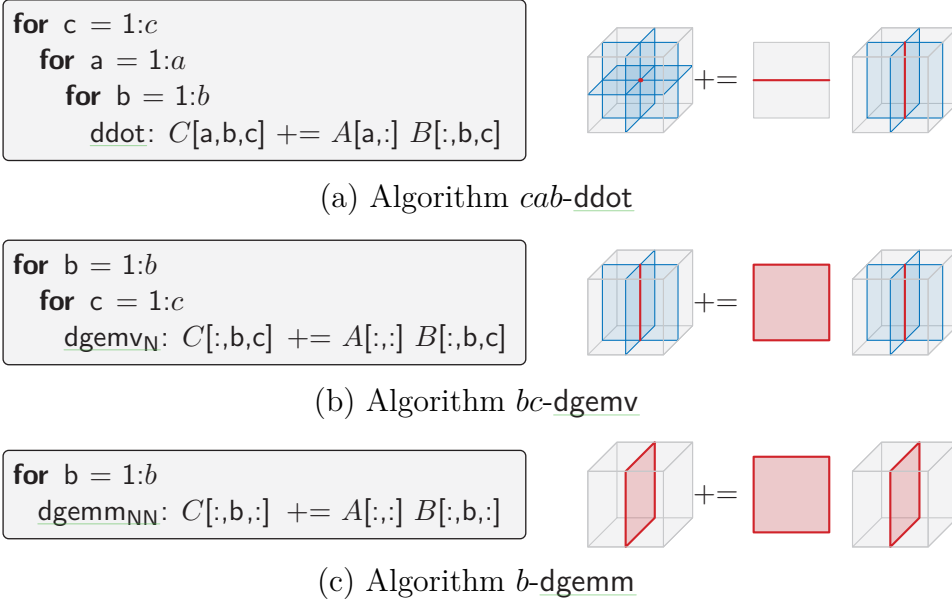


Figure 1.4: Sample of algorithms for the tensor contraction $C_{abc} := A_{ai}B_{ibc}$. All slicings are visualized in **blue**; the kernel operands (the intersections) are in **red**. The name of each algorithm stems from the dimensions its **for**-loops index and its BLAS kernel.

is visualized as follows:

$$\begin{array}{c} a \\ \text{---} \\ \text{Cube } C \\ \text{---} \\ b \end{array} \begin{array}{c} c \\ \text{---} \\ \text{---} \\ \text{---} \end{array} := \begin{array}{c} a \\ \text{---} \\ \text{Cube } A \\ \text{---} \\ i \end{array} \begin{array}{c} i \\ \text{---} \\ \text{Cube } B \\ \text{---} \\ b \end{array} \begin{array}{c} c \\ \text{---} \\ \text{---} \\ \text{---} \end{array} .$$

The entries $C[a,b,c]$ of the resulting three-dimensional tensor $C \in \mathbb{R}^{a \times b \times c}$ are computed as

$$\forall a \forall b \forall c : C[a,b,c] := \sum_i A[a,i] B[i,b,c] .$$

As further described in [Section 6.1](#), this contraction can be performed by a total of 36 alternative algorithms, each consisting of one or more **for**-loops with a single BLAS kernel at its core. Three examples of such algorithms using BLAS Level 1, 2, and 3 kernels are shown in [Figure 1.4](#). These

1 Introduction

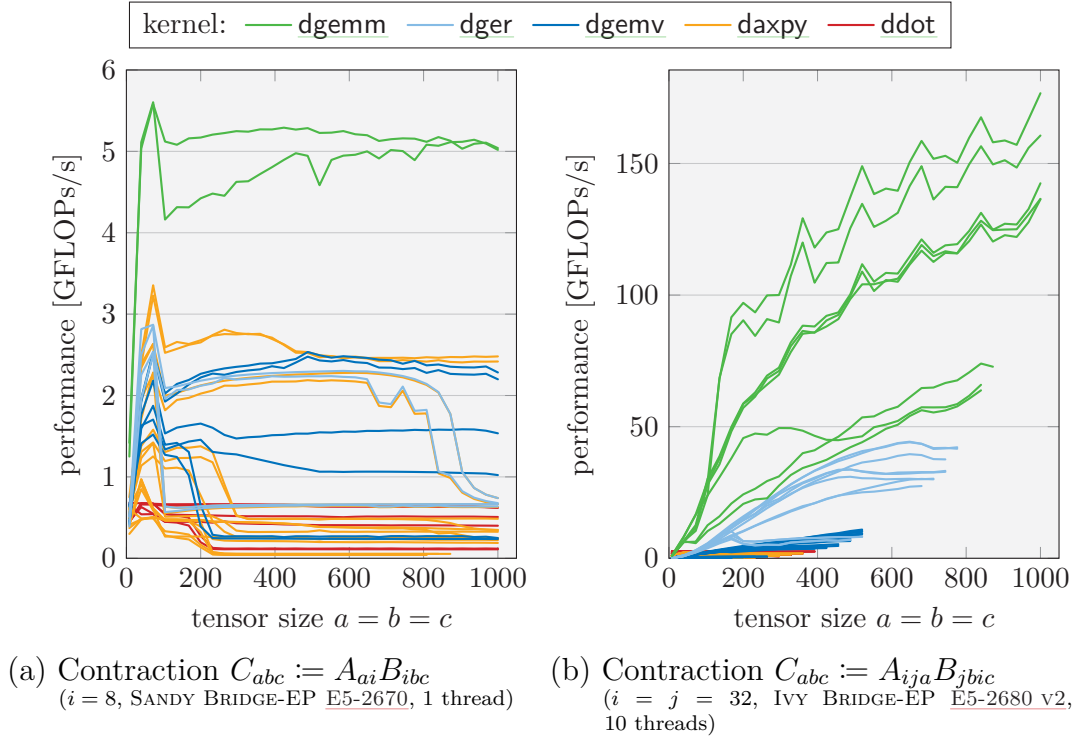


Figure 1.5: Performance of tensor contraction algorithms.

(OPENBLAS, median of 10 repetitions)

algorithms use MATLAB’s “:” slicing notation⁴ to access matrices and vectors within the tensors A , B , and C ; the resulting operand shapes within the tensors passed to the BLAS kernel are shown alongside the algorithms.

Each tensor contraction can be computed via BLAS kernels through many—even hundreds—of algorithms, each with its own performance behavior. The *optimization challenge* of identifying the fastest among such a set of *alternative algorithms* is especially difficult due to the in practice commonly encountered *skewed dimensions* (i.e., one or more dimensions are extremely small) for which most BLAS implementations are typically not optimized.

⁴ The index “:” in a tensor refers to all elements along that dimension, e.g., $A[a,:]$ is the a -th row of A .

Example 1.5: Performance of contraction algorithms

Let us consider the tensor contraction $C_{abc} := A_{ai}B_{ibc}$ from [Example 1.4](#) with tensors $A \in \mathbb{R}^{n \times 8}$, $B \in \mathbb{R}^{8 \times n \times n}$, and thus $C \in \mathbb{R}^{n \times n \times n}$; for $n = 100$, this can be visualized as follows:

$$\begin{array}{c} a \\ \text{---} \\ \text{---} \\ \text{---} \\ b \end{array} \text{---} C \text{---} \begin{array}{c} c \end{array} := \begin{array}{c} a \\ \text{---} \\ \text{---} \\ \text{---} \\ i \end{array} A \text{---} \begin{array}{c} i \\ \text{---} \\ \text{---} \\ \text{---} \\ b \end{array} B \text{---} \begin{array}{c} c \end{array} .$$

[Figure 1.5a](#) presents the performance of all 36 algorithms for this contraction on a HARPETOWN [E5450](#) with single-threaded OPENBLAS. While the two **dgemm**-based algorithms ([—](#)) are clearly faster than the others, they differ in performance by up to 23.32%; with other kernels the difference are even more extreme, exceeding a factor of 60 for the **daxpy**-based algorithms ([—](#)).

[Figure 1.5b](#) showcases the performance of algorithms for the more complex contraction $C_{abc} := A_{ija}B_{jbic}$ on all 10 cores of an IVY BRIDGE-EP [E5-2680 v2](#) using multi-threaded OPENBLAS. In this scenario, the performance of the **dgemm**-based algorithms alone differs by up to $3\times$.

One could argue that only **dgemm**-based algorithms are viable candidates to achieve the best performance; while for the most part this observation is true, due to skewed dimensions, even the performance of only these algorithms can differ dramatically. Furthermore, some contractions (e.g., $C_a := A_{iaj}B_{ji}$) cannot be implemented via **dgemm** in the first place. Therefore, we aim at the accurate prediction of any BLAS-based contraction, irrespective of which kernel is used.

1.2.2 Prediction through Micro-Benchmarks

At first sight the situation seems similar to the selection of blocked algorithms: We want to avoid exhaustive performance measurements and select the best algorithm *without executing* any of the alternatives; our strategy is once again to predict each algorithm’s performance by estimating its invoked kernel’s runtime.

1 Introduction

However, while performance models accurately estimates the performance of such kernels for many operand sizes, they perform rather poorly for operations with skewed dimensions: For extremely thin or small operands, BLAS kernels exhibit strong size-dependent performance fluctuations, which are impractical to capture and represent in performance models.

While we cannot rely on performance models, analyzing the structure of tensor contraction algorithms suggests a different approach: In contrast to blocked algorithms, a contraction algorithm performs its entire computation in a series of calls to a *single BLAS kernel* of with operands of *fixed size*. Based on this observation, we estimate the performance of such calls by constructing a small set of *micro-benchmarks* that executes the kernel only a few times, and thus performs only a fraction of the algorithm’s computation. Since memory locality plays an especially important role in contractions with skewed dimensions, we carefully recreate the stat of the processor’s caches within the micro-benchmarks to time the kernel in conditions analogous to those in the actual algorithm.

Based on such micro-benchmarks, we can predict the total runtime of contraction algorithms for tensors of various shapes and sizes. These predictions reliably single out the fastest algorithm from a set of alternatives several orders of magnitude faster than a single algorithm execution.

1.3 Related Work

This overview of related research is structured as follows: [Section 1.3.1](#) summarizes the history and state-of-the-art of dense linear algebra (DLA) libraries and algorithms, [Section 1.3.2](#) addresses performance measurements and profiling tools, [Section 1.3.3](#) presents performance modeling and prediction efforts, and [Section 1.3.4](#) discusses developments in high-performance tensor contractions.

1.3.1 Dense Linear Algebra Libraries and Algorithms

We begin with a brief history of the fundamental DLA libraries BLAS and LAPACK and prominent implementations in [Section 1.3.1.1](#). We then focus on blocked algorithms and their tuning opportunities in [Section 1.3.1.2](#), and finally give an overview of alternative algorithms and libraries for distributed-memory and accelerator hardware in, respectively, [Sections 1.3.1.3](#) and [1.3.1.4](#).

1.3.1.1 BLAS and LAPACK

The development of standardized DLA libraries began in 1979 with the inception of the BASIC LINEAR ALGEBRA SUBPROGRAMS (*BLAS*) [\[63\]](#), a FORTRAN interface specification for, initially, various “Level 1” scalar and vector operations. It was subsequently extended to kernels for “Level 2” matrix-vector [\[40\]](#) and “Level 3” matrix-matrix [\[39\]](#) operations in, respectively, 1988 and 1990. The aim of the BLAS specification is to enable performance portable applications: DLA codes reach high performance on different hardware by using architecture-specific BLAS implementations. Although computer architectures have evolved dramatically in the last 40 years, this principle of performance portability is still at the core of all current DLA libraries.

The BLAS specification is accompanied by a reference implementation [\[96\]](#) that, while fully functional and well documented, is deliberately simple and thus slow; to reach high performance, users instead link with optimized *BLAS implementations*. The oldest *open-source* implementation still in use is the AUTOMATICALLY TUNED LINEAR ALGEBRA SOFTWARE (ATLAS) [\[85, 86, 87, 95\]](#), first released in 1997; this auto-tuning-based library’s main proficiency is to yield decent performance on a wide range of hardware platforms with little developer and user effort. The first major open-source implementation hand-tuned for modern processors with cache hierarchies was GOTOBLAS [\[50, 51, 107\]](#). It reaches up to around 90 % of a processor’s peak floating-point performance for both sequential and multi-threaded Level 3 kernels and good bandwidth-bound performance for Level 1 and 2 operations. After GOTOBLAS’s discontinuation in 2010, its code-base and approach were picked up and

1 Introduction

extended to more recent processors in the OPENBLAS library [118], which is currently the fastest open-source implementation for many architectures. Also inspired by GOTOBLAS’s approach is the fairly recent BLAS-LIKE LIBRARY INSTANTIATION SOFTWARE (BLIS) [73, 81, 82, 97], an open-source framework that provides optimized kernels for basic DLA operations, such as the BLAS, based on one hand-tuned micro-kernel per architecture.

In addition to open-source implementations, many hardware *vendors* maintain and distribute their own high-performance *BLAS*, e.g., INTEL’s MATH KERNEL LIBRARY (MKL) [109], APPLE’s framework ACCELERATE [105], and IBM’s ENGINEERING AND SCIENTIFIC SUBROUTINE LIBRARY (ESSL) [104].

BLAS forms the basis for DLA libraries covering more advanced operations. The earliest library built on top of first BLAS Level 1 and later Level 2 was LINPACK [38, 115], a package of solvers for linear equations and least-squares problems from the 1970s and 1980s. LINPACK together with EISPACK [47, 101], a collection of eigenvalue solvers, was superseded by the LINEAR ALGEBRA PACKAGE (*LAPACK*) [16, 111] in 1992. LAPACK has since been extended with new features and algorithms, and is still under active development. Just like BLAS, LAPACK functions as a de-facto standard interface specification for many advanced DLA operations; libraries such as OPENBLAS and MKL adopt its interface and provide tuned implementations of various routines.

For more details on BLAS and LAPACK, and their kernels and implementations used throughout this work, see [Appendix B](#).

1.3.1.2 Blocked Algorithms

LAPACK uses *blocked algorithms* for most of its dense operations. The core idea behind these algorithms is to leverage a processor’s cache hierarchy by increasing the spacial and temporal locality of operands, as well as casting most of an operation’s computation in terms of BLAS Level 3 kernels. As a result, complex operations can reach performance levels close to the hardware’s theoretical peak.

However, for each operation, there typically exist multiple *alternative blocked*

algorithms, of which LAPACK offers only one, but not always the fastest. The alternative algorithms for a given operation can be derived from its mathematical formulation systematically [24] and automatically [44, 45]. Based on these principles, LIBFLAME [93, 94, 112] offers many alternative algorithms for each operation, and for several operations provides more efficient default algorithms than LAPACK. In this work we consider LIBFLAME’s blocked algorithms for various operations, and aim to predict which of them is most efficient for given scenarios.

Another caveat of blocked algorithms is their *block sizes*, which need to be carefully *tuned* to maximize performance. Since this is a well-known aspect of blocked algorithms [23, 84], LAPACK encapsulates and exposes all its tuning parameters in `ilaenv`, a central routine that is used to configure the library at compile time; for many operations the block sizes used by LAPACK’s reference implementation of `ilaenv` (64 for most algorithms) have been too small on recent hardware for quite some time. Although the necessity of optimizing block sizes is well understood and taken care of by implementations such as MKL, it remains non-trivial, and in fact few end-users and application-developers are aware of it. The automated model-based optimization of the block size for blocked algorithms is the second major goal of this work.

1.3.1.3 Alternatives to Blocked Algorithms

An alternative to blocked algorithms is *recursive algorithms*, which avoid both the algorithm selection and block-size optimization. They are also known as “cache oblivious” algorithms [27, 46] since they minimize the data-movement between cache levels [53]. Recursion has been suggested for many DLA operations, such as the LU decomposition [49, 77], the Cholesky decomposition [83], triangular matrix inversion [61], two-sided linear systems [19], tall-and-skinny QR factorization [42], and Sylvester-type equation solvers [60, 120].

However, since no readily-available recursion-based library comparable to LAPACK existed, we developed the RECURSIVE LAPACK COLLECTION (*RELAPACK*) [1, 121]. RELAPACK provides recursive implementations for

48 LAPACK routines, and outperforms not only the reference implementation but in many cases also optimized libraries such as OPENBLAS and MKL.

A second alternative to blocked algorithms tailored to shared-memory machines are task-based *algorithms-by-blocks*, also known as “block algorithms” or “tiled algorithms”. However, these algorithms not only introduce a specialized storage scheme of matrices “by block”, but also require custom task scheduling mechanisms. Implementations of such schedulers include QUARK [91] as part of PLASMA [14], DAGuE [26], SMPs [18], and SUPERMATRIX [33].

1.3.1.4 Distributed-Memory and Accelerators

Distributed-memory machines and super-computers are indispensable for large-scale DLA computations. The first noteworthy extension of the BLAS and the LAPACK to this domain was the SCALABLE LINEAR ALGEBRA PACKAGE (SCALAPACK) [25, 122], written in FORTRAN and based on BLAS, LAPACK, and the MESSAGE PASSING INTERFACE (MPI). However, SCALAPACK is only sparingly updated (last in 2012), and, instead, the state of the art for distributed-memory DLA is ELEMENTAL [71, 103], an actively developed C++ library, based on LIBFLAME’s methodology in and object-oriented and templated programming techniques.

Since *accelerators* such as XEON-PHI coprocessors and graphics processors lend themselves well to compute-intensive operations, they are a natural target for DLA codes. While some classic BLAS implementations such as ATLAS, BLIS, and MKL, can be used on the x86-based XEON PHIs, separate libraries are required for graphics processors: NVIDIA’s CUBLAS [99] provides high-performance BLAS kernels for CUDA-enabled graphics cards, and CLBLAS [98] targets OPENCL-capable devices. Furthermore, MATRIX ALGEBRA ON GPU AND MULTICORE ARCHITECTURES (MAGMA) [78, 116] targets BLAS and LAPACK operations on heterogeneous machines (e.g., CPU + GPU).

1.3.2 Performance Measurements and Profiling

Runtime measurements of both application codes and algorithms are crucial in the investigation of performance behaviors, bottlenecks, as well as optimization and tuning in general; hence, numerous tools facilitate such measurements. Simple timers are accessible in virtually any language and environment: e.g., `time` in Unix, `rdtsc` in x86 assembly, `gettimeofday()` in C, `omp_get_wtime()` in OPENMP, `tic` and `toc` in MATLAB, and `timeit` in PYTHON. Several more advanced tools *profile* executions of functions and communications in applications by tracing or sampling: e.g., GPROF [52, 106], VAMPIR [128], TAU [72, 125], SCALASCA [48, 123], and INTEL’s VTUNE [110]. While such tools are invaluable in the performance analysis of application codes, their generality makes them somewhat unwieldy for our purposes of investigating DLA kernel performance. Therefore, we designed EXPERIMENTAL LINEAR ALGEBRA PERFORMANCE STUDIES (*ELAPS*) [3, 102], a framework for performance measurements and analysis of DLA routines and algorithms, further detailed in Section 2.2.

1.3.3 Performance Modeling and Predictions

Predicting and modeling application performance is an important aspect of high-performance computing, and the term “performance modeling” is used to describe many different techniques and approaches. This section gives a brief overview of such approaches with focus on methods for DLA algorithms.

The well-established *Roofline model* [88] does not predict performance, but relates an algorithm’s attained performance to the hardware’s potential: As detailed in A.5.3, it allows to evaluate an execution’s resource efficiency by relating its algorithm’s arithmetic intensity and int performance relative to the hardware’s peak main-memory bandwidth and floating-point performance. It has been applied, implemented, and extended in numerous publications, such as [59, 65, 70]. Notably, Benner et al. use the roofline model (the arithmetic intensity in particular) to optimize the block size for a blocked matrix inversion algorithm [23].

1 Introduction

Model-based performance tuning of BLAS implementations was suggested for both ATLAS [92] and BLIS [66], showing that near-optimal BLAS performance can be reached without measurement-based auto-tuning: Instead they, e.g., select blocking sizes according to the BLAS implementation and the target processor’s cache sizes. Note that these approaches are used to tune BLAS kernels, and do not actually predict their performance; hence they cannot serve as a basis for our predictions.

Previous work in our research group by Iakymchuk et al. constructed accurate *analytical performance models* for small DLA kernels [56, 57]. These models target problems that fit within a HARPETOWN E5450’s last-level cache (L2), and are based on the number of memory-stalls and arithmetic operations as well as their overlap incurred by specific kernel implementations. As such, they require not only a deep understanding of the processor architecture, but also a detailed analysis of the kernel implementation. While the resulting models yield accurate predictions within a few percent of reference measurements, they are not easily extended to larger problems and other operations. Therefore, this work instead considers automatically generated, measurement-based models.

Alonso et al. construct *piecewise runtime and energy models*—somewhat similar to those presented in this work—for the BLIS implementations of `dgemm` and `dtrsm` [15] on a SANDY BRIDGE-EP E5-2620. However, their approach is based on extensive knowledge of BLIS [66], and their models only represent one degree of freedom (by considering only square matrices or operations on panel matrices with fixed width/height). Their average runtime model accuracy for `dgemm` and `dtrsm` is, respectively, 1.5 % and 4.5 %, with local errors of up to, respectively, 4.5 % and 7 %. Catalán et al. extend this work to multi-threaded `dgemm`, `dtrsm`, and `dsyrk` in order to predict the performance of a blocked Cholesky decomposition algorithm with fixed block size [32]; their average runtime prediction errors are 3.7 % and 2.4 %, depending on the parallelization within BLIS. In contrast to these publications, the modeling framework presented in this work, which was developed around the same time, is fully automated, applicable to any BLAS- or LAPACK-like routine, not

limited to one implementation and hardware, and offers models with multiple degrees of freedom.

In a separate effort Yamamoto constructs measurement-based, yet hardware- and *implementation-independent models* in the form of a series of univariate polynomials (one kernel argument is represented by the polynomial, the other varied in the series) for several BLAS Level 3 kernels [89, 90]. These models are used to predict the performance of both a blocked reduction to tridiagonal form [89] and a blocked multishift QR algorithm [90]. The resulting prediction error on an unspecified AMD OPTERON is reported to be below 10% for the single-threaded tridiagonalization, and is on average around 10% for the QR algorithm using multi-threaded BLAS. In contrast, the more general piecewise models proposed in this work yield considerable smaller prediction errors for various blocked algorithms.

Several research projects model the performance of *distributed-memory* applications. A general purpose approach by Calotoiu et al. builds basic performance models for kernels in application codes based on performance profiling [30, 31], allowing to investigate the complexity and scalability of application components. In the field of distributed-memory DLA, most modeling efforts target SCALAPACK using domain-specific knowledge through, e.g., polynomial fitting [67] or hierarchical modeling of kernels [36].

1.3.4 Tensor Contractions

Tensor contractions are at the core of scientific computations, such as machine learning [13], general relativity [62, 64], and quantum chemistry [21, 34]. Since generally speaking such contractions are high-dimensional matrix-matrix multiplications, they are closely related to BLAS Level 3 operations, and in fact most contractions can be cast in terms of one or more calls to `dgemm`, either by adding loops or transpositions; this is implemented in many frameworks, such as the TENSOR CONTRACTION ENGINE (TCE) [54, 126], the CYCLOPS TENSOR FRAMEWORK (CTF) [74, 100], the MATLAB TENSOR TOOLBOX [17, 117], and LIBTENSOR [43, 113].

1 Introduction

In contrast to these implementations, which rely on a single algorithm for each contraction (potentially selected through heuristics), previous work in our group by Di Napoli et al. investigated the automated generation of all alternative BLAS-based algorithms [37]. [Chapter 6](#) picks up this work and presents a performance prediction framework for such algorithms that allow to automatically identify the fastest algorithm [6].

More recent and ongoing work in our group by Springer et al. attempts to go break the barrier between contraction algorithms and `dgemm` implementations. Following the structured design of BLIS [81], they propose code generators that provide high-performance algorithms tailored to specific contraction problems that reach close to optimal performance [75]. Their tools construct numerous alternative implementations, and identify the fastest through a combination of heuristics and micro-benchmarks.

2 Performance Effects and Measurements

This work is concerned with predicting the performance of dense linear algebra routines and algorithms through measurement-based performance models and micro-benchmarks. To fully focus on modeling and prediction in the following chapters, we here establish how accurate runtime measurements are obtained, and address common influences on such measurements and their effects. Furthermore, we present a performance measurement tool and framework tailored to dense linear algebra routines that we developed to serve as the foundation for the experiments, models, and benchmarks throughout this work.

In detail, this chapter covers the following material:

- [Section 2.1](#) presents common effects observed when measuring the runtime of dense linear algebra routines. In particular, it addresses *library initialization overhead*, *fluctuations* (e.g., due to *system noise* and *varying processor frequency*), *thread pinning*, *NUMA effects*, and *caching*.
- [Section 2.2](#) introduces the *ELAPS Framework* that evolved from the performance measurement tools developed for this work. ELAPS provides the *SAMPLER*, a low-level tool for measurements of BLAS- and LAPACK-like dense linear algebra routines, as well as a *PYTHON framework* with a graphical user interface and various utility functions to set up experiments and process their results.

Additionally, for readers new to performance studies, [Appendix A](#) provides an introduction into the terminology and concepts of topics such as computational workload, timings, performance, hardware limitations, and efficiency.

	OPENBLAS	BLIS	MKL	reference
1st <code>dgemm</code>	1.10 ms	1.32 ms	8.14 ms	37.96 ms
2nd <code>dgemm</code>	0.90 ms	0.95 ms	0.86 ms	37.93 ms
overhead	0.20 ms	0.38 ms	7.28 ms	0.04 ms

Table 2.1: BLAS library initialization overhead for two identical `dgemmNN`.
 ($m = n = k = 200$, SANDY BRIDGE-EP E5-2670, 1 thread)

2.1 Performance Effects for Dense Linear Algebra Kernels

At the core of any study on performance are accurate runtime measurements. However, while in principle, timing a computation is as simple as “start timer—compute—stop timer”, obtaining reliable and stable timings is not trivial. In this section, we present the most relevant effects and influences on measurements of dense linear algebra routines; in particular, we address initialization overhead (Section 2.1.1), different types of fluctuations (Section 2.1.2), thread pinning (Section 2.1.3), NUMA effects (Section 2.1.4) and caching (Section 2.1.5).

2.1.1 Library Initialization Overhead

Many high-performance dense linear algebra libraries, such as optimized implementations of BLAS and LAPACK, initialize (e.g., detect hardware, allocate buffers, etc.) the first time one of their kernels is invoked. These *initializations* imply an *overhead* that can significantly increase the first library invocation’s runtime.

Example 2.1: Library initialization overhead

Table 2.1 presents the runtime of two consecutive matrix-matrix multiplications $C_1 := A_1 B_1 + C_1$ and $C_2 := A_2 B_2 + C_2$ (`dgemmNN`) with disjoint $A_1, A_2, B_1, B_2, C_1, C_2 \in \mathbb{R}^{200 \times 200}$ on a SANDY BRIDGE-EP E5-2670 with single-threaded OPENBLAS, BLIS, and MKL; the two calls to `dgemmNN` are the first and only invocations of BLAS in program.

The timings show that the libraries have substantially different overheads:

- The reference BLAS implementation has a negligible overhead but is around $40\times$ slower than the optimized libraries.
- OPENBLAS and BLIS are optimized for the SANDY BRIDGE, and when first invoked, these libraries perform some initializations, such as allocating auxiliary buffers, that introduce an overhead of, respectively, 0.20 ms and 0.38 ms.
- In addition to the allocation of auxiliary buffers, MKL dynamically detects the processor architecture to accordingly select optimized kernels. Hence it has by far the largest overhead of 7.28 ms, which dominates its first invocation's runtime.

Since we mostly use optimized libraries such as OPENBLAS, BLIS, and MKL, we counter the initialization overhead by simply preceding any set of measurements with an unrelated kernel invocation.

2.1.2 Fluctuations

Once the initialization overhead is overcome, repeated timings of the same kernel on the same data may still exhibit significant *performance fluctuations*. Such fluctuations can be caused by a variety of effects, such as background applications and system noise (Section 2.1.2.1), INTEL TURBO BOOST (Section 2.1.2.2), or other changes in processor frequency (Section 2.1.2.3).

2.1.2.1 Background and System Noise

The potentially most disturbing, yet also quite easily avoidable source of fluctuations are other *background processes* competing for the processor's resources.

Example 2.2: Influence of background noise

Figure 2.1 presents the runtime of 1000 repetitions of the matrix-matrix multiplication $\boxed{C} := \boxed{A} \boxed{B} + \boxed{C}$ (dgemm_{NN}) with $\boxed{A}, \boxed{B}, \boxed{C} \in \mathbb{R}^{100 \times 100}$ on a BROADWELL i7-5557U (as part of MACBOOK PRO with APPLE's

2 Performance Effects and Measurements

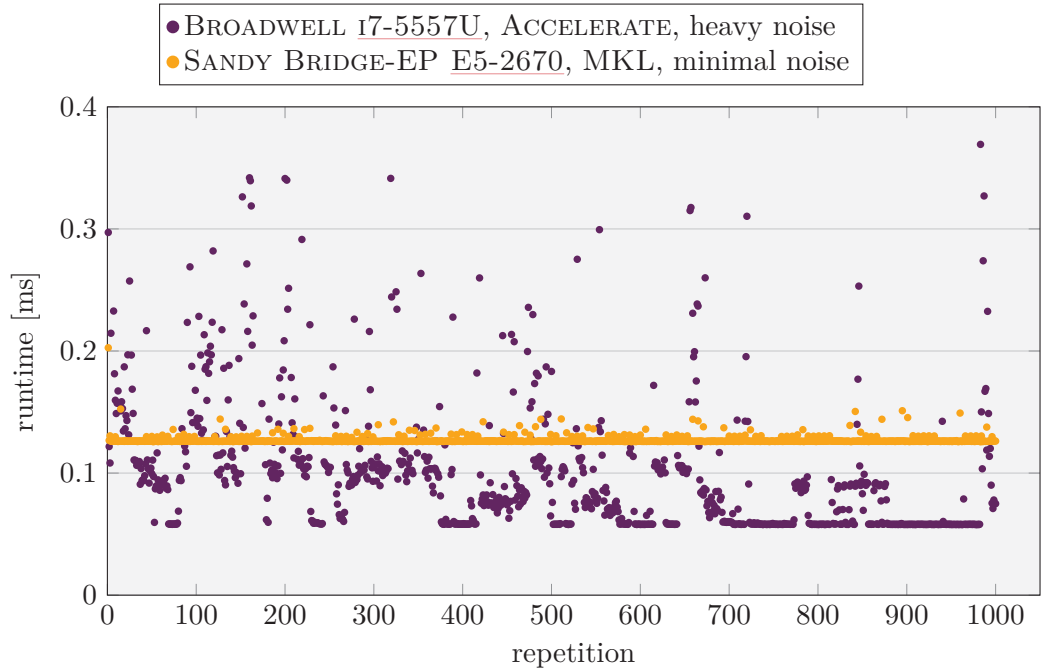


Figure 2.1: Runtime fluctuations `dgemmNN` caused by background processes and system noise.

($m = n = k = 100$, 1 thread)

framework ACCELERATE and a SANDY BRIDGE-EP [E5-2670](#) (as part of RWTH’s computing cluster) with MKL.

On the BROADWELL [i7-5557U](#) (●) with various other applications running in the background (e.g., browser and music player), the fluctuations are enormous: The measurement standard deviation is over $4\times$ the mean runtime. On the SANDY BRIDGE-EP [E5-2670](#) (●) with no other user applications running during measurements, the fluctuations are already much smaller at 2.36 % of the average time. For larger problem sizes, the fluctuations are considerably smaller, and quickly fall below 0.1 %.

While these type of fluctuations can be avoided to some extent by ensuring that no other applications run during measurements, they cannot be avoided altogether even with exclusive access to dedicated high-performance hardware—the remaining fluctuations are known as *system noise* [80]. Hence, for our

experiments, models, and micro-benchmarks all our measurements are repeated at least five times and *summary statistics* of the runtime (or performance) are presented, such as the minimum or median.

2.1.2.2 INTEL TURBO BOOST

Compute-bound dense linear algebra computations, such as BLAS Level 3 and LAPACK-level routines, benefit directly from increased processing frequencies. Therefore, they usually trigger INTEL TURBO BOOST and constantly run at the maximum turbo frequency if possible. Since this frequency cannot be sustained indefinitely on most machines, the processor frequency is eventually lowered and henceforth fluctuates to keep the hardware within its power and thermal limits.

Example 2.3: TURBO BOOST

Figure 2.2 presents the runtime of repeated matrix-matrix multiplications $C := A B + C$ (`dgemmNN`) with $A, B, C \in \mathbb{R}^{1300 \times 1300}$ alongside the processor's temperature and frequency¹ on both cores of a BROADWELL I7-5557U with multi-threaded ACCELERATE; in this experiment, no other resource intensive programs run in the background.

In the beginning, the processor is at a cool 53 °C (—) and each `dgemmNN` takes about 60 ms (●) at the maximum turbo frequency of 3.4 GHz (—). The processor temperature increases steadily up to 105 °C around repetition 200 (12s into the experiment); at this point the frequency is reduced and continuously adjusted between 3 GHz and 3.2 GHz such that this temperature threshold is not exceeded. This change in frequency, as well as its fluctuations towards the end have a direct effect on the `dgemmNN`'s runtime: It increases by about 10 % to roughly 67 ms.

The behavior of TURBO BOOST depends enormously on the computation environment: While on a work-station or laptop the processor temperature increases rapidly and the maximum turbo frequency is not sustained for long, on dedicated high-performance compute clusters, efficient cooling allows for the

¹ Obtained through the INTELPOWER GADGET.

2 Performance Effects and Measurements

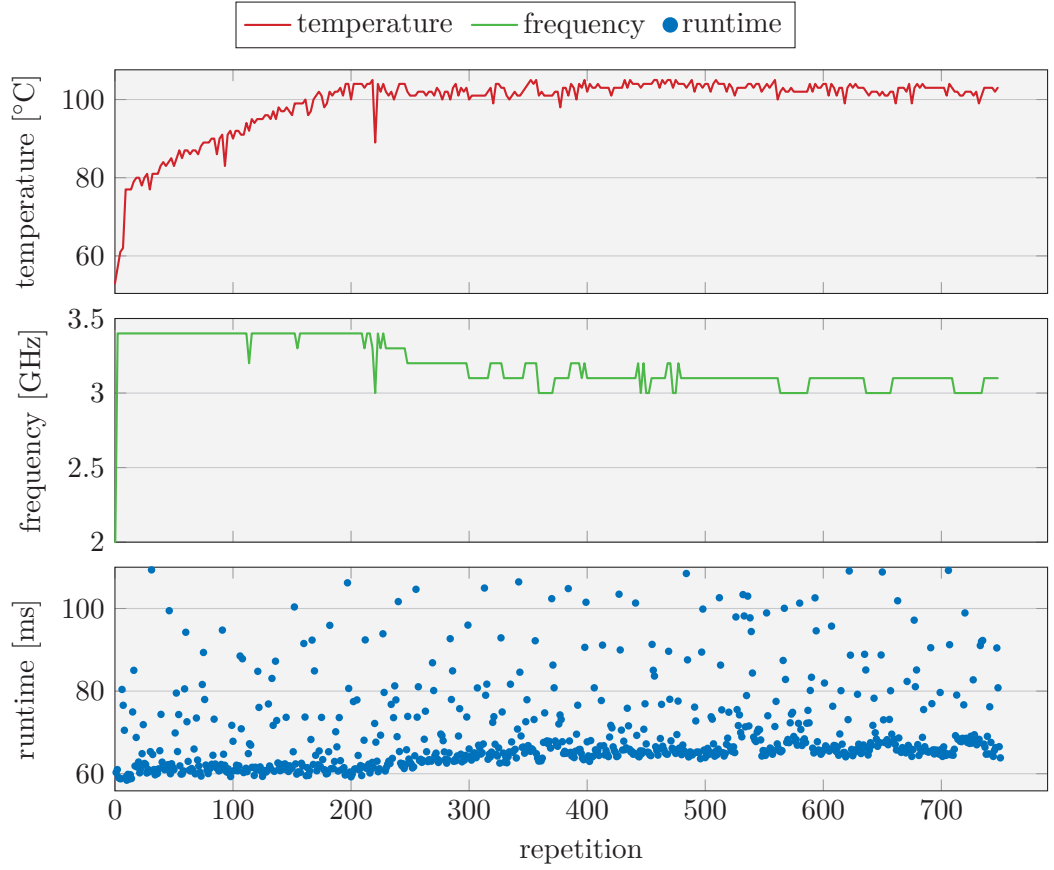


Figure 2.2: Effect of TURBO BOOST on the runtime of `dgemmNN`.
(Note: y -axes are not 0-based.)

($m = n = k = 1300$, BROADWELL I7-5557U, 2 threads, ACCELERATE)

processor to operate at the maximum turbo frequency for much longer, if not indefinitely. However, even in our main computing facilities at the RWTH IT CENTER, we observed notable fluctuations of the frequency below its maximum with negative impacts on our measurement quality and stability.

Throughout this work, we consider processors with and without enabled TURBO BOOST. While the performance of these two cases is not directly comparable, we consider our methodologies for both scenarios. In particular, TURBO BOOST is disabled on our SANDY BRIDGE-EP E5-2670 (unless

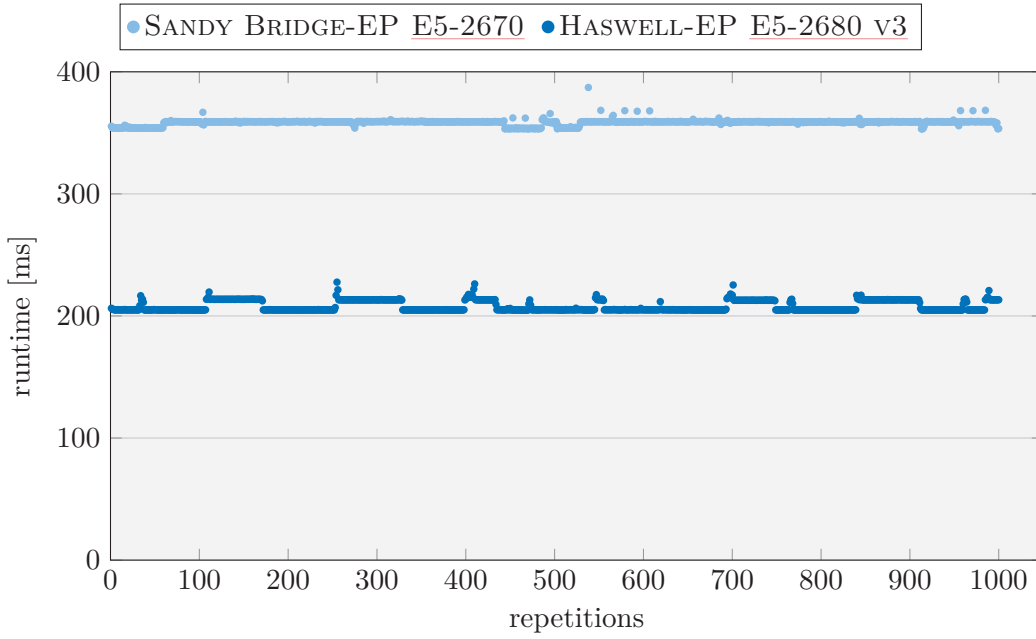


Figure 2.3: Varying runtime for a skewed `dgemmNN` over a period of time.

($m = k = 4000$, $n = 200$, 1 thread, OPENBLAS)

otherwise stated) and enabled on our HASWELL-EP E5-2680 v3—an overview of all hardware configurations is given in [Appendix C](#).

2.1.2.3 Distinct Long-Term Performance Levels

Even with TURBO BOOST disabled, a processor’s speed is not always fixed to its base frequency and we instead observed jumps between two or more *performance levels*.

Example 2.4: Performance levels

Figure 2.3 presents the runtime of 1000 repetitions of the matrix-matrix multiplication $\hat{C} := \hat{A} \hat{B} + \hat{C}$ (`dgemmNN`) with $\hat{A} \in \mathbb{R}^{4000 \times 4000}$ and $\hat{B}, \hat{C} \in \mathbb{R}^{4000 \times 200}$ on a SANDY BRIDGE-EP E5-2670 and a HASWELL-EP E5-2680 v3 (both with TURBO BOOST disabled) with single-threaded OPENBLAS.

On both machines, we can clearly make out two distinct runtime levels:

on the SANDY BRIDGE, the measurements jump between 354 ms and 359 ms, which are 1.4 % apart, and on the HASWELL with twice the floating-point performance per cycle, the two levels at 205 ms and 213 ms differ by 3.9 %. There is no discernible pattern to the jumps between these levels and the processors commonly stay at the same level for 10 s or longer (50 repetitions at 200 ms each).

Since we found no means to eradicate this type of fluctuations, we adopt our measurement setups to account for them: Whenever we have more than one measurement point (e.g., varying the routines or problem sizes), we not only repeat each measurement several times in isolation, but also shuffle the repetitions. As a result, the repetitions for each data point are spread across the entire experiment duration and summary statistics such as the minimum and median yield a stable runtime estimate for only one performance level.

In summary, we can avoid or account for various types of fluctuations within our measurements.

2.1.3 Thread Pinning

Which processor cores a program runs on is generally controlled by the operating system, and in fact most schedulers every now and then move threads between cores at runtime. However, since dense linear algebra kernels immensely rely on temporal data locality within the cache hierarchy and caches shared across multiple cores, moving or physically separating threads may significantly decrease a computation’s efficiency. Counteracting these effects by restricting threads to physical cores is called *thread pinning*.

Example 2.5: Thread pinning

Figure 2.4 presents the compute-bound efficiency (see Appendix A.5) of the matrix-matrix multiplication $C := A \cdot B + C$ (`dgemmTN`) with $A \in \mathbb{R}^{2000 \times 64}$, $B \in \mathbb{R}^{2000 \times 2000}$ and $C \in \mathbb{R}^{64 \times 2000}$ (an example taken from within LAPACK’s blocked `dlauumL`) using OPENBLAS with an increasing number of threads on a two-processor SANDY BRIDGE-EP E5-2670 machine with and without thread pinning.

2.1 Performance Effects for Dense Linear Algebra Kernels

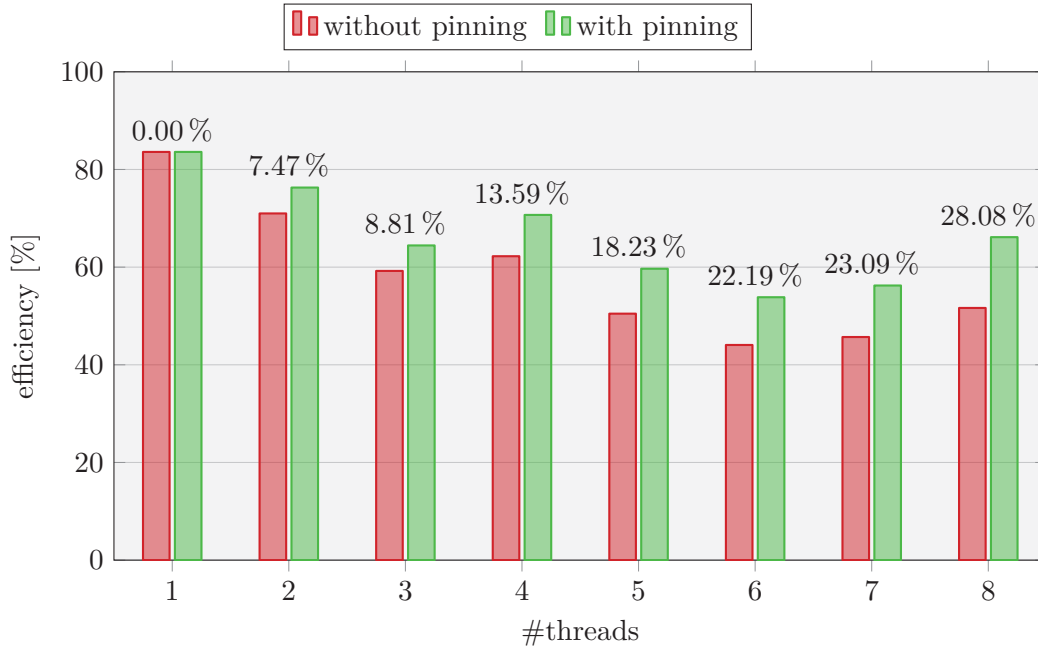


Figure 2.4: Effects of thread pinning on the compute-bound efficiency of a multi-threaded `dgemmTN`. Annotations: speedup of `with pinning` over `without pinning`.

($m = 64$, $n = k = 2000$, SANDY BRIDGE-EP E5-2670, OPENBLAS, median of 100 repetitions)

While the single-threaded `dgemmTN` is not affected by pinning, with two threads, the execution pinned to two cores of one socket (`with pinning`) is 7.47 % faster than the unpinned version (`without pinning`); this difference increases with the number of threads up to 28.08 % on 8 cores.

To ensure that BLAS implementations reach their full potential, throughout this work all measurements are performed with threads pinned to the cores of a single processor.

2.1.4 NUMA Effects

Multi-core and especially multi-processor high-performance machines feature *Non-Uniform Memory Access* (NUMA) architectures: Their physical main-memory is divided into distinct *NUMA domains*, which are connected to individual processors or subset of cores via separate memory channels. While

Work distribution	Data distribution	
	1 NUMA domain	2 NUMA domains
1 processor (8 cores)	6.59 ms	6.91 ms
2 processors (16 cores)	5.79 ms	3.81 ms

Table 2.2: Effects of data distribution in a NUMA machine on the performance of `dgemmNN`.

($m = k = 4000$, $n = 8$, SANDY BRIDGE-EP E5-2670, MKL, median of 1000 repetitions)

some machines have one NUMA domain per processor, others have one domain per memory controller, which are associated to groups of cores.

Each core can access every NUMA domain by routing traffic through the corresponding memory controller. However, compared to accessing a core’s own NUMA domain, accessing remote domains is slower—especially when crossing processor boundaries. Furthermore, a machine’s peak memory bandwidth can only be attained by using all of its memory controllers for all NUMA domains.

Example 2.6: NUMA effects

Table 2.2 presents the runtime of the almost² bandwidth-bound matrix-matrix multiplication $C := A B + C$ (`dgemmNN`) with $A \in \mathbb{R}^{4000 \times 4000}$, and $B, C \in \mathbb{R}^{4000 \times 8}$ using MKL on a two-processor SANDY BRIDGE-EP E5-2670 machine with the data and workload both individually either limited to one processor and its single NUMA domain or spread across both. Two effects can be observed:

- Performing the `dgemmNN` on a single processor while spreading the matrices across both NUMA domains increases the runtime by 4.86 % because accesses to remote data are slightly slower.
- Computing on both processors while all matrices are stored in the first NUMA domain results in a speedup over the single-core computation of only 13.82 % since all data accesses are handled by the first processor’s

² While this `dgemmNN`’s arithmetic intensity is 1.99 FLOPs/byte, on the SANDY BRIDGE-EP E5-2670 computations are theoretically bandwidth-bound only below 1.28 FLOPs/byte (see Appendix A.5).

2.1 Performance Effects for Dense Linear Algebra Kernels

	OPENBLAS	BLIS	MKL	reference
out-of-cache	0.60 ms	1.27 ms	0.68 ms	6.81 ms
in-cache	0.33 ms	1.02 ms	0.41 ms	6.63 ms
overhead	0.27 ms	0.25 ms	0.27 ms	0.18 ms

Table 2.3: Influence of caching on the execution time of `dgemv`.

($m = n = 1000$, SANDY BRIDGE-EP E5-2670, 1 thread, median of 100 repetitions)

memory controllers. However, when the matrices are distributed across both NUMA domains, the speedup increases to 72.96 % because both computation and memory traffic are now shared between the processors.

2.1.5 Caching

The location of operands in a computer’s memory hierarchy—also referred to as the *cache precondition*—can have significant influence on a routine’s performance; an operation whose operands already reside in the processor’s cache (called an *in-cache* scenario or operating on “warm” data) is faster than the same operation that has to load its operands from the slow main memory (*out-of-cache*, “cold” data). This effect is strongest for bandwidth-bound operations that cannot amortize memory stalls with computations.

Example 2.7: Caching

Table 2.3 presents the runtime of the matrix-vector multiplication $y := A x + y$ (`dgemvN`) with $A \in \mathbb{R}^{1000 \times 1000}$ either in- or out-of-cache³ and the same $x, y \in \mathbb{R}^{1000}$ on one core of a SANDY BRIDGE-EP E5-2670 with different BLAS implementations.

Even though the implementations differ by more than 10× in runtime, the overhead of loading A from main memory is comparable between 0.18 ms and 0.27 ms; for OPENBLAS, this corresponds to a runtime increase of over 80 %. Furthermore, the overhead is identical for the two fastest implementations MKL and OPENBLAS, a little lower for the less optimal BLIS, and lowest for the totally unoptimized reference implementation.

The cache precondition of an operation, i.e., which of its operands are where in the memory hierarchy, largely depends on the operation’s context within an algorithm or application. [Chapters 5 and 6](#) address caching in more detail.

2.1.6 Summary

This section studied various effects on the performance of dense linear algebra computations. While some can be avoided altogether, others can be accounted for by specific measurement setups. In the remainder of this work, all measurements are accordingly configured to yield stable results.

2.2 Measurements and Experiments: ELAPS

This section introduces *Experimental Linear Algebra Performance Studies* (ELAPS), the performance measurement framework that serves as the basis for all experiments, modeling procedures, and benchmarks throughout this work. ELAPS was initially developed specifically for our modeling and benchmarking applications, but has since evolved into a versatile general purpose tool-set for various dense linear algebra performance experiments. It is available as an open-source project on GITHUB [102].

ELAPS consists of two layers: The bottom layer offers the SAMPLER, a low-level tool for runtime and performance counter measurements ([Section 2.2.1](#)); the top layer is a PYTHON framework that, among other features, offers user-friendly access to performance experiments and a graphical user interface ([Section 2.2.2](#)).

Publication

The work presented in section is in parts based on research published in:

- [3] Elmar Peise and Paolo Bientinesi. *The ELAPS Framework: Experimental Linear Algebra Performance Studies*. Technical report. Acceptor for publication

³ To place A out of cache, each repetition uses a different memory location for it.

in The International Journal of High Performance Computing Applications. AICES, RWTH Aachen University, Nov. 2016. arXiv: [1504.08035 \[cs.PF\]](#).

2.2.1 The *SAMPLER*

The *SAMPLER* is a command-line performance measurement tool written in C/C++; it essentially times arbitrary executions of dense linear algebra routines. Each *SAMPLER* instance typically provides access to all BLAS and LAPACK routines from one—potentially machine-specific—implementation (e.g., OPENBLAS, BLIS, or MKL), but it is easily extended to other routines with similar interfaces at compile time.

At runtime, the input to the *SAMPLER* determines which routine invocations are executed and timed. The interface provides the following work-flow:

1. Read from standard input a list of *calls*, i.e., routine names with corresponding lists of arguments.
2. Execute the specified calls, and measuring their runtime in terms of processor cycles; optionally track further performance counters through the PERFORMANCE APPLICATION PROGRAMMING INTERFACE (PAPI) [[28](#), [119](#)].
3. Print the measured performance numbers to standard output.

The *SAMPLER* provides configuration options and commands that enable a wide range of performance studies:

- Routine operands can be individually allocated, subdivided, and initialized; this allows to create specific preconditions for calls, such as symmetric positive definite matrices and the placement of operands in the cache hierarchy.
- Any routine that follow the interface conventions of BLAS and LAPACK (see [Appendix B](#)) can be sampled.

2 Performance Effects and Measurements

- Parallel regions allow to execute several routines in parallel through OPENMP. Within such regions, sequential blocks allow run parallel sequences of calls instead.
- Hardware counters (e.g., for cache misses or stalls) can be analyzed through PAPI.

We conclude this section with an example of simple performance experiments in the SAMPLER. A more detailed presentation of the sampler is given in [3], and a complete specification of its interface can be found in its documentation [102].

Example 2.8: The SAMPLER

We interactively start a SAMPLER linked with OPENBLAS on a HASWELL-EP E5-2680 v3. To measure the runtime of the matrix-matrix multiplication $C := A B + C$ (`dgemmNN`) with $A, B, C \in \mathbb{R}^{1000 \times 1000}$, we first allocate three double-precision operands of size $1000 \times 1000 = 1\,000\,000$ doubles as follows:

```
dmalloc A 1000000
dmalloc B 1000000
dmalloc C 1000000
```

To also study the number of Level 3 cache misses, we enable the PAPI counter `PAPI_L3_TCM`:

```
set_counters PAPI_L3_TCM
```

Next, we pass five repeated `dgemmNN`-calls to the SAMPLER and start the measurements with the command `go`:

```
dgemm N N 1000 1000 1000 1 A 1000 B 1000 1 C 1000
dgemm N N 1000 1000 1000 1 A 1000 B 1000 1 C 1000
dgemm N N 1000 1000 1000 1 A 1000 B 1000 1 C 1000
dgemm N N 1000 1000 1000 1 A 1000 B 1000 1 C 1000
dgemm N N 1000 1000 1000 1 A 1000 B 1000 1 C 1000
go
```

After roughly 340 ms, we receive the following output:

2.2 Measurements and Experiments: ELAPS

```
146867632 47155
143853672 10981
143771180 7144
143439224 6764
143589228 6542
```

Here, each line corresponds to one of the five `dgemmNN` invocations, while the first and second entry, respectively, report the number of cycles and Level 3 cache misses. The first `dgemmNN` causes considerable more cache misses than the following and has a slightly higher runtime.

Next, we measure $y := 1.5x + y$ (`daxpy`) with $x, y \in \mathbb{R}^{100\,000}$ using ad-hoc memory locations for the vectors:

```
daxpy 100000 1.5 [100000] 1 [100000] 1
daxpy 100000 1.5 [100000] 1 [100000] 1
daxpy 100000 1.5 [100000] 1 [100000] 1
daxpy 100000 1.5 [100000] 1 [100000] 1
daxpy 100000 1.5 [100000] 1 [100000] 1
```

We end the input stream (ctrl+D) and the SAMPLER produces the following output before terminating:

```
209740 760
157047 0
156753 0
157022 0
157088 0
```

Of the five `daxpys` only the first caused 760 cache misses because it needs to load the kernel itself (the operands were randomized prior to the measurements and thus are still in cache); as a result, the first execution of the inherently bandwidth-bound BLAS Level 1 kernel took about 27% longer than the following.

While the SAMPLER can be used interactively, its interface mainly intended for scripting, which allows its use in various components throughout this work. For interactive use, the ELAPS PYTHON Framework offers a user-friendly interface and tools.

2.2.2 The ELAPS PYTHON Framework

The *ELAPS PYTHON Framework* provides a comprehensive set of tools to facilitate easy and fast, yet powerful performance experimentation in dense linear algebra. It covers various aspects of performance studies:

- Users can easily design *experiments* either through PYTHON scripts or a specialized graphical user interface (GUI): the *PLAYMAT*. Such experiments allow to investigate how performance and efficiency vary depending on factors such as caching, algorithmic parameters, problem size, and parallelism. The experiment design is assisted by features such as built-in knowledge of BLAS and LAPACK signatures and the automatic propagation of problem sizes to various operands within and across routine calls.
- With a simple click (or a method call), an experiment’s measurements are *executed* using a compiled *SAMPLER*. Here, a wide range of execution setups are possible, ranging from *local* executions on laptops, workstations, or interactive nodes to *remote executions* on accelerators or clusters and super-computers through batch-job schedulers.
- The measurements result in experiment *reports* that can be evaluated through further tools and a separate GUI: the *VIEWER*. These cover the core aspects of performances analyses, such as applying different metrics (e.g., runtime [ms], performance [GFLOPs/s], efficiency [%]), combining measurement repetitions into summary statistics (e.g., minimum, median, mean), generating publication-quality *plots*, and *exporting raw data*.

Since we are concerned with performance modeling and prediction, covering ELAPS’s whole spectrum of features for performance experimentation would exceed this work’s focus and scope—interested readers are referred to [3] and encouraged to clone the project from GITHUB [102]. At this point, we limit the presentation of ELAPS to two examples: one that demonstrates the installation process, and another that shows a typical workflow of designing and evaluating a performance experiment through the GUIs.

Example 2.9: ELAPS installation

In this example, we work on a dedicated SANDY BRIDGE-EP [E5-2670](#) remotely through `ssh`; `OPENBLAS`, `PYTHON 2.7`, `PYQT4`, and `MATPLOTLIB` are already available. We begin by cloning ELAPS:

```
$ git clone https://github.com/elmar-peise/ELAPS.git
[...]
$ cd ELAPS
```

Next, we create a `SAMPLER` configuration `Sampler/cfg/OpenBLAS.cfg` (from the provided template) to compile a `SAMPLER` with `OPENBLAS`:

```
$ cd Sampler
$ cat cfg/OpenBLAS.cfg
. ./gathercfg.sh
DFLOPS_PER_CYCLE=8
LINK_FLAGS="-L/path/to/openblas/lib/ \
-lopenblas -lgfortran"
BACKEND_PREFIX="OPENBLAS_NUM_THREADS={nt}"
$ ./make.sh cfgs/OpenBLAS.cfg
[...]
$ cd ..
```

As part of the configuration file, `gathercfg.sh` automatically detects various hardware properties, such as the processor model and frequency, and number of available sockets, cores, and (hyper-)threads.

Now ELAPS is ready for experimentation.

Example 2.10: ELAPS workflow

To evaluate the `OPENBLAS` library on our SANDY BRIDGE-EP [E5-2670](#), we measure the performance of the representative BLAS Level 1, 2, and 3 kernels `ddot`, `dgemv`, and `dgemm`. We start the `PLAYMAT` (`bin/PlayMat`) and through a few clicks construct the experiment shown in [Figure 2.5](#). It consists of the three operations $\alpha := x^T y$ (`ddot`), $y := A x + y$ (`dgemvN`), and $C := A B + C$ (`dgemmNN`) with $A, B, C \in \mathbb{R}^{n \times n}$ and $x, y \in \mathbb{R}^n$, and increasing problem size $n = 10, 20, \dots, 1500$; for each problem size the three operations are repeated 10 times.

2 Performance Effects and Measurements

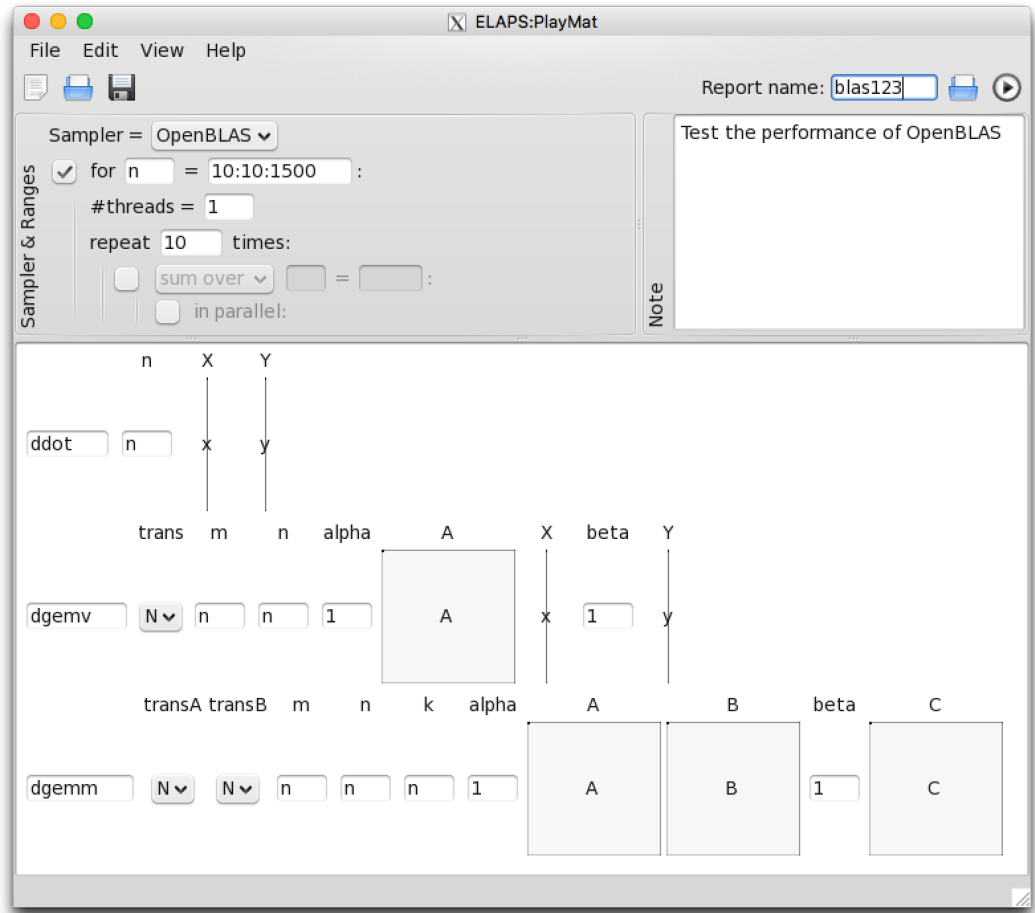


Figure 2.5: Setting up an ELAPS experiment in the PLAYMAT via X11.

A further click starts the experiment execution on the SAMPLER compiled in [Example 2.9](#). We open the resulting report in the VIEWER and quickly obtain a plot of the three routines' median performance as seen in [Figure 2.6](#).

The results show that the performance of the compute-bound `dgemmNN` quickly increases with the problem size and plateaus around 19.3 GFLOPs/s; considering the SANDY BRIDGE's single-threaded peak floating-point performance of 20.8 GFLOPs/s (TURBO BOOST disabled), this corresponds to an efficiency of 92.79 %. The performance of the bandwidth-bound `dgemvN` and `ddot` on the other hand is considerably lower and only reaches, respectively,

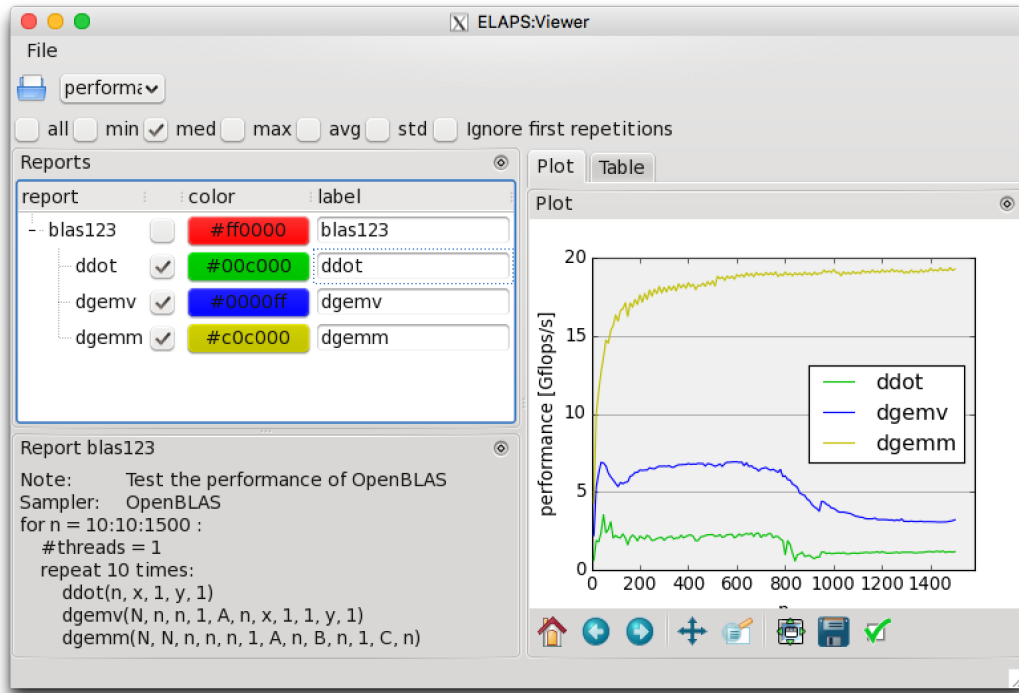


Figure 2.6: The ELAPS VIEWER showing a performance plot.

6.7 GFLOPs/s and 2.3 GFLOPs/s. However, from problem size $n = 800$ to 1000, the performance of these kernels drops by roughly a factor of 2, because their operands ($3n^2 + 2n$ doubles) are larger than the last-level cache (L3) of 20 MiB beyond $n = 935$.

2.3 Summary

This chapter covered the basic phenomena and tools encountered throughout this work: It gave an overview of important effects on the performance of dense linear algebra kernels, including overheads, fluctuations, thread pinning, NUMA effects, and caching. It then introduced the runtime and performance measurement and analysis framework ELAPS, which serves as the basis for all experiments, modeling procedures, and benchmarks throughout this work.

3 Performance Modeling

Many dense linear algebra operations, such as matrix decompositions, reductions, and inversions are commonly implemented as blocked algorithms. Since such algorithms generally cast their entire computation as a sequence of calls to BLAS Level 3 and unblocked LAPACK kernels, we predict their runtime by estimating and summing the runtime of these calls. To motivate how we obtain such estimates for the underlying kernels, recall (from [Section 1.1.1](#)) that every blocked algorithm traverses the input matrix (or matrices) with a fixed block size, and in each traversal step it performs the same kernel operations on the exposed sub-matrices. The sizes of these sub-matrices depend on three factors: the input problem size, the block size, and the traversal progress. Therefore, in order to predict blocked algorithms, we seek a procedure to estimate the runtime of a few compute kernels with potentially widely varying operand sizes.

Our solution to obtain such estimates is measurement-based performance models: For each hardware and software setup and each compute kernel, we construct a separate performance model that represents the kernel’s runtime as a function of its arguments. To efficiently obtain highly accurate models, we tailor them specifically to dense linear algebra computations.

The remainder of this chapter is concerned with the design and automated generation of such models:

- To guide the development of our models, [Section 3.1](#) studies how the runtime of dense linear algebra kernels depends on their arguments. The study reveals the effects of different argument types: While some have little to no effect and can thus be safely ignored in our models to

3 Performance Modeling

reduce their complexity (i.e., their dimensionality), others require careful treatment.

- Based on these insights, [Section 3.2](#) introduces the structure of our performance models and their automated adaptive-refinement-based generation.
- [Section 3.3](#) presents the configuration options of the modeling process and analyzes the resulting models. It studies the trade-off between low model generation cost versus high accuracy, and determines a suitable configuration to generate all models for our predictions.

Following the design and generation of our models, [Chapter 4](#) employs them to predict the performance of blocked algorithms and evaluate the predictions' accuracy and practical value.

Publication

The work presented in this chapter is in parts based on research previously published in:

- [8] Elmar Peise and Paolo Bientinesi. *Cache-aware Performance Modeling and Prediction for Dense Linear Algebra*. Technical report. AICES, RWTH Aachen University, Nov. 2014. arXiv: [1409.8602 \[cs.PF\]](#).
- [11] Elmar Peise and Paolo Bientinesi. "Performance Modeling for Dense Linear Algebra". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. SCC '12. IEEE Computer Society, Nov. 2012, pages 406–416. DOI: [10.1109/SC.Companion.2012.60](#).
- [12] Elmar Peise. "Hierarchical Performance Modeling for Ranking Dense Linear Algebra Algorithms". Master's thesis. Aachen Institute for Computational Engineering Science, RWTH Aachen, May 2012. arXiv: [1207.5217 \[cs.PF\]](#).

3.1 Kernel Argument Analysis

Although maximizing our models' accuracy is our primary focus, we aim to avoid unnecessary complexity and generation cost. For this purpose, we base our model design on domain-specific knowledge regarding the performance

influence of various kernel arguments, which is built up and illustrated in this section.

While dense linear algebra kernels typically have between 5 and 15 arguments, these arguments' semantics divide them among a small set of *argument types*. These argument types play distinct roles in the kernel operation, and have significantly different effects on the attained performance. In the following we study each argument type, and then use the obtained knowledge to design performance models to best represent the observed features.

We consider the following argument types, which cover all BLAS and most LAPACK routines:

- *Flag* arguments identify the form of the operation, such as the order of operands and transpositions (Section 3.1.1).
- *Size* arguments specify the operand sizes (Section 3.1.5).
- *Scalar* arguments contain real or complex scalars that typically multiply (parts of) an operation (Section 3.1.2).
- *Data* arguments are (pointers to) vector and matrix operands (Section 3.1.6).
- *Leading dimension* arguments accompany matrix arguments and specify the distance in memory between two consecutive entries in each matrix row (Section 3.1.3); they allow algorithms to operate not only on contiguously stored matrices but also on sub-matrices.
- *Increment* arguments similarly accompany vectors and specify the distance between consecutive entries (Section 3.1.4); they allow to operate not only on contiguous (column) vectors but, e.g., on rows of matrices.

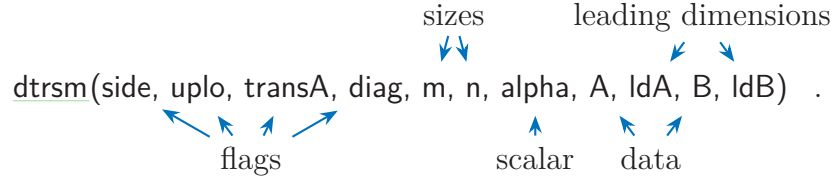
Example 3.1: Argument types

Let us consider `dtrsm`, the double-precision triangular linear system solver with multiple right-hand-sides (e.g., $\boxed{B} := \boxed{A}^{-1} \boxed{B}$). This representative BLAS Level 3 kernel contains most of the above argument types, and is a key component for many LAPACK-level algorithms; hence it is an ideal

3 Performance Modeling

candidate to illustrate both the semantics of the argument types in this example and their performance effects in the following sections.

`dtrsm` is invoked with 11 arguments:



The semantics of these arguments are as follows:

- `side`, `uplo`, `transA`, and `diag` are flag arguments.
 - `side` $\in \{L, R\}$ determines from which side B is multiplied with A^{-1} , i.e., the *left* ($B := A^{-1}B$) or *right* ($B := B A^{-1}$),
 - `uplo` $\in \{L, U\}$ indicates a *lower*- or *upper*-triangular system matrix (A or A^T),
 - `transA` $\in \{N, T\}$ specifies whether A appears *non-transposed* or *transposed*, and
 - `diag` $\in \{N, U\}$ determines whether the diagonal entries of A are stored *normally* or all implicitly equal to 1, making A „unit triangular”.

All $2^4 = 16$ combinations of these four flag arguments are possible. For instance, $(\text{side}, \text{uplo}, \text{transA}, \text{diag}) = (L, U, N, N)$ identifies the operation $B := A^{-1}B$, and (R, L, T, N) yields $B := B A^{-T}$.

- `m` and `n` are size arguments; they determine the size of $B \in \mathbb{R}^{m \times n}$ and accordingly $A \in \mathbb{R}^{m \times m}$ if `side` = L and $A \in \mathbb{R}^{n \times n}$ if `side` = R.
- `alpha` is a scalar argument; it multiplies the whole linear system, i.e., $B := \alpha A^{-1}B$.
- `A` and `B` are data arguments; they represent the operands A and B (as pointers to their first entries).

- `ldA` and `ldB` are leading dimension arguments for, respectively, `A` and `B`.

A brief overview of not only `dtrsm` but all BLAS and LAPACK routines used throughout this work and their arguments is given in [Appendix B](#).

In the following, we consider the influence of each argument type on the performance of kernels, and determine how they shall be handled in our models.

3.1.1 Flag Arguments

Flag arguments accept only a few discrete values—in most cases two. However, since they specify which form of the operation is performed, they may trigger entirely different execution branches in kernel implementations, and thus result in independent runtimes.

Example 3.2: Flag arguments

Figure 3.1 shows the runtime of

`dtrsm`(^{side}`side`, ^{uplo}`uplo`, ^{transA}`transA`, ^{diag}`diag`, ^m256, ⁿ256, ^{alpha}1.0, ^A`A`, ^{ldA}256, ^B`B`, ^{ldB}256) ,

i.e., an operation like $\boxed{B} := \boxed{A}^{-1} \boxed{B}$ with $\boxed{A}, \boxed{B} \in \mathbb{R}^{256 \times 256}$, for all 16 combinations of the flag arguments `side`, `uplo`, `transA`, and `diag` on a SANDY BRIDGE-EP [E5-2670](#) and a HASWELL-EP [E5-2680 v3](#) with single-threaded OPENBLAS, BLIS, and MKL.

Across all machines and libraries, we encounter a large spectrum of performance dependencies, which cannot be summarized in a single pattern. In particular, each argument influences the runtime of the implementations differently:

- For non-square $\boxed{B} \in \mathbb{R}^{m \times n}$, `side` affects the `dtrsm`'s minimal FLOP-count: While for `side` = `L` its cost is $m^2 n$ FLOPs, for `side` = `R` it is mn^2 FLOPs. Hence changing the value of `side` will generally lead to an entirely different runtime.

Since this example uses $m = n = 256$, the `dtrsm` requires at least

3 Performance Modeling

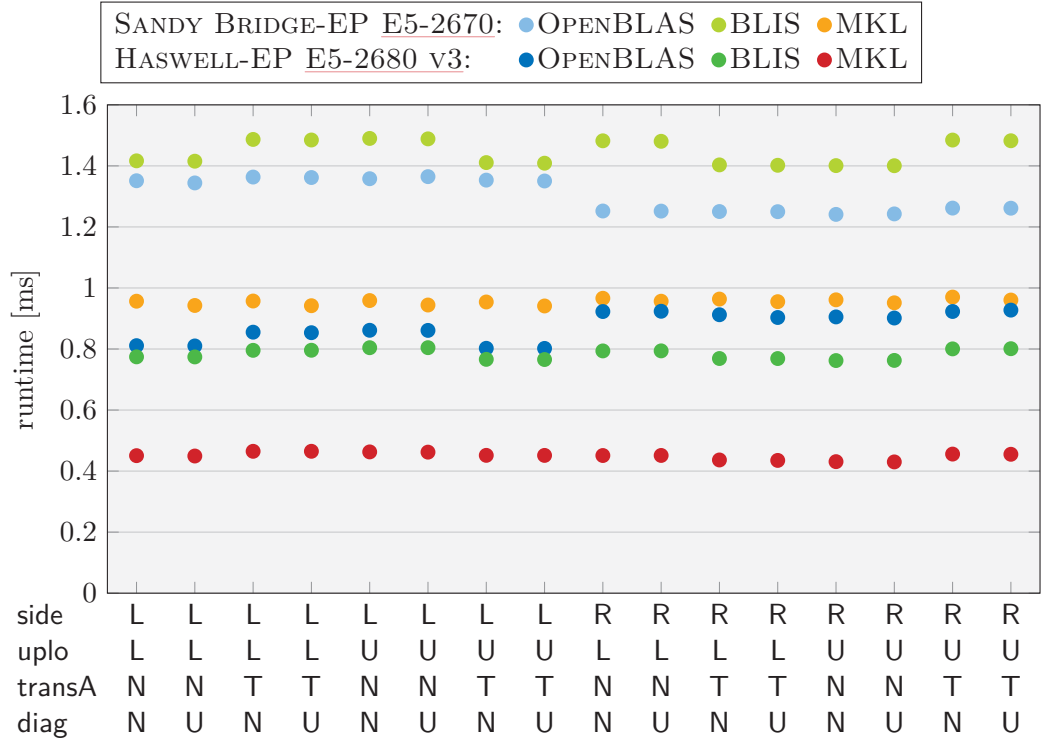


Figure 3.1: Runtime of `dtrsm` as a function of its flag arguments.

($m = n = 256$, 1 thread, median of 100 repetitions)

256³ FLOPs for both values of `side`. However, in our measurements, `side` still has the largest impact on performance, which is most evident for OPENBLAS: While on the SANDY BRIDGE (●) the `dtrsm` takes on average 104.52 μ s (8.35 %) longer for `side = L` than with `side = R`, on the HASWELL-EP E5-2680 v3 (●) `side = L` is 82.845 μ s (9.06 %) slower than `side = R`.

- The effects of `uplo` and `transA` are closely related, which is most evident in BLIS (●, ●). Possibly due to the similarity of the operations, $(\text{uplo}, \text{transA}) = (\text{L}, \text{N})$ and (U, T) commonly share a runtime that is different from (L, T) and (U, N) .
- `diag` has almost no influence on the runtime of most implementations.

Only MKL (●, ●)—the fastest implementation across all setups—takes advantage of `diag = U`, and avoids the division instructions.

Note that both the magnitude of the flag arguments’ influence as well as the type of the resulting runtime characteristics vary both from one architecture to another and between implementations.

Since flag arguments can have a decisive impact on a kernel’s runtime with no general discernible patterns across architectures and implementations, we will generate a separate performance (sub-)model for each different combination of flags. However, note that in our target range of algorithms, we encounter only a limited set of such combinations, and will therefore not generate models for all possibilities.

3.1.2 Scalar Arguments

At first sight, scalar arguments should not have any effect on a kernel’s runtime—after all, they only scale a kernel operand independent of the argument’s value. However, at closer inspection, we find that for certain values—namely -1 , 0 , and 1 —this multiplication can be avoided altogether. Since in applications and algorithms, scalar arguments to kernels are almost exclusively -1 , 0 , and 1 , most kernel implementations feature optimized execution branches for these values. Just as for flag arguments, such branches can noticeably impact a kernel’s runtime and performance.

Example 3.3: Scalar arguments

Figure 3.2 shows the runtime of

```
side uplo transA diag m n alpha A ldA B ldB
dtrsm( L , L , N , N , 100, 800, alpha, A, 100, B, 100) ,
```

i.e., $\underline{B} := \alpha A^{-1} \underline{B}$ with $A \in \mathbb{R}^{100 \times 100}$ and $\underline{B} \in \mathbb{R}^{100 \times 800}$, for $\alpha \in \{0.6, 0, -1, 1\}$ on a SANDY BRIDGE-EP E5-2670 and a HASWELL-EP E5-2680 v3 with single-threaded OPENBLAS, BLIS, and MKL. While $\alpha = 0.6$ represents the “general case”, $\alpha = 0$, -1 , and 1 are special values

3 Performance Modeling

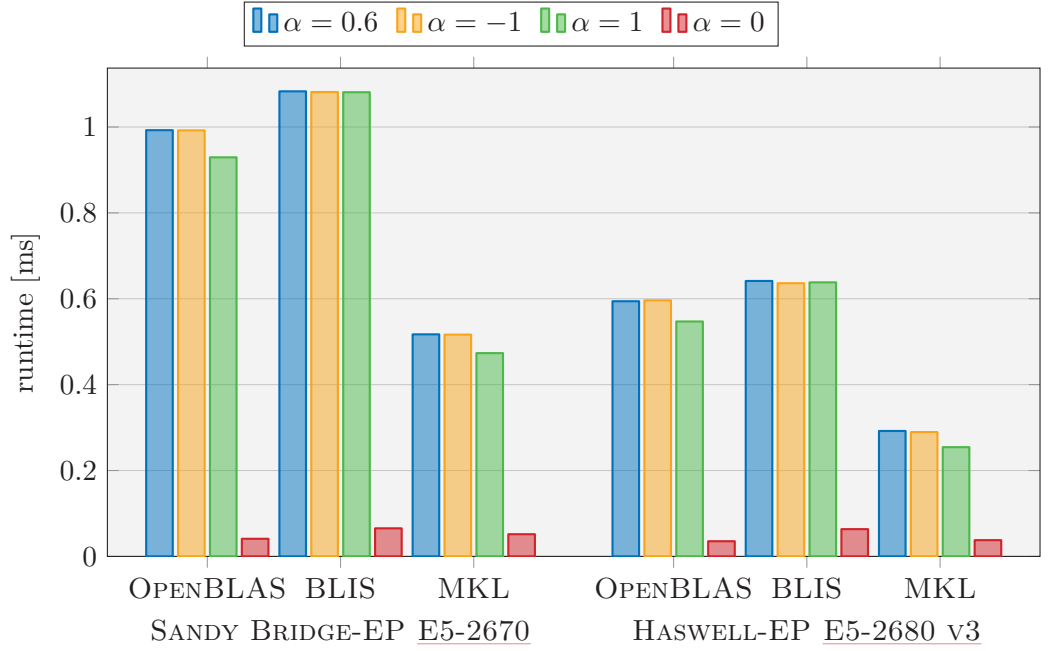


Figure 3.2: Runtime of `dtrsmLLNN` with different values for α .

($m = 100$, $n = 800$, 1 thread, median of 100 repetitions)

for which implementations can avoid multiplications—in algorithms and applications $\alpha = 1$ and -1 are the most common values.

All implementations take advantage of $\alpha = 0$ (red). In this case, the `dtrsmLLNN` only sets $\mathbf{B} := \mathbf{0}$ and no computations are performed. Furthermore, all implementations treat $\alpha = -1$ (orange) just like the general case (blue) resulting in the same runtime.

$\alpha = 1$ (green) is handled differently by the three implementations: While BLIS attains the same performance as for $\alpha = 0.6$ and -1 , OPENBLAS and MKL are on average 9.66 % faster compared to these cases, indicating optimizations that avoid multiplications with 1. While we can appreciate that OPENBLAS and MKL are faster for $\alpha = 1$, put into perspective the increase in runtime for other values of α is surprisingly high: In our example, scaling \mathbf{B} accounts for only 1 % of the `dtrsmLLNN`’s minimal FLOP-count, yet makes the operation almost 10 % slower.

To represent the influence of scalar arguments on kernel performance in our

models, we will treat them like flag arguments with the four possible values -1 , 0 , 1 , and “any other value”. Since blocked algorithms almost exclusively use the values -1 and 1 , we will not observe a four-fold increase in the complexity of our models.

3.1.3 Leading Dimension Arguments

Leading dimension arguments determine the memory access strides of kernels that load multiple columns of a matrix simultaneously. They only have a small influence on kernel performance, but we need to be aware of certain patterns to avoid undesirable effects when generating our performance models.

3.1.3.1 Alignment to Cache-Lines

Data is moved through the memory hierarchy in blocks of 64 bytes (= 8 doubles) called *cache-lines*.¹ Hence using multiples of the cache-lines size as memory access strides typically shows a more regular and often better performance compared to other strides.

Example 3.4: Aligning leading dimensions to cache-lines

Figure 3.3 shows the runtime of

$\text{dtrsm}(\overset{\text{side}}{L}, \overset{\text{uplo}}{L}, \overset{\text{transA}}{N}, \overset{\text{diag}}{N}, \overset{m}{256}, \overset{n}{256}, \overset{\text{alpha}}{1.0}, \overset{A}{A}, \overset{\text{ldA}}{ld}, \overset{B}{B}, \overset{\text{ldB}}{ld})$,

i.e., $\begin{bmatrix} B \end{bmatrix} := \begin{bmatrix} A^{-1} \end{bmatrix} \begin{bmatrix} B \end{bmatrix}$ with $\begin{bmatrix} A \end{bmatrix}, \begin{bmatrix} B \end{bmatrix} \in \mathbb{R}^{256 \times 256}$, for leading dimensions² $ld = 256, \dots, 320$ in steps of 1 on a SANDY BRIDGE-EP E5-2670 and a HASWELL-EP E5-2680 v3 with single-threaded OPENBLAS, BLIS, and MKL.

For all setups, the `dtrsmLLNN`’s runtime exhibits some regular pattern in terms of the leading dimension arguments—with an average amplitude of 2.19 %. However the patterns are quite different: While OPENBLAS’s runtime on the SANDY BRIDGE (—) drops equally at every even leading

¹ The cache-line size is generally not fixed but for most processors it is 64 byte.

² Since A and B have 256 rows, the leading dimensions are at least 256.

3 Performance Modeling

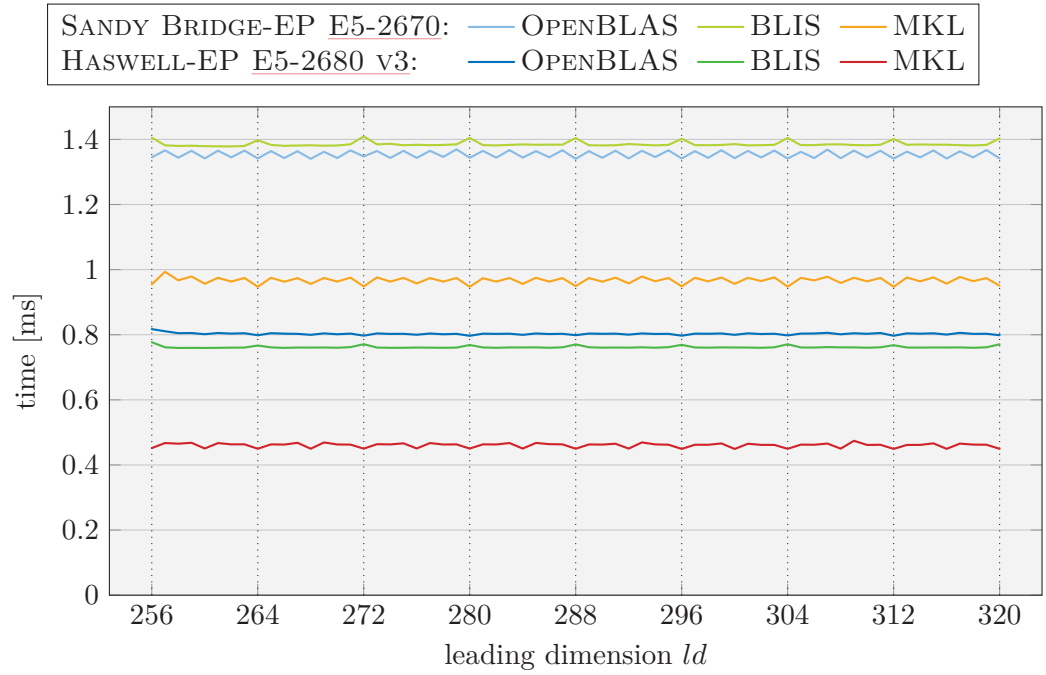


Figure 3.3: Runtime of `dtrsm` as a function of its leading dimension arguments on a small scale. Dotted lines: multiples of 8.

($m = n = 256$, 1 thread, median of 100 repetitions)

dimension, MKL on the HASWELL (—) dips only at multiples of 4, and on the SANDY BRIDGE (—) it has stronger dips at multiples of 8. BLIS on the other hand shows the exact opposite behavior: On both platforms (—, —) its runtime spikes slightly at multiples of 8.

Independent of the specific behavior of each setup, a smooth runtime curve is obtained when only multiples of 8 are considered as leading dimensions.

To avoid small performance irregularities, we will generate our models using *multiples of the cache-line size* for leading dimensions—in double-precision: multiples of 8.

3.1.3.2 Set-Associative Cache Conflicts

The Level 1 and 2 caches in our processors are *8-way set-associative*: They are divided into sets of 8 cache-lines, and when a cache-line is loaded, its address's

least significant bits determine which of the sets it is assigned to; within the set, an architecture-dependent cache replacement policy determines in which of the 8 slots it is stored. When the address space is accessed contiguously, consecutive cache-lines are loaded into consecutive sets, and the cache is filled evenly. In the worst case, however, the address space is accessed with a stride equal to the number of sets, and all loaded cache-lines are associated to the same set: Only 8 cache-lines are cached, and each additional line results in a *cache conflict miss* causing a recently loaded line to be evicted. This effect should be avoided whenever possible.

On recent INTEL XEON processors, the Level 1 data cache (L1d) fits 32 KiB organized as 64 sets of 8 cache-lines. A memory location with address a is a part of cache-line $\lfloor a/64 \rfloor$ (due to the size of 64 byte per line) and assigned to set $\lfloor a/64 \rfloor \bmod 64$ (due to the capacity of 64 sets). The Level 2 cache (L2) in turn fits 256 KiB in 1024 sets; here address a is assigned to set $\lfloor a/64 \rfloor \bmod 1024$.

In a double-precision matrix stored with leading dimension ld , consecutive elements in each row are $8ld$ bytes apart (1 double = 8 bytes). Hence, for $ld = 512$, the consecutive row elements starting at address a_0 are stored at $a_i = a_0 + 8ld \cdot i = a_0 + 4096i$, and associated to the same set in the L1d cache:

$$\begin{aligned} \left\lfloor \frac{a_i}{64} \right\rfloor \bmod 64 &= \left\lfloor \frac{a_0 + 4096i}{64} \right\rfloor \bmod 64 \\ &= \left(\left\lfloor \frac{a_0}{64} \right\rfloor + 64i \right) \bmod 64 \\ &= \left\lfloor \frac{a_0}{64} \right\rfloor \bmod 64. \end{aligned}$$

The same problem occurs for leading dimensions that are multiples of 512, and even below 512 powers of 2 have a similar effect: E.g., with $ld = 256$ the elements of a row are associated to only two of the cache's 64 sets. Similarly, for the L2 cache with 1024 sets, consecutive row-elements are associated to the same cache set for leading dimensions that are multiples of 8192, and multiples of 4096 utilize only two sets.

3 Performance Modeling

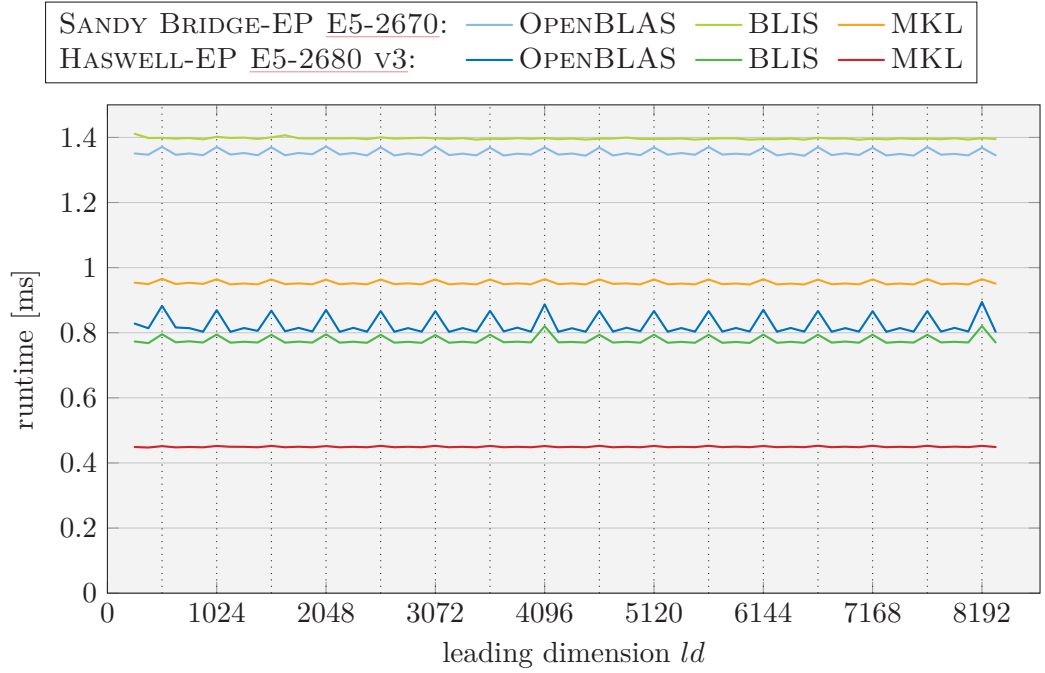


Figure 3.4: Runtime of `dtrsmLLNN` as a function of its leading dimension arguments on a large scale. Dotted lines: multiples of 512.

($m = n = 256$, 1 thread, median of 100 repetitions)

Example 3.5: Cache conflict misses caused by leading dimensions

Figure 3.4 shows the runtime of

$\text{dtrsm}(\text{side}, \text{uplo}, \text{transA}, \text{diag}, m, n, \text{alpha}, A, \text{ldA}, B, \text{ldB})$,

i.e., $\boxed{B} := \boxed{A}^{-1} \boxed{B}$ with $\boxed{A}, \boxed{B} \in \mathbb{R}^{256 \times 256}$, for leading dimensions $ld = 256, \dots, 8320$ in steps of 128 on a SANDY BRIDGE-EP E5-2670 and a HASWELL-EP E5-2680 v3 with single-threaded OPENBLAS, BLIS, and MKL.

For most setups the runtime spikes above the baseline at multiples of 512. However, the average magnitude of these spikes ranges from 0.14% for BLIS on the SANDY BRIDGE (—) to 8.37% for OPENBLAS on the HASWELL (—). Especially for OPENBLAS (—, —), there are

additional, yet lower spikes of 1.40 % at multiples of 256. Furthermore, on the HASWELL for both OPENBLAS (—) and BLIS (—) the spikes are especially high at $ld = 4096$ and 8192 , exceeding the baseline by, respectively, 6.55 % and 11.24 %.

To prevent distortions from unfortunate leading dimensions in our model generation altogether, we will *avoid multiples of 256* for these arguments.

Note that by using leading dimensions that are multiples of 8, yet not of 256 in our measurements, our models will not yield accurate predictions for kernel invocations that do not follow this pattern. However, predicting the performance of such unfortunate invocations, which can be systematically avoided, is not part of our models' purpose and would exceed the scope of this work.

3.1.4 Increment Arguments

With our focus on predicting algorithms that primarily use BLAS Level 3 (matrix-matrix operations) and unblocked LAPACK kernels, the performance of vector operations is not our primary focus. However, to make our performance modeling technique applicable to all types of operations, this section briefly studies the influence of increment arguments on kernel performance.

Increment arguments directly determine the memory access strides of vector operands. In algorithms and applications, they are typically either 1 to access contiguous vectors (e.g., columns of matrices) or the leading dimension of a matrix, i.e., $\gg 1$, to access matrix rows. While in the first case, a vector of length n occupies $\lfloor n/8 \rfloor$ cache-lines,³ in the second case it is spread across n cache-lines. As a result, increments of 1 cause less data movement and are thus favorable in terms of performance.

Beyond the ideal increment of 1, the influences of increment arguments on performance exhibit periodic patterns similar to those for leading dimensions. However, in comparison the resulting effects are commonly far more severe

³ Assuming the first entry is aligned to the beginning of a cache-line.

3 Performance Modeling

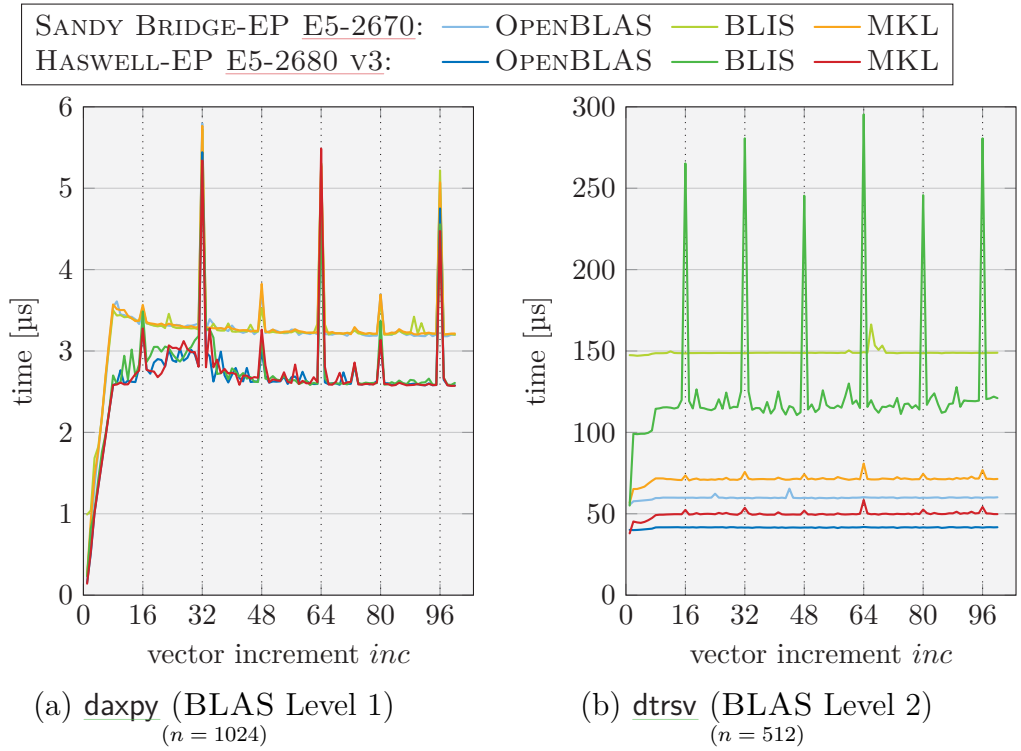


Figure 3.5: Runtime of `daxpy` and `dtrsvLNN` as a function of their increment arguments. Dotted lines: multiples of 16.

(1 thread, median of 100 repetitions)

because cache misses directly increase the runtime for bandwidth-bound (matrix-)vector operations.

Example 3.6: Increment arguments in BLAS Level 1

Figure 3.5a shows the runtime of the BLAS Level 1 calls

```
daxpy(n, alpha, X, incX, Y, incY),
```

i.e., $y := 2x + y$ with $x, y \in \mathbb{R}^{1024}$, for increments $inc = 1, \dots, 100$ in steps of 1 on a SANDY BRIDGE-EP E5-2670 and a HASWELL-EP E5-2680 v3 with single-threaded OPENBLAS, BLIS, and MKL.

The results for all three implementations are similar on both machines: The `daxpy`'s runtime is shortest for $inc = 1$, and increases steadily until

$inc = 8$; the difference in performance between these two cases lies between $3.53\times$ for BLIS on the SANDY BRIDGE (—) (whose BLAS Level 1 is not optimized for our architectures) and $18.29\times$ for MKL on the HASWELL (—).

Beyond $inc = 8$, the runtime spikes above a steady baseline of $3.20\mu s$ on the SANDY BRIDGE (—, —, —) and $2.73\mu s$ on the HASWELL (—, —, —) by up to 95.88% at each multiple of 32 and slightly less by 16.80% for other multiples of 16.

Example 3.7: Increment arguments in BLAS Level 2

Figure 3.5b shows the runtime of the BLAS Level 2 calls

$\text{dtrsv}(\overset{\text{uplo}}{L}, \overset{\text{trans}}{N}, \overset{\text{diag}}{N}, \overset{n}{512}, \overset{A}{A}, \overset{ldA}{1000}, \overset{X}{X}, \overset{incX}{inc})$,

i.e., $x := A^{-1}x$ with $A \in \mathbb{R}^{512 \times 512}$ and $x \in \mathbb{R}^{512}$, for increments $inc = 1, \dots, 100$ in steps of 1 on a SANDY BRIDGE-EP E5-2670 and a HASWELL-EP E5-2680 v3 with single-threaded OPENBLAS, BLIS, and MKL.

We immediately notice that BLIS on the HASWELL (—) has runtime spikes similar to those for `daxpy`, which hints at an implementation of `dtrsv` in terms of BLAS Level 1 kernels. For all other setups, the runtime is considerably smoother with the exception of MKL (—, —), which shows small spikes of 6.03% at multiples of 16.

Since in practice increments are either 1 or equal to the leading dimension of a matrix, we will treat them in our models like flag arguments that take the values 1 and “any large value”, for which we *avoid multiples of 16* to avoid outlier measurements.

3.1.5 Size Arguments

A kernel’s size arguments determine its minimal FLOP-count and thus directly influence on its runtime. In the following, we study this influence first for small changes in the operand sizes (Section 3.1.5.1) and then on a larger scale (Section 3.1.5.2).

3 Performance Modeling

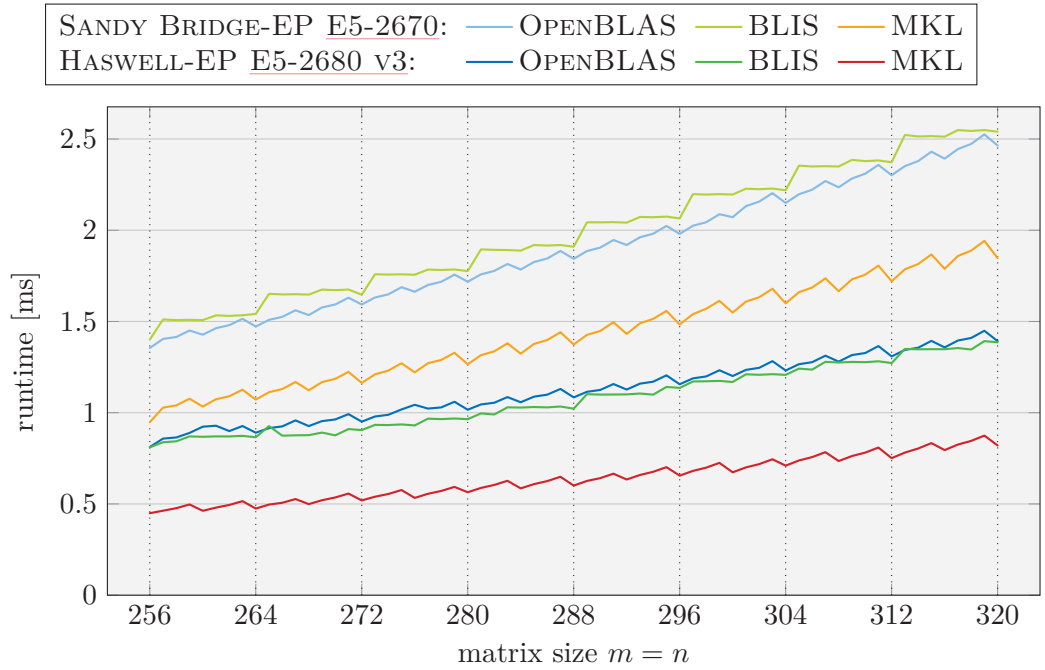


Figure 3.6: Runtime of `dtrsmLLNN` as a function of its size arguments on a small scale. Dotted lines: multiples of 8.

(1 thread, median of 100 repetitions)

3.1.5.1 Smalls Scale Behavior

Optimizations of compute kernels commonly involve vectorization and loop unrolling of length 4 or 8. These optimizations typically have a direct influence on a kernel's runtime for small variations of the size arguments.

Example 3.8: Small variations of size arguments

Figure 3.6 shows the runtime of

```

side uplo transA diag m n alpha A ldA B ldB
dtrsm( L , L , N , N , n, n, 1.0, A, 400, B, 400) ,

```

i.e., $B := A^{-1} B$ with $A, B \in \mathbb{R}^{n \times n}$, for $n = 256, \dots, 320$ in steps of 1 on a SANDY BRIDGE-EP E5-2670 and a HASWELL-EP E5-2680 v3 with single-threaded OPENBLAS, BLIS, and MKL.

All setups show periodic patterns in their runtimes. While these patterns

differ between the implementations, most have local runtime minima at multiples of 4, and all of them have minima at multiples of 8.

To avoid runtime artefacts introduced by vectorization and loop unrolling, we will build our models on measurements that *use multiples of 8* for all size arguments.

3.1.5.2 Piecewise Polynomial Behavior

Since an operation’s minimal FLOP-count is generally a (multivariate) polynomial function of the size arguments, one might expect that (for compute-bound kernels) it translates directly into an equally polynomial runtime. However, since a kernel’s performance is generally not constant for varying operand sizes, a single polynomial is often insufficient to accurately represent a kernel’s runtime for large ranges of problem sizes.

Example 3.9: Polynomial fitting for size arguments

Figure 3.7a shows the runtime of

```
side uplo transA diag m n alpha A ldA B ldB
dtrsm( L , L , N , N , n, n, 1.0, A, 1000, B, 1000) ,
```

i.e., $B := A^{-1}B$ with $A, B \in \mathbb{R}^{n \times n}$, with $n = 24, \dots, 536$ in steps of 16 on a SANDY BRIDGE-EP E5-2670 and a HASWELL-EP E5-2680 v3 with single-threaded OPENBLAS, BLIS, and MKL.

At first sight, the runtime for all setups follows a smooth cubic behavior—perfectly in line with the operation’s minimal cost of n^3 FLOPs. However, if for each setup we fit the measurements with a single cubic polynomial that minimizes the least-squares relative error (details in Section 3.2.4), we are left with the approximation error shown in Figure 3.7b. The absolute relative approximation error⁴ lies between 0.86 % for BLIS on the SANDY BRIDGE (—) and 11.22 % for OPENBLAS on the HASWELL (—); on average it is 5.30 %.

If we look closer at the approximation errors in Figure 3.7b—especially for OPENBLAS on the HASWELL (—)—we observe a piecewise smooth(er)

3 Performance Modeling

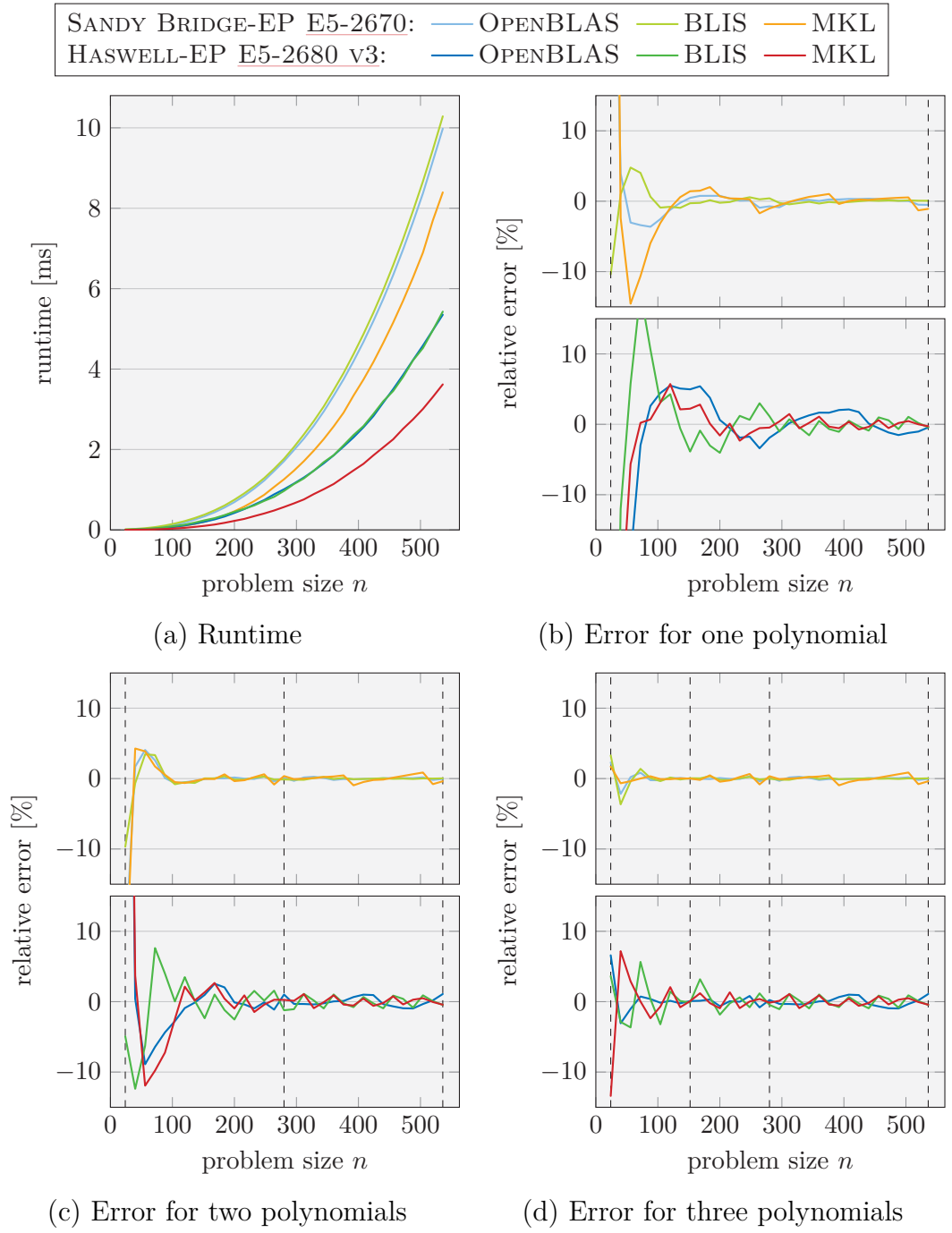


Figure 3.7: Runtime and error of piecewise cubic polynomial fits `dtrsmLLNN`.

Dashed lines: polynomial boundaries.

(1 thread, median of 100 repetitions)

behavior. Motivated by this observation, we now fit not one polynomial to each data-set but two: one for the first half ($n \leq 280$) and one for the second half ($n \geq 280$). For this two-split polynomial fit the approximation error is shown in [Figure 3.7c](#): The largest error is now reduced to 5.25 % for MKL on the HASWELL (—), and the average error is 2.55 %—less than half of the original approximation error. (Based on a more detailed analysis, a better splitting point than $\frac{24+536}{2} = 280$ could have been chosen, but as [Figure 3.7b](#) shows such choices would be notably different for each setup.) Within the new approximation, the error for the second polynomial ($n \geq 280$) is already quite low—on average 0.38 %. Hence, in a second step, we further subdivide only the first half of the domain ($n \leq 280$) at $n = 152$, and generate a new approximation consisting of three polynomials. As [Figure 3.7d](#) shows, the error of this approximation is below 1.28 % (—) in all cases and on average 0.71 %.

To account for the not purely polynomial influence of a kernel’s size arguments on its runtime, we will represent it in our models through *piecewise polynomials*. Details on the such piecewise polynomial representations and their automated generation are given in [Sections 3.2.4](#), [3.2.5](#), and [3.3](#).

3.1.6 Data Arguments

With few exceptions (such as eigensolvers), the executed instructions and thus the runtime of kernels do not depend on their operands’ numerical values. However, the runtime may depend on where these operands are located within the memory hierarchy: Kernels whose operands reside in cache prior to their invocation run faster.

⁴ For a polynomial $p(x)$ fit to measurements y_1, \dots, y_N in points x_1, \dots, x_N we consider the error $1/N \sum_{i=1}^N |y_i - p(x_i)|/y_i$. Note that the least-squares fitting minimizes not this sum of absolute relative errors but the sum of squared relative errors.

3 Performance Modeling

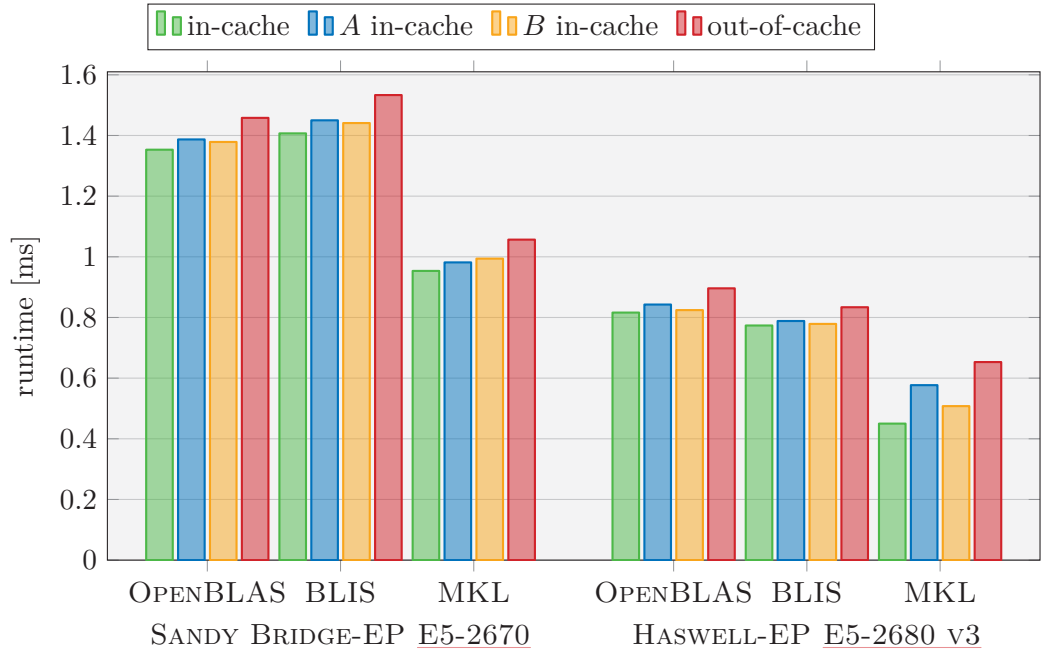


Figure 3.8: Runtime of $\text{dtrsm}_{\text{LLNN}}$ with in-cache and out-of-cache operands.
 ($m = n = 256$, 1 thread, median of 100 repetitions)

Example 3.10: Data arguments

Figure 3.8 shows the runtime of

$\text{dtrsm}(\text{side}, \text{uplo}, \text{transA}, \text{diag}, m, n, \text{alpha}, A, \text{ldA}, B, \text{ldB})$
 $\text{dtrsm}(\text{L}, \text{L}, \text{N}, \text{N}, 256, 256, 1.0, A, 256, B, 256)$,

i.e., $B := A^{-1}B$ with $A, B \in \mathbb{R}^{256 \times 256}$, for A and B a-priori either in- or out-of-cache on a SANDY BRIDGE-EP E5-2670 and a HASWELL-EP E5-2680 v3 with single-threaded OPENBLAS, BLIS, and MKL.

Across all setups, the pure in-cache scenario (green) is consistently faster than out-of-cache (red) by between 7.75% (OPENBLAS on the SANDY BRIDGE) and 45.08% (MKL on the HASWELL). While the scenarios where either only A or only B is in-cache (blue, orange) are always between these extremes, which of the two is faster depends on both the architectures and the BLAS implementation.

The exact effects of caching on kernel runtime and performance are hard to predict. However, since blocked algorithms operate on matrices with high locality, we will generate our models with in-cache operands where possible: By repeating each measurement twice, the most-recently-used portions of a kernel’s operands (the entire operands for small operations) from the first repetition are in-cache prior to the second repetition. Only these second repetitions’ measurements are used to construct our models.

We will revisit caching in more detail in [Chapters 5 and 6](#).

3.1.7 Summary

This section studied the effects of various argument types on kernel runtime. In summary, these effects and our decisions on how to represent them in our models are as follows:

- Flag arguments ([Section 3.1.1](#)) can invoke separate execution branches within kernel implementations. Hence we will generate a separate sub-model for each relevant combination of flag arguments.
- Scalar arguments ([Section 3.1.2](#)) affect the performance of kernels only for the special values that allow to avoid certain arithmetic operations. Hence we will scalars them just like flags with the possible values -1 , 0 , 1 , and “any other value”.
- Size arguments ([Section 3.1.5](#)) greatly influence a kernel’s runtime by determining its minimal FLOP-count. While this FLOP-count is usually polynomial in the operand sizes, a kernel’s runtime can typically not be represented accurately by a single polynomial. Hence, we will model the effect of size arguments on runtime as piecewise polynomials. Furthermore, to avoid small-scale runtime artefacts, we will ensure that in all measurements all size arguments are multiples of 8.
- Data arguments ([Section 3.1.6](#)) do not affect the runtime of targeted kernels through their numeric values. However, the operand’s location

3 Performance Modeling

in the processor’s memory hierarchy prior to a kernel invocation may lead to different performance. While we could account for this effect by generating separate models for specific memory preconditions (such as in- and out-of-cache), we will focus on models based on repeated measurements that correspond to in-cache data for operands smaller than the cache.

- Leading dimension arguments ([Section 3.1.3](#)) generally have only a minor effect on kernel runtime, but they should be chosen as multiples of 8, yet not of 512. To generate our models, we will hence set all leading dimensions to a constant value, such as 5000.
- Increment arguments ([Section 3.1.4](#)) are typically 1 or equal to a matrix’s leading dimension. We will hence treat them as flag arguments with the two values 1 and “any large value”. Since multiples of 16 as leading dimensions can incur runtime spikes, especially in BLAS Level 1 kernels, we will choose a fixed large value for the second case that is not a multiple of 16, such as 5000.

Based on these decisions on how to represent the influence of various argument types on kernel runtime in our measurement-based performance models, the following section describes our models’ structure and their automated generation.

3.2 Model Generation

After analyzing the performance effects of various argument types on dense linear algebra kernels in the previous section, we now turn to the design and generation of our performance models.

[Section 3.2.1](#) introduces the model structure and how their coverage is configured. The following sections detail how each model (and sub-model) is generated based on measurements: [Section 3.2.2](#) describes the selection of measurement points in the kernel’s argument space; [Section 3.2.3](#) discusses

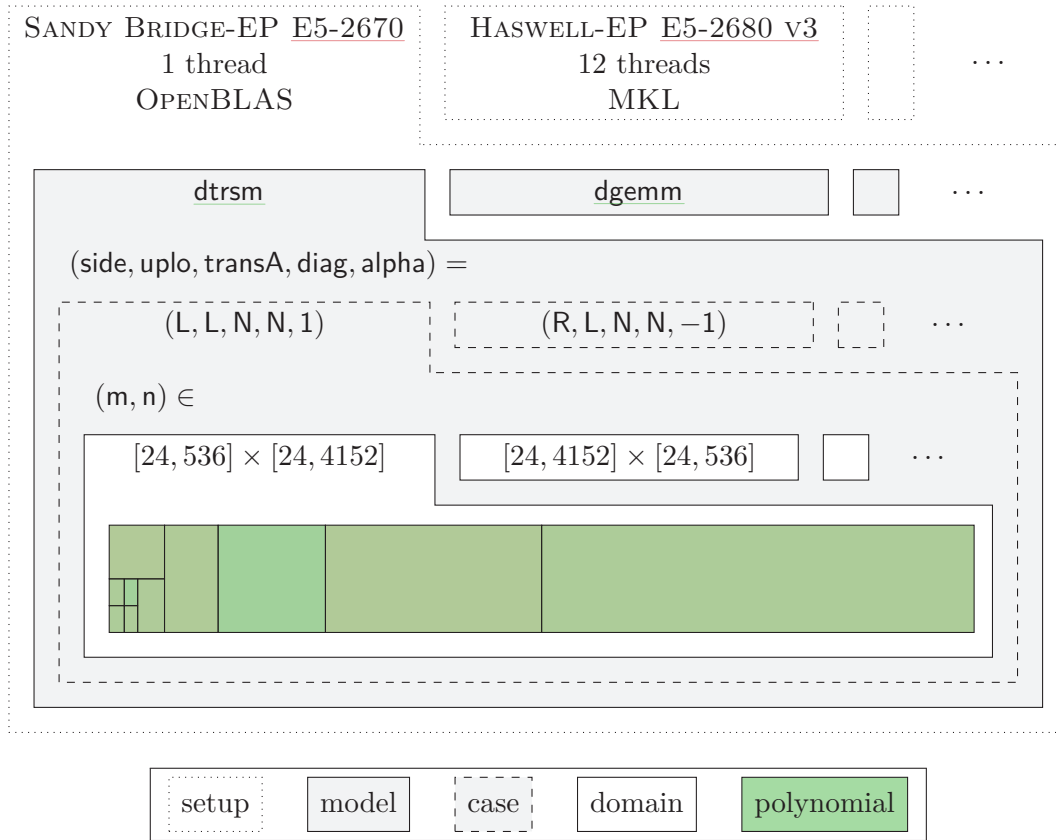



Figure 3.9: Structure of the performance models.

how repeated measurements at these points are used to compute summary statistics of the expected kernel runtime; [Section 3.2.4](#) specifies how set of measurements is least-squares fitted with a single polynomial; and finally [Section 3.2.5](#) introduces the adaptive refinement approach that covers the range of problem sizes with piecewise polynomials.


3.2.1 Model Structure

Based on the analyses of how a kernel's different argument types affect its performance in [Section 3.1](#), we arrive at the structure for our *performance models* depicted in [Figure 3.9](#).



For each setup  consisting of the hardware platform, number of threads,

3 Performance Modeling


and the BLAS implementation, a separate set of models is generated. Independent kernels can be modeled for each setup.

Each model  represents the runtime of one kernel (e.g., `dtrsm` or `dgemm`): It is essentially a function of the kernel’s arguments that returns runtime estimates.⁵ To account for variations in kernel runtime for fixed arguments, each estimate is not a single number, but a set of basic summary statistics, such as minimum, median, average, and standard deviation.

Each model takes two sets of kernel arguments into account:

- Flag and scalar arguments (and increment arguments for vector operations) are limited to a few discrete values: the distinct options for flags and the values -1 , 0 , 1 , and “any other value” for scalars (and either 1 or a “any large value” for increments). For a given kernel invocation, the combination of these argument values identifies one of several discrete *cases* . To best match the application scenario, each model can be configured to represent only a subset of these cases.
- Size arguments take values from potentially large ranges of problem sizes. In our models, these represented ranges are specified as (collections of) rectangular (generally: hyper-cuboidal) *domains* . For each model and case, these domains can be separately selected.

All other arguments, such as data arguments and leading dimensions, are not represented in our models.

For each case and domain, we generate a separate *sub-model* that represents the kernel runtime as a *piecewise polynomial*. Each polynomial piece  actually consists of a small list of polynomials corresponding to the modeled runtime summary statistics.

Since implementing the composition of models from sub-models and the corresponding separation and treatment of argument types is fairly straight forward, the following sections focuses on the generation of a single sub-model.

⁵ Optionally further performance counters provided by the SAMPLER or derived metrics can be modeled. However, throughout this work we solely focus on runtime models.

3.2.2 Sample Distribution

For a fixed setup, discrete case, and rectangular domain, we model a kernel's runtime by taking a series of measurements—referred to as samples—and fitting a polynomial to the measured runtime. The first step is to select a *sampling point distribution*, i.e., a set of points in the domain at which the kernel runtime is measured.

An intuitive option would be to (pseudo-)randomly distribute the sampling points within the domain. However, this approach does not guarantee that, e.g., points close to the domain's boundary are well represented in the sampling set, which in these areas greatly reduces the accuracy of polynomials fitted to such data. Hence we do not use random sampling point distributions, and instead consider two regular grid patterns:

- The simplest structured pattern is a regular *Cartesian grid* that covers the whole domain evenly with points. In one dimension, a Cartesian grid of n points x_0, \dots, x_{n-1} between 0 and 1 is defined as

$$x_i = \frac{i}{n-1} \ .$$

With regards to the adaptive refinement approach (see [Section 3.2.5](#)), the Cartesian grid's advantage is its high *sample reuse*: When the domain is divided in two along one dimension, all points of the original grid are also points in the two new grids. Hence, the number of points in which new measurement are required is reduced significantly.

- However, fitting a polynomial behavior with an even distribution of samples is not ideal. A better alternative is to use *Chebyshev nodes* [29, Section 8.3], which minimize the approximation error by essentially moving the sampling points closer to the region's boundaries. In one dimension, the n Chebyshev nodes x_0, \dots, x_{n-1} between -1 and 1 are given by

$$x_i = \cos\left(\frac{2i+1}{2n}\pi\right) \ .$$

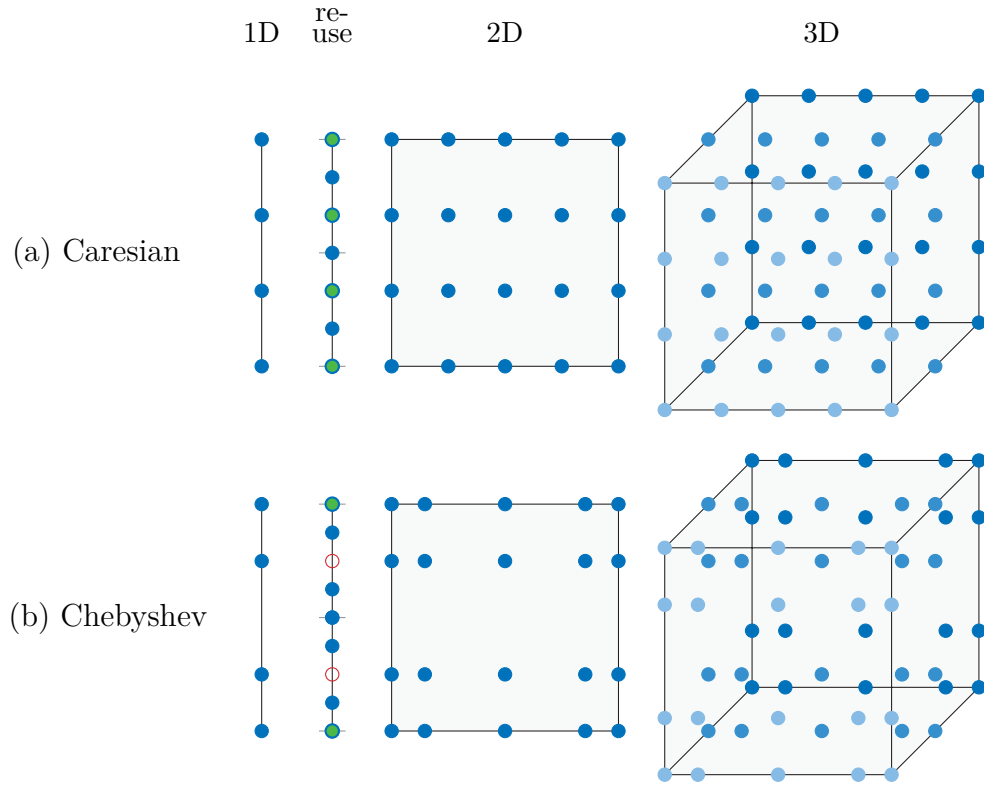


Figure 3.10: Sampling point distributions and reuse.

In contrast to the Cartesian grid with perfect sample reuse, the Chebyshev nodes offer no reuse at all. Furthermore, they do not include points on the domain's boundary. We hence use a slightly modified configuration that moves the Chebyshev nodes to include the boundary:

$$x_i = \cos\left(\frac{i}{n-1}\pi\right) \ .$$

We refer to these points as a *Chebyshev grid*.

Example 3.11: Sampling point distributions

Figure 3.10 visualizes the two alternative sampling point distributions for 1D, 2D, and 3D domains. We select 4 points along the first dimension, 5 along the second, and 3 along the third.

The point reuse is shown for the 1D case: When the domain is split in

half, all points from the original Cartesian grid ● are reused in the refined grid, and only three new points ● are generated; for the Chebyshev grid, however, only the two outermost points ● are reused, while the other two ○ are not matched by points in the refined grid, and five new points ● are generated.

Once the sampling points are chosen, we avoid implementation-dependent performance artefacts of size argument increments in steps of 1 (see [Section 3.1.5.1](#)) by *rounding* all generated grid points to *multiples of 8* along each dimension.

3.2.3 Repeated Measurements and Summary Statistics

Based on the kernel and the modeled cases, each sampling point is turned into a *measurement call*: While the flag, size, and scalar arguments are determined by the case and the point, the leading dimensions are set to a fixed large value (such as 5000), and the operand sizes are deduced automatically.

To both avoid outliers and represent measurement fluctuations in our models, each such constructed measurement call is then executed by the SAMPLER (see [Section 2.2.1](#)) not only once, but *repeatedly*—typically between 5 and 20 times. To avoid the effects of frequency fluctuations (see [Section 2.1.2.2](#) and [2.1.2.3](#)), the repetitions for each measurement call are not executed in a single batch but shuffled among all calls’ repetitions to obtain measurements across the whole SAMPLER execution for each call. Furthermore, each repetition, executes the measurement call twice in a row, to ensure consistent cache preconditions, which offer high temporal locality (“warm” data) for small operations.

Once obtained, the collected measurement results for each call are turned into *summary statistics*: minimum, median, maximum, average, and standard deviation. In the next step, each of these statistics is fitted with a separate polynomial.

3.2.4 Relative Least-Squares Polynomial Fitting

The starting point for the polynomial fitting procedure is a set of sampling *points* $\mathbf{x}_1, \dots, \mathbf{x}_N \in \mathbb{R}^d$ (from the d -dimensional range of size arguments) and corresponding measurement *values* $y_i \in \mathbb{R}$ (i.e., per summary statistic).⁶ As the set of polynomial basis functions, we use *monomials* $m_1, \dots, m_M: \mathbb{R}^d \rightarrow \mathbb{R}$ whose maximum *degree* is determined by the kernel's asymptotic complexity (given by its minimal FLOP-count), yet may be further increased. The polynomial p is constructed as a linear combination of these monomials with *weights* $\beta_1, \dots, \beta_M \in \mathbb{R}$:

$$p(\mathbf{x}) = \sum_{j=1}^M \beta_j m_j(\mathbf{x}) .$$

Example 3.12: Polynomial basis functions

If we model the runtime of `dt_rsmLLNN` by letting its cost of m^2n FLOPs determine the maximum monomial degree, we use a bivariate polynomial in $\mathbf{x} = (x_1, x_2)$ of the form

$$p(\mathbf{x}) = \beta_1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_1^2 + \beta_5 x_1 x_2 + \beta_6 x_1^2 x_2 = \sum_{j=1}^6 \beta_j m_j(\mathbf{x}) ,$$

i.e., with the monomial basis

$$\begin{array}{lll} m_1(\mathbf{x}) = 1 & m_2(\mathbf{x}) = x_1 & m_3(\mathbf{x}) = x_2 \\ m_4(\mathbf{x}) = x_1^2 & m_5(\mathbf{x}) = x_1 x_2 & m_6(\mathbf{x}) = x_1^2 x_2 . \end{array}$$

Had we chosen to increase the monomial degree in each dimension by one, we would use a polynomial with the 12 basis monomials:

$$\begin{array}{lll} m_1(\mathbf{x}) = 1 & m_2(\mathbf{x}) = x_1 & m_3(\mathbf{x}) = x_2 \\ m_4(\mathbf{x}) = x_1^2 & m_5(\mathbf{x}) = x_1 x_2 & m_6(\mathbf{x}) = x_2^2 \\ m_7(\mathbf{x}) = x_1^3 & m_8(\mathbf{x}) = x_1^2 x_2 & m_9(\mathbf{x}) = x_1 x_2^2 \end{array}$$

⁶ Technically, we have $\mathbf{x}_i \in \mathbb{N}_0^d$ and $y \in \mathbb{N}_0$; however, for the fitting procedure these points are treated as floating-point tuples.

$$m_{10}(\mathbf{x}) = x_1^3 x_2 \quad m_{11}(\mathbf{x}) = x_1^2 x_2^2 \quad m_{12}(\mathbf{x}) = x_1^3 x_2^2 .$$

The weights β_j are chosen by *minimizing the squared relative error*

$$S(\beta_1, \dots, \beta_M) \stackrel{\text{def}}{=} \sum_{i=1}^N \left(\frac{y_i - p(\mathbf{x}_i)}{y_i} \right)^2 = \sum_{i=1}^N \left(1 - \sum_{j=1}^M \frac{\beta_j m_j(\mathbf{x}_i)}{y_i} \right)^2 .$$

With

$$\beta \stackrel{\text{def}}{=} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_M \end{pmatrix} \text{ and } X \stackrel{\text{def}}{=} \begin{pmatrix} \frac{m_1(\mathbf{x}_1)}{y_1} & \frac{m_2(\mathbf{x}_1)}{y_1} & \dots & \frac{m_M(\mathbf{x}_1)}{y_1} \\ \frac{m_1(\mathbf{x}_2)}{y_2} & \frac{m_2(\mathbf{x}_2)}{y_2} & \dots & \frac{m_M(\mathbf{x}_2)}{y_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{m_1(\mathbf{x}_N)}{y_N} & \frac{m_2(\mathbf{x}_N)}{y_N} & \dots & \frac{m_M(\mathbf{x}_N)}{y_N} \end{pmatrix} ,$$

this error can be expressed as

$$\begin{aligned} S(\beta) &= \|1 - X\beta\|^2 \\ &= (1 - X\beta)^T (1 - X\beta) \\ &= 1 - 2\beta^T X^T 1 + \beta^T X^T X \beta . \end{aligned}$$

Since $S(\beta)$ is convex, we can find its minimum by setting its derivative to zero:

$$\frac{\partial S(\beta)}{\partial \beta} = -2X^T 1 + 2X^T X \beta = 0 .$$

Rewritten as

$$(X^T X) \beta = X^T 1 ,$$

this is known as the *normal equations*, which have a unique solution because, since the m_j are linearly independent, X has full rank. To obtain a numerically stable solution of the normal equations, we use NUMPY's `linalg.lstsq`, which is based on the singular value decomposition of X .

3.2.5 Adaptive Refinement

So far, we have determined how sampling points are chosen in a given rectangular (generally: hyper-cuboidal) domain, how summary statistics are computed from repeated measurements in these points, and how a multivariate polynomial is fitted to one of these statistics. We now describe how a domain is adaptively subdivided and fitted with a piecewise function consisting of such polynomials.

The basis for this adaptive subdivision is an *error measure* for the approximation accuracy. To compute this measure, we consider the polynomial fit of a selected *reference statistic*; typical choices are the minimum or median since they are insensitive to fluctuations. For the selected reference statistic, we compute the point-wise absolute relative error e_i for the polynomial approximation p in each measurement point \mathbf{x}_i with respect to the measurement statistic y_i :

$$e_i \stackrel{\text{def}}{=} \left| \frac{y_i - p(\mathbf{x}_i)}{y_i} \right| .$$

Next, the error measure is computed from the set of errors $\{e_1, \dots, e_N\}$ as its average, maximum, or ninetieth percentile.

Based on this error measure, the *adaptive refinement* process subdivides the initial domain as follows: It starts by sampling the entire domain and fits one polynomial to all measurements (for each statistic). If either the error measure for this approximation is below a specified *error bound* (i.e., a threshold value) or the size of the domain along each dimension is below a configurable *minimum width*, the process terminates. Otherwise, the domain is split in half along its *relatively largest dimension*: If along each dimension i the domain spans the interval $[l_i, u_i]$, we choose the dimension for which u_i/l_i is the largest. Along this dimension s , the new interval is split in half⁷ (rounded to the nearest

⁷ We choose the interval's center, since it guarantees the most regular subdivision. A more guided choice would require either advanced knowledge of the kernel implementation or a significantly higher sampling resolution.

multiple of 8) at

$$m_s \stackrel{\text{def}}{=} \text{round} \left(\frac{l_s + u_s}{2}, 8 \right) = 8 \left\lfloor \frac{l_s + u_s + 8}{16} \right\rfloor ,$$

and the new domains are defined by the intervals $[l_s, m_s]$ and $[m_s, u_s]$. The process is applied recursively to both new domains until either the error bound or the minimum width is reached.

Note that the resulting performance models are not smooth because the polynomial pieces are not required to match at the boundaries. Since our applications do not require any continuity in our models, this does not pose a problem. Hence, we do not apply, e.g., splines to generate smooth models at increased cost.

Example 3.13: Adaptive refinement

Figure 3.11 illustrates the adaptive refinement process for

`side uplo transA diag m n alpha A ldA B ldB`
`dtrsm(L , L , N , N , m, n, 1.0, A, 5000, B, 5000) ,`

i.e., $\underline{B} := \underline{A}^{-1} \underline{B}$ with $\underline{A} \in \mathbb{R}^{m \times m}$ and $\underline{B} \in \mathbb{R}^{m \times n}$, with $m \in [24, 536]$ and $n \in [24, 4152]$ on a SANDY BRIDGE-EP E5-2670 with single-threaded OPENBLAS. We use a Chebyshev sampling point distributions with 6 and 5 values along, respectively, dimensions m and n , and apply adaptive refinement to fit a piecewise polynomial to the minimum of 15 measurement repetitions until either the maximum error across the sampling points falls below 1 % or all domain dimensions fall below 64.

The initial distribution of sampling points ● is shown in Figure 3.11a. The polynomial fit to samples in these points has an error measure of 4.21 %. Since this exceeds the error bound of 1 %, the domain is split in half along the (relatively) larger dimension n at $n = \frac{4152+24}{2} = 2088$.

The sampling points for the two new domains are displayed in Figure 3.11b; the error measure for their newly fitted polynomials is 3.36 % ($n \leq 2088$)

3 Performance Modeling

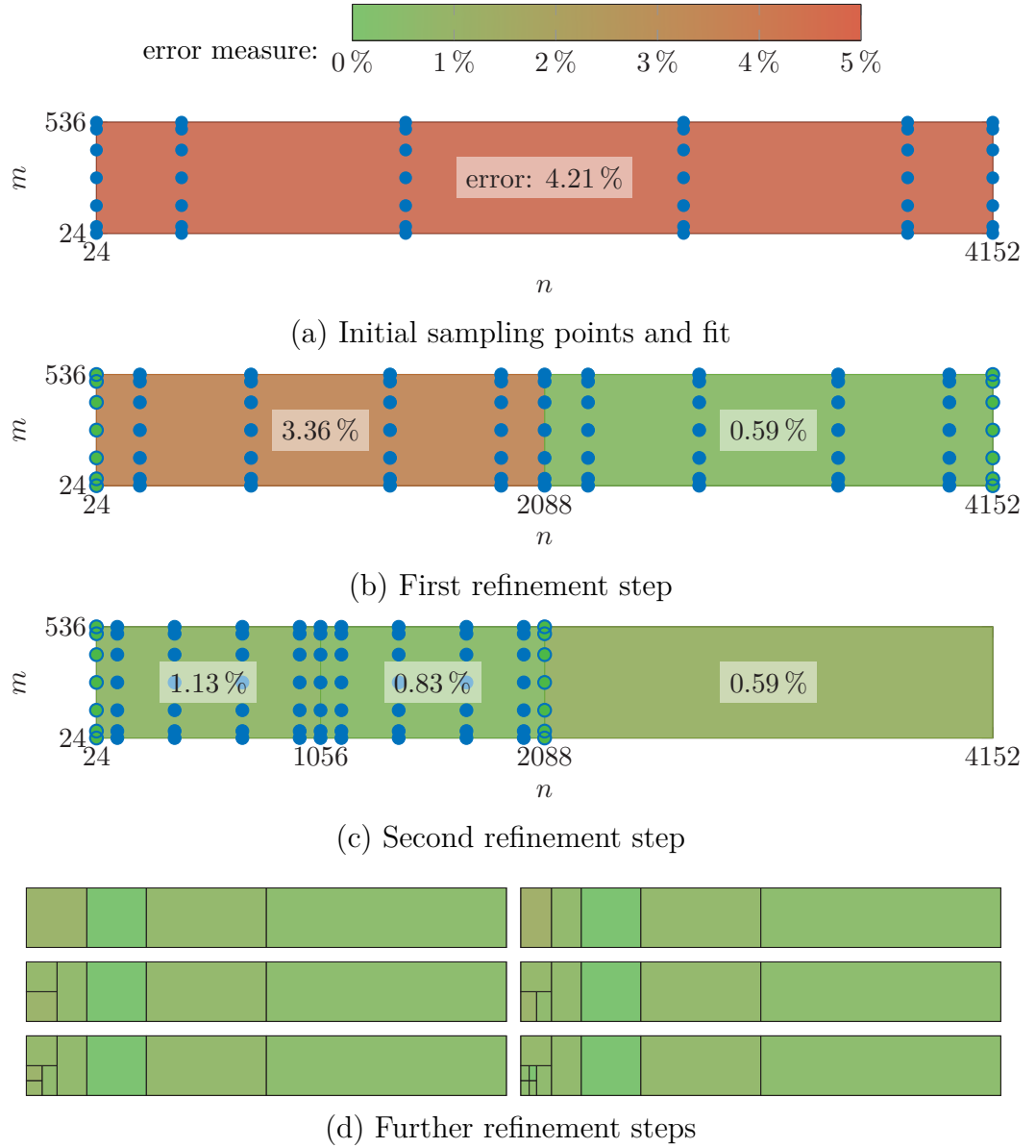


Figure 3.11: Modeling through adaptive refinement for `dtrsmLLNN`.

(SANDY BRIDGE-EP [E5-2670](#), 1 thread, OPENBLAS)

and 0.59 % ($n \geq 2088$). While the latter is already below the error bound of 1 %, the approximation for $n \leq 2088$ is further refined.

After the next refinement step (Figure 3.11c) the error is reduced to 1.13 % ($n \leq 1056$) and 0.83 % ($1056 \leq n \leq 2088$). As illustrated in Figure 3.11d, further steps are applied until the error measure is globally below 1 % after a total 8 refinements—the process was solely terminated by globally reaching the target error bound and not the minimum width of 64.

While the sampler configuration in this example was chosen to demonstrate the adaptive refinement process, the increased number of polynomial pieces for smaller problem sizes is typical and in practice commonly triggers the minimum-width termination criterion. However, for kernels with a cubic asymptotic complexity (such as BLAS Level 3), generating models for such small problem sizes is quite cheap compared to larger sizes.

With the adaptive refinement procedure, we can now generate models for a wide range of dense linear algebra kernels, and proceed to take a closer look at the generated models.

3.3 Model Generator Configuration

We now discuss the configuration options of the adaptive refinement process, and examine how they affect the model accuracy and generation cost. We then select a default configuration to generate the models used for our performance predictions in Chapter 4.

3.3.1 Configuration Parameters

The adaptive refinement is controlled by a total of eight *configuration parameters*. They allow to control the model accuracy, but also affect the time spent for the required measurements. The eight parameters regulate the model generation as follows:

- To represent the runtime of a kernel, the monomial basis for the fitted polynomials needs to at least cover the kernel’s asymptotic complexity (i.e.,

3 Performance Modeling

its minimal FLOP-count). To better represent performance variations, however, the maximum degree of the monomials can be increased in each each dimension (i.e., size argument). We refer to this increase as *overfitting*; practical values are *between 0 and 2*.

- To fit a polynomial to a routine's runtime, the number of sampling points along each dimension needs to be at least one more than the corresponding polynomial degree. However, since this minimal number of points yields a polynomial that fits the measurements perfectly, we cannot use it to compute an approximation error. We hence increase the number of sampling points per dimension by at least one, and to further improve the approximation accuracy, further points can be added; we refer to the total number of points added as *oversampling*; practical values are values *between 1 and 10*.
- We introduced two alternatives to *distribute* sampling points on *grids* that cover the domains of problem sizes: a *Cartesian* grid and a *Chebyshev* grid.
- For each sampling point, we perform several *measurement repetitions*; practical values are *between 5 and 20*.
- From the repetitions, we compute several runtime summary statistics: minimum, median, maximum, average, and standard deviation. One of these is selected as the *reference statistic*; practical choices are the *minimum and median*.
- From the absolute relative errors in the reference statistic for all sampling points, we compute the *error measure* which is these relative errors' *average, maximum, or 90th percentile*.
- The first termination criterion for the adaptive refinement process is the approximation accuracy: The refinement stops when the computed error measure is below a *target error bound*; practical values for this bound are *between 1 % and 5 %*.

- The second termination criterion is the size of the domains: The refinement stops when a new domain is smaller than a *minimum width* along all dimensions; typical values are *32 and 64*.

3.3.2 Trade-Off and Configuration Selection

In the following, we analyze the accuracy of our models and their generation cost, and select a configuration to generate the models for the performance predictions in the [Chapter 4](#).

We consider the model generation for

side uplo transA diag m n alpha A IdA B IdB
`dtrsm(L , L , N , N , m, n, 1.0 , A, 5000, B, 5000) ,`

i.e., $\boxed{B} := \boxed{A}^{-1} \boxed{B}$ with $\boxed{A} \in \mathbb{R}^{m \times m}$ and $\boxed{B} \in \mathbb{R}^{m \times n}$, for sizes $m \in [24, 536]$ and $n \in [24, 4152]$ on a SANDY BRIDGE-EP [E5-2670](#) and a HASWELL-EP [E5-2680 v3](#) using single-threaded OPENBLAS, BLIS, and MKL.

For each setup, our first step is to exhaustively measure the `dtrsmLLNN`'s runtime 15 times in all points (m, n) in the domain $[24, 536] \times [24, 4152]$ at which both m and n are multiples of 8—a total of 504 075 measurements. These measurements are used both as the basis for our model generation and to evaluate the model accuracy across the entire domain (contrary to the model generation, which can only evaluate the error in its sampling points).

We generate models for all 2880 configurations obtained from combining the parameter values shown in [Table 3.1](#). These configurations result in a wide range of models with significantly different accuracies and generation costs. To evaluate them, we quantify the *model error* as the averaged relative error of the predicted minimum runtime $p(\mathbf{x}_i)$ relative to the measured minimum y_i across all $N = 33\,605$ points \mathbf{x}_i of the domain:

$$\text{model error} \stackrel{\text{def}}{:=} \frac{1}{N} \sum_{i=1}^N \frac{|p(\mathbf{x}_i) - y_i|}{y_i} ;$$

3 Performance Modeling

parameter	values
overfitting	0, 1, 2
oversampling	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
distribution grid	Cartesian, Chebyshev
measurement repetitions	5, 10, 15
reference statistic	minimum, median
error measure	90th percentile, maximum
target error bound	1 %, 2 %
minimum width	32, 64

Table 3.1: Configuration parameters for the model generation and their studied values.

	model error		model cost	
	minimum	maximum	minimum	maximum
overfitting	1	0	0	1
oversampling	10	2	1	9
distribution grid	Cartesian	Chebyshev	Cartesian	Cartesian
measurement repetitions	15	5	5	5
reference statistic	median	minimum	minimum	median
error measure	maximum	90th perc.	maximum	maximum
target error bound	1 %	2 %	2 %	1 %
minimum width	32	32	64	32
model error	0.12 %	0.92 %	0.73 %	0.22 %
model cost	5.48 min	1.68 s	0.96 s	15.49 min

Table 3.2: Model configuration parameters for minimum and maximum error and cost.

(SANDY BRIDGE-EP [E5-2670](#), 1 thread, OPENBLAS)

furthermore, we define the *model cost* as the total runtime of the required measurements used as samples.

Example 3.14: Model accuracy

[Figure 3.12](#) shows the structure and point-wise accuracy of the four models with minimum and maximum accuracy and cost for single-threaded OPENBLAS on a SANDY BRIDGE-EP [E5-2670](#); [Table 3.2](#) lists the corresponding configurations. Both the cheapest and least accurate model use only a single

3.3 Model Generator Configuration

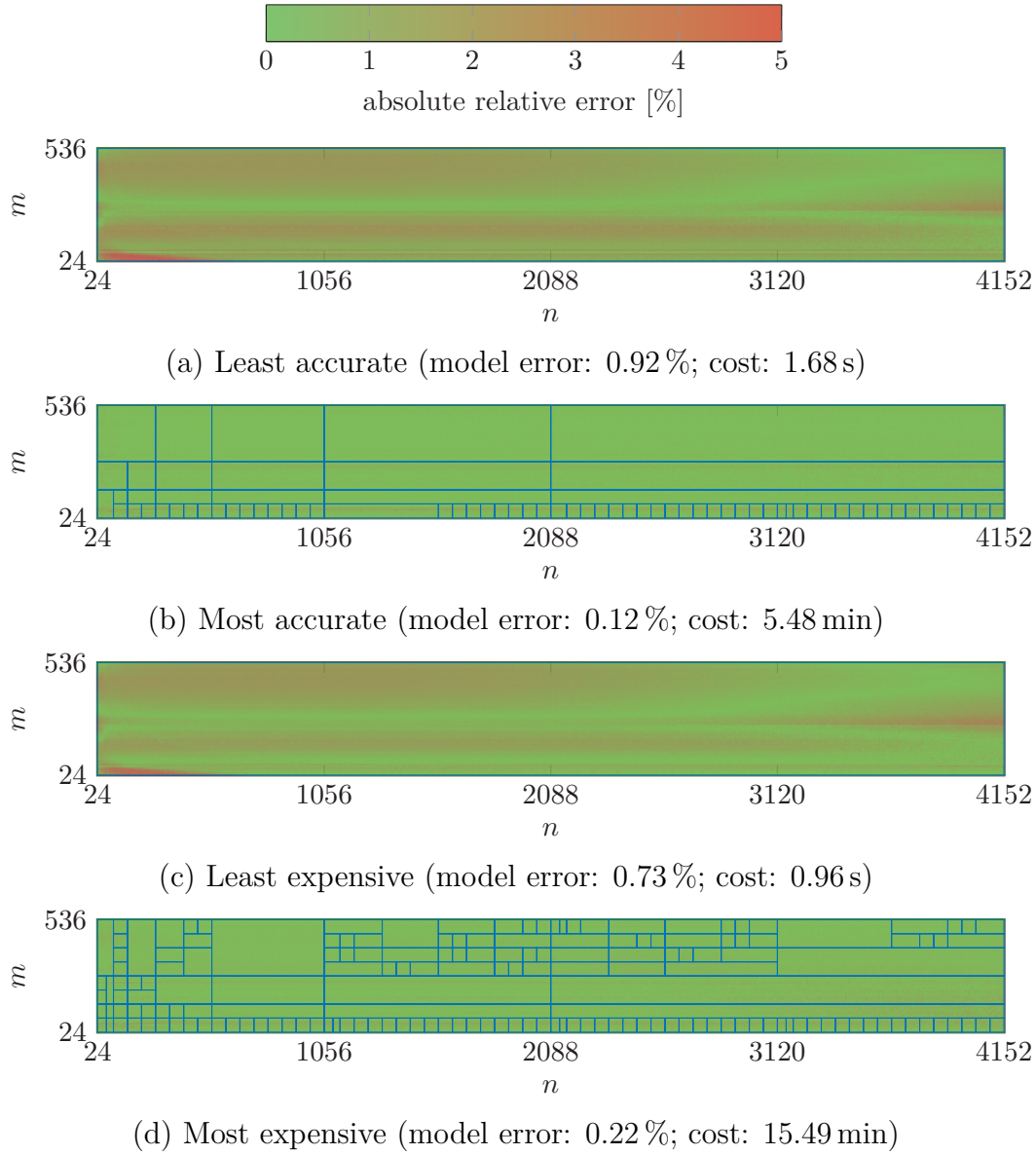


Figure 3.12: Accuracy and structure of models for `dtrsmLLNN`.

(SANDY BRIDGE-EP [E5-2670](#), 1 thread, OPENBLAS)

3 Performance Modeling

polynomial for the entire domain but also offer only poor accuracy. The expensive and accurate models on the other hand subdivide the domain repetitively, and thus find a better fitting piecewise polynomial.

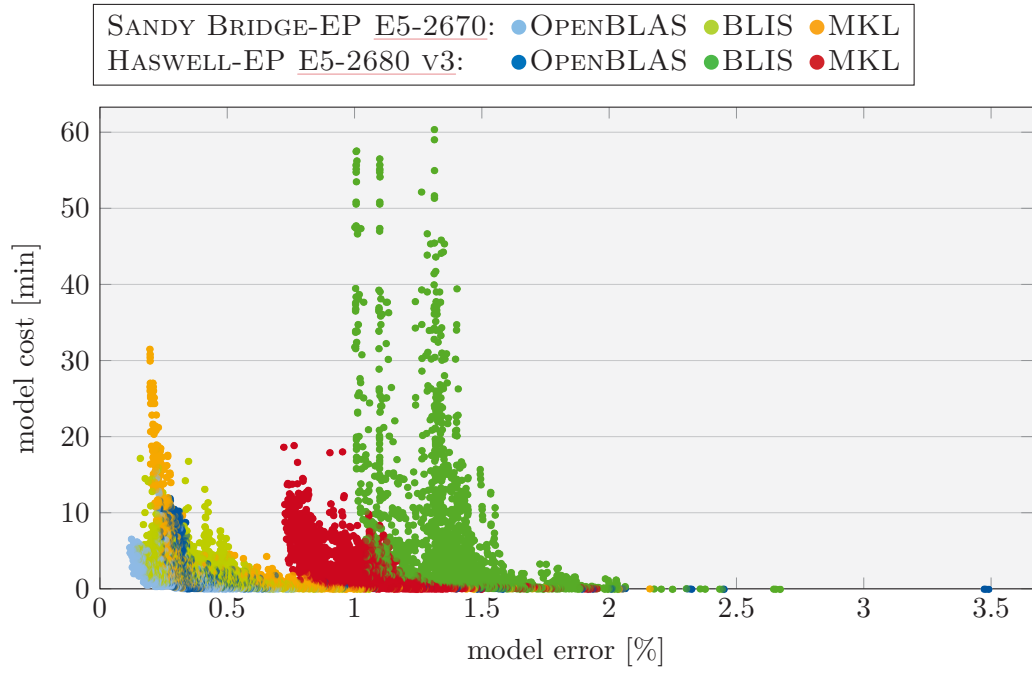
The accuracy and cost of all 2880 generated models for each setup are presented in [Figure 3.13a](#); in this plot, the preferable models with low error and cost are found close to the origin. All setups share the same general trend: Models with low accuracy are quite cheap, while models with high accuracy are more expensive. Hence we are faced with a *trade-off between accuracy and cost*. However, selecting a good configuration is not straight-forward: Models with practically identical accuracy are up to a factor of 16 apart in generation cost, and a cheap and accurate configuration for one setup may be neither for other setups. In the following, we describe how we approach the search-space of all considered configurations, and identify a desirable default configuration that we subsequently use to generate the models for all setups and kernels needed for our performance predictions in [Chapter 4](#).

Before we begin to reduce our search space, we notice that on the HASWELL, the models for both BLIS (●) and MKL (●) are on average less than half as accurate than for the other setups. The cause is a rather jagged performance behavior, which is difficult to represent accurately. Hence, to identify a good default configuration, we consider only the SANDY BRIDGE (●, ●, ●) and OPENBLAS on the HASWELL (●).

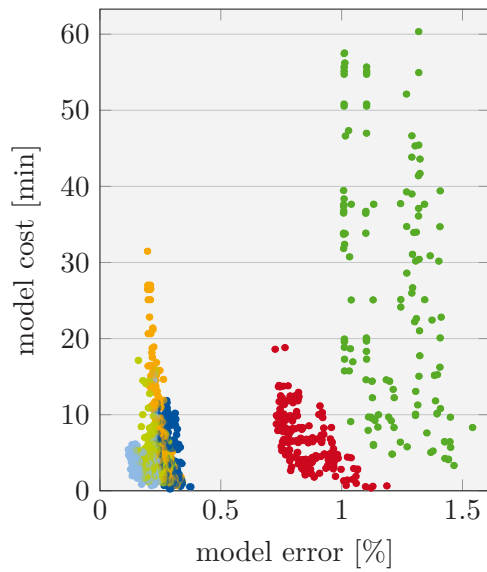
Our first step is to *prune by accuracy*: We discard any configuration that for any of the considered setups yields a model error larger than $1.5\times$ the minimum error for that setup; in other words, all remaining configurations generate models that are at most 50 % less accurate than the most accurate model. This step reduces the number of potential configurations from 2880 to 163; all remaining configurations use an oversampling value of 3 or higher, and a target error bound of 1 %. [Figure 3.13b](#) shows the 163 remaining models' accuracy and cost.

Our second step is to similarly *prune by cost*: We discard any configuration that for any considered setup takes longer than the first quartile in generation

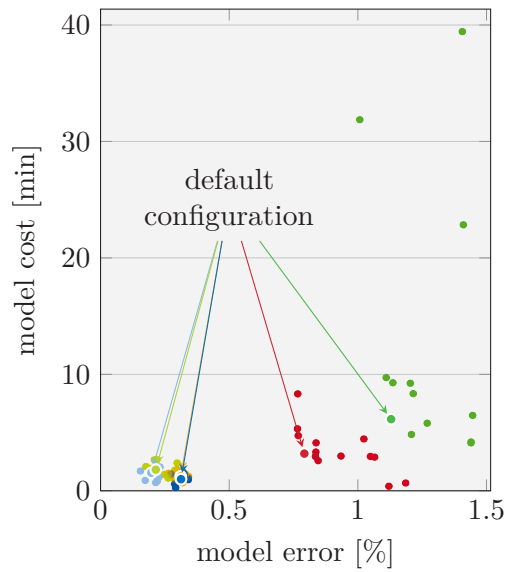
3.3 Model Generator Configuration



(a) All 2880 configurations



(b) Within $2\times$ of most accurate



(c) Below 10th percentile in cost

Figure 3.13: Model configuration trade-off in accuracy versus cost and steps towards selecting a default configuration.

(1 thread, error in the minimum measure)

3 Performance Modeling

	overfitting	oversampling	distribution grid	measurement repetitions	reference statistic	error measure	target error bound	minimum width
(1)	0	4	Chebyshev	10	median	maximum	1 %	32
(2)	0	4	Chebyshev	15	minimum	maximum	1 %	32
(3)	1	5	Chebyshev	10	median	maximum	1 %	32
(4)	1	5	Chebyshev	10	minimum	maximum	1 %	32
(5)	1	8	Cartesian	5	minimum	maximum	1 %	32
(6)	2	4	Cartesian	5	median	maximum	1 %	64
(7)	2	4	Cartesian	10	median	maximum	1 %	32
(8)	2	4	Chebyshev	10	median	maximum	1 %	32
(9)	2	4	Chebyshev	10	median	maximum	1 %	64
(10)	2	4	Chebyshev	10	minimum	maximum	1 %	32
(11)	2	4	Chebyshev	10	minimum	maximum	1 %	64
(12)	2	4	Chebyshev	15	minimum	maximum	1 %	32
(13)	2	4	Chebyshev	15	minimum	maximum	1 %	64
(14)	2	7	Cartesian	10	minimum	maximum	1 %	32

Table 3.3: Model generator configurations remaining after pruning.

Bold: majority value. **Blue:** default configuration.

time for that setup; in other words, the remaining models are all within the 25 % that are generated the fastest. This step further reduces the number of potential configurations from 163 to 14, as shown in [Figure 3.13c](#).

The parameter values for the 14 remaining configurations are shown in [Table 3.3](#). For each parameter, we can find one value that is common to at least 8 of the 14 configurations (highlighted in **bold**). We choose our *default configuration* by selecting this most common value for each parameter. It corresponds to line (10) in [Table 3.3](#) (highlighted in **blue**), and is marked for each setup in [Figure 3.13c](#). Note that it also serves as a good choice between accuracy and cost for BLIS (●) and MKL (●) on the HASWELL, which were not included in the pruning process.

3.3.3 Variations of the Default Configuration

While the configuration was found to yield good accuracies at reasonable costs for almost all encountered kernels, it proves to be quite expensive for kernels with *three degrees of freedom*, which for the predictions in [Chapter 4](#) only applies to *dgemm* with its three size arguments *m*, *n*, and *k*. To reduce the modeling cost for this kernel, we adjust the default configuration by reduce the overfitting from 2 to 0, and increasing the minimum width from 32 to 64.

Furthermore, the performance of BLAS kernels becomes less smooth when we bring *multi-threading* into the picture. Hence, to avoid excessive partitioning as seen in [Figure 3.12d](#), we increase the minimum width for all models to 64, and for *dgemm* to 256.

3.4 Summary

This chapter first studied the effects of various kernel argument types on performance, and then introduced the structure of our performance models and their automated measurement-based generation. Since this generation process offers various configuration parameters, we studied the trade-off between the resulting models' accuracy and generation cost, and concluded with the selection of default configurations, which are used to generate all models for the following chapter's performance predictions.

4 Model-Based Predictions for Blocked Algorithms

With accurate performance models at hand, we predict the runtime and performance of blocked algorithms in order to both select the fastest algorithm for a given operation from available alternatives and tune its block size. We thereby arrive at a near-optimal solution entirely without executing any of the potential algorithms and configurations; compared to tuning through empirical measurements, accurate model-based performance predictions are orders of magnitude faster.

For this chapter, we generated performance models to predict all studied algorithms with problem sizes up to $n = 4152$ and block sizes between $b = 24$ and 536. E.g., our models for `dtrsm` each cover 4 cases (combinations of flag argument values) and domains (ranges of problem sizes) of size $[24, 536] \times [24, 4152]$.

We begin by introducing runtime, performance, and efficiency predictions for executions of blocked algorithms in [Section 4.1](#), followed by accuracy metrics for such predictions in [Section 4.2](#). Next, we present a detailed study on the prediction accuracy for a blocked Cholesky decomposition under various conditions in [Section 4.3](#) and a broader accuracy evaluation for a range of blocked LAPACK algorithms in [Section 4.4](#). We then apply our predictions to identify the fastest blocked algorithms for different operations in [Section 4.5](#), and finally determine near-optimal block sizes for a range of algorithms in [Section 4.6](#).

Publication

The work presented in this chapter is in parts based on research previously published in:

- [1] Elmar Peise and Paolo Bientinesi. “Algorithm 979: Recursive Algorithms for Dense Linear Algebra—The ReLAPACK Collection”. In: *ACM Trans. Math. Softw.* 44.2 (Sept. 2017), 16:1–16:19. DOI: [10.1145/3061664](https://doi.org/10.1145/3061664).
- [8] Elmar Peise and Paolo Bientinesi. *Cache-aware Performance Modeling and Prediction for Dense Linear Algebra*. Technical report. AICES, RWTH Aachen University, Nov. 2014. arXiv: [1409.8602](https://arxiv.org/abs/1409.8602) [[cs.PF](#)].
- [11] Elmar Peise and Paolo Bientinesi. “Performance Modeling for Dense Linear Algebra”. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. SCC ’12. IEEE Computer Society, Nov. 2012, pages 406–416. DOI: [10.1109/SC.Companion.2012.60](https://doi.org/10.1109/SC.Companion.2012.60).
- [12] Elmar Peise. “Hierarchical Performance Modeling for Ranking Dense Linear Algebra Algorithms”. Master’s thesis. Aachen Institute for Computational Engineering Science, RWTH Aachen, May 2012. arXiv: [1207.5217](https://arxiv.org/abs/1207.5217) [[cs.PF](#)].

4.1 Performance Prediction

Based on our performance models, we now predict the runtime and performance of individual blocked algorithm executions. For each algorithm, the problem size and the block size uniquely determine the exact *sequence of calls* (i.e., kernel invocations). For each call \mathcal{C} in this sequence and a selected hardware and software setup, our performance models provide a *runtime estimate* $t_{\text{est}}(\mathcal{C})$. Summing these estimates yields our *runtime prediction*

$$t_{\text{pred}} \stackrel{\text{def}}{=} \sum_{\text{calls } \mathcal{C}} t_{\text{est}}(\mathcal{C}) . \quad (4.1)$$

Example 4.1: Runtime prediction

Table 4.1 lists the sequence of calls to invert a lower-triangular matrix of size $n = 800$ (i.e., $\underline{A} := \underline{A}^{-1}$ with $\underline{A} \in \mathbb{R}^{800 \times 800}$) using blocked algorithm 1 (Figure 1.1b on Page 5) with block size $b = 300$; for each call the table’s

4.1 Performance Prediction

step	call \mathcal{C}	$t_{\text{est}}(\mathcal{C})$
1	<small>side uplo transA diag m n alpha A ldA B ldB</small> <code>dtrmm(R , L , N , N , 300, 0, 1.0, A_{00}, 800, A_{10}, 800)</code>	0.00 ms
	<small>side uplo transA diag m n alpha A ldA B ldB</small> <code>dtrsm(L , L , N , N , 300, 0, -1.0, A_{11}, 800, A_{10}, 800)</code>	0.00 ms
	<small>uplo n A ldA</small> <code>dtrti2(L , 300, A_{11}, 800)</code>	2.64 ms
2	<small>side uplo transA diag m n alpha A ldA B ldB</small> <code>dtrmm(R , L , N , N , 300, 300, 1.0, A_{00}, 800, A_{10}, 800)</code>	1.71 ms
	<small>side uplo transA diag m n alpha A ldA B ldB</small> <code>dtrsm(L , L , N , N , 300, 300, -1.0, A_{11}, 800, A_{10}, 800)</code>	2.07 ms
	<small>uplo n A ldA</small> <code>dtrti2(L , 300, A_{11}, 800)</code>	2.64 ms
3	<small>side uplo transA diag m n alpha A ldA B ldB</small> <code>dtrmm(R , L , N , N , 200, 600, 1.0, A_{00}, 800, A_{10}, 800)</code>	4.15 ms
	<small>side uplo transA diag m n alpha A ldA B ldB</small> <code>dtrsm(L , L , N , N , 200, 600, -1.0, A_{11}, 800, A_{10}, 800)</code>	2.17 ms
	<small>uplo n A ldA</small> <code>dtrti2(L , 200, A_{11}, 800)</code>	0.85 ms
$t_{\text{pred}}:$		16.22 ms

Table 4.1: Sequence of calls, runtime estimates, and accumulated prediction for the inversion of a lower-triangular matrix with blocked algorithm 1. ($n = 800$, $b = 300$, SANDY BRIDGE-EP E5-2670, OPENBLAS, 1 thread, statistic: median)

last column presents median runtime estimates from performance models for a SANDY BRIDGE-EP E5-2670 with single-threaded OPENBLAS. The sum of these estimates is our runtime prediction for the entire algorithm: $t_{\text{pred}} = 16.22$ ms.

Note that with block size $b = 300$, the algorithm traverses the input matrix of size $n = 800$ in three steps, and in each step the sub-matrices A_{00} , A_{10} , and A_{11} refer to different portions of $\begin{bmatrix} A \\ \end{bmatrix}$, i.e., after every three calls in Table 4.1. As a result, the first two calls perform no operations since their size arguments n are 0 (i.e., their operand operand A_{10} has a width of 0); hence their estimated runtime is 0 ms.

Our performance models estimate the runtime of kernel invocations not as a single number but as a range of summary statistics: minimum t_{est}^{\min} , median $t_{\text{est}}^{\text{med}}$, maximum t_{est}^{\max} , mean (average) t_{est}^{μ} , and standard deviation t_{est}^{σ} . Each of these

statistics is also available for our *prediction*:

$$t_{\text{pred}}^s \stackrel{\text{def}}{=} \sum_{\text{calls } \mathcal{C}} t_{\text{est}}^s(\mathcal{C}) \quad \text{for } s \in \{\min, \text{med}, \max, \mu\} , \quad (4.2)$$

$$t_{\text{pred}}^\sigma \stackrel{\text{def}}{=} \sqrt{\sum_{\text{calls } \mathcal{C}} t_{\text{est}}^\sigma(\mathcal{C})^2} . \quad (4.3)$$

Note that the definition for the standard deviation t_{pred}^σ assumes uncorrelated estimates $t_{\text{est}}^\sigma(\mathcal{C})$.

Example 4.2: Prediction summary statistics

For the algorithm execution in [Example 4.1](#), our predictions yield the following summary statistics:

$$\begin{aligned} t_{\text{pred}}^{\min} &= 16.18 \text{ ms} & t_{\text{pred}}^{\text{med}} &= 16.22 \text{ ms} & t_{\text{pred}}^{\max} &= 16.46 \text{ ms} \\ t_{\text{pred}}^\mu &= 16.25 \text{ ms} & t_{\text{pred}}^\sigma &= 95.88 \text{ } \mu\text{s} . \end{aligned}$$

The predictions indicate only minimal runtime fluctuations: The predicted standard deviation t_{pred}^σ is only 0.59 % of the mean t_{pred}^μ .

Predictions for derived metrics, such as performance and (compute-bound) efficiency, are obtained from the runtime prediction in combination with properties of the operation and the execution hardware (see [Appendix A](#)):

- The *performance prediction* p_{pred} is computed from the runtime prediction and the operation's cost (i.e., minimal FLOP-count):

$$p_{\text{pred}}^{\min} \stackrel{\text{def}}{=} \frac{\text{cost}}{t_{\text{pred}}^{\max}} \quad p_{\text{pred}}^{\text{med}} \stackrel{\text{def}}{=} \frac{\text{cost}}{t_{\text{pred}}^{\text{med}}} \quad p_{\text{pred}}^{\max} \stackrel{\text{def}}{=} \frac{\text{cost}}{t_{\text{pred}}^{\min}} \quad (4.4)$$

$$p_{\text{pred}}^\mu \stackrel{\text{def}}{=} \frac{\text{cost}}{t_{\text{pred}}^\mu} \left(1 + \frac{t_{\text{pred}}^{\sigma 2}}{t_{\text{pred}}^{\mu 2}} \right) \quad p_{\text{pred}}^\sigma \stackrel{\text{def}}{=} \text{cost} \times \frac{t_{\text{pred}}^\sigma}{t_{\text{pred}}^\mu} . \quad (4.5)$$

Note that the definitions of the performance prediction's mean p_{pred}^μ and standard deviation p_{pred}^σ are, respectively, second- and first-order approximations through Taylor expansions [22, Section 4.3.2].

- The *efficiency prediction* e_{pred} is obtained from the performance prediction and the processor's peak (floating-point) performance:

$$e_{\text{pred}}^s \stackrel{\text{def}}{=} \frac{p_{\text{pred}}^s}{\text{peak performance}} \quad \text{for } s \in \{\min, \text{med}, \max, \mu, \sigma\} . \quad (4.6)$$

Example 4.3: Performance and efficiency predictions

Following Examples 4.1 and 4.2, we consider that the inversion of a triangular matrix of size $n = 800$ has a minimal cost of $\frac{1}{6}n(n+1)(2n+1)$ FLOPs = 170 986 800 FLOPs and obtain the following performance prediction:

$$\begin{aligned} p_{\text{pred}}^{\min} &= 10.39 \text{ GFLOPs/s} & p_{\text{pred}}^{\max} &= 10.57 \text{ GFLOPs/s} \\ p_{\text{pred}}^{\text{med}} &= 10.54 \text{ GFLOPs/s} \\ p_{\text{pred}}^{\mu} &= 10.52 \text{ GFLOPs/s} & p_{\text{pred}}^{\sigma} &= 62.09 \text{ MFLOPs/s} . \end{aligned}$$

If we compare this prediction to the SANDY BRIDGE-EP [E5-2670's](#) theoretical single-threaded peak performance of 20.8 GFLOPs/s, we arrive at the following efficiency prediction:

$$\begin{aligned} e_{\text{pred}}^{\min} &= 49.93 \% & e_{\text{pred}}^{\text{med}} &= 50.68 \% & e_{\text{pred}}^{\max} &= 50.81 \% \\ e_{\text{pred}}^{\mu} &= 50.59 \% & e_{\text{pred}}^{\sigma} &= 0.30 \% . \end{aligned}$$

4.2 Accuracy Quantification

We evaluate the accuracy of our performance models by comparing their predictions to *measurements*. For this purpose, we time the predicted algorithm ten times (with the SAMPLER), and compute the summary statistics minimum t_{meas}^{\min} , median $t_{\text{meas}}^{\text{med}}$, maximum t_{meas}^{\max} , mean t_{meas}^{μ} , and standard deviation t_{meas}^{σ} . In contrast to our predictions, measurement statistics for other metrics, such as performance p_{meas} and efficiency e_{meas} , are obtained by first computing the metric value for each individual data-point, and then applying the corresponding statistic.

Example 4.4: Algorithm performance measurements

Measuring the runtime of the triangular matrix inversion from [Example 4.1](#) ten times yields the following results:

16.25 ms 16.27 ms 16.26 ms 16.27 ms 16.26 ms
 16.26 ms 16.28 ms 16.27 ms 16.26 ms 16.26 ms .

From these repetitions, we obtain the following summary statistics:

$$\begin{aligned} t_{\text{meas}}^{\min} &= 16.25 \text{ ms} & t_{\text{meas}}^{\text{med}} &= 16.26 \text{ ms} & t_{\text{meas}}^{\max} &= 16.27 \text{ ms} \\ t_{\text{meas}}^{\mu} &= 16.26 \text{ ms} & t_{\text{meas}}^{\sigma} &= 7.61 \mu\text{s} . \end{aligned}$$

These measurements exhibit even less fluctuations than our models predicted ([Example 4.2](#)): The runtime standard deviation t_{meas}^{σ} is only 0.05 % of the mean t_{meas}^{μ} .

We compute the *prediction error* x_{err} for any metric x as the difference between the prediction and the measurement:

$$x_{\text{err}}^s \stackrel{\text{def}}{=} x_{\text{pred}}^s - x_{\text{meas}}^s \quad \text{for } x \in \{t, p, e\}, s \in \{\min, \text{med}, \max, \mu, \sigma\} .$$

To compare the prediction error for different algorithms and problem sizes, we relate it to the predicted metric (e.g., the median measured runtime). For this purpose, we compute the *relative error (RE)* x_{RE} with respect to the measurement:

$$x_{\text{RE}}^s \stackrel{\text{def}}{=} \frac{x_{\text{err}}^s}{x_{\text{meas}}^s} \quad \text{for } x \in \{t, p, e\}, s \in \{\min, \text{med}, \max, \mu, \sigma\} .$$

Furthermore, to average errors across multiple data-points (e.g., problem sizes or setups), we use the *absolute relative error (ARE)* x_{ARE} :

$$x_{\text{ARE}}^s \stackrel{\text{def}}{=} |x_{\text{RE}}^s| \quad \text{for } x \in \{t, p, e\}, s \in \{\min, \text{med}, \max, \mu, \sigma\} .$$

Example 4.5: Prediction error

The error of our runtime predictions from [Example 4.2](#) with respect to the measurements from [Example 4.4](#) is as follows:

$$\begin{aligned} t_{\text{err}}^{\min} &= -76.99 \mu\text{s} & t_{\text{err}}^{\text{med}} &= -38.38 \mu\text{s} & t_{\text{err}}^{\max} &= 208.65 \mu\text{s} \\ t_{\text{err}}^{\mu} &= -13.15 \mu\text{s} & t_{\text{err}}^{\sigma} &= 88.27 \mu\text{s} . \end{aligned}$$

The corresponding relative error is

$$\begin{aligned} t_{\text{RE}}^{\min} &= -0.47 \% & t_{\text{RE}}^{\text{med}} &= -0.24 \% & t_{\text{RE}}^{\max} &= 1.28 \% \\ t_{\text{RE}}^{\mu} &= -0.08 \% & t_{\text{RE}}^{\sigma} &= 1160 \% . \end{aligned}$$

Note that the median, minimum, and mean runtimes are slightly under-predicted, yet well within 1 % of the measurements. However, the prediction for the maximum is somewhat less accurate; this is to be expected since it is inherently more susceptible to fluctuations. Finally, since the standard deviation was predicted as only 0.59 % of the mean but measured even lower at only 0.05 %, its relative error is gigantic; while this observation is confirmed in the following section, it does not diminish the otherwise high accuracy of our predictions.

4.3 Accuracy Case Study: Cholesky Decomposition

This section presents an in-depth evaluation of the prediction accuracy for various execution scenarios of a single algorithm on a fixed hardware and software setup: We consider the lower-triangular Cholesky decomposition

$$\begin{array}{|c|} \hline L \\ \hline \end{array} \begin{array}{|c|} \hline L^T \\ \hline \end{array} := \begin{array}{|c|} \hline A \\ \hline \end{array}$$

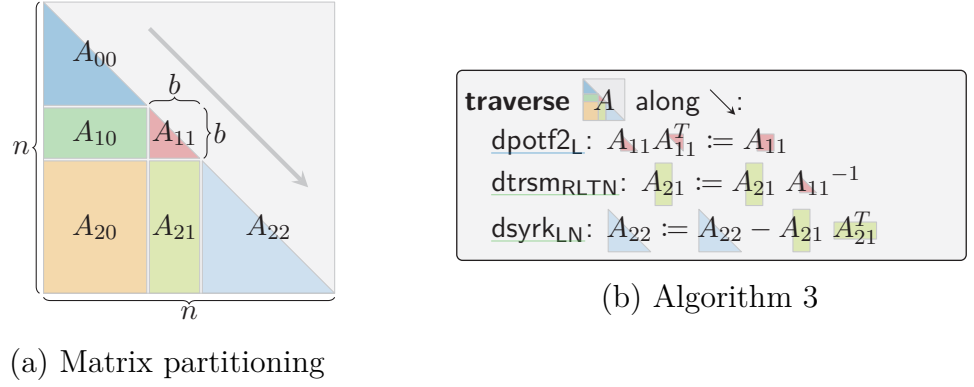


Figure 4.1: Blocked algorithm 3 for the lower-triangular Cholesky decomposition.

of a symmetric positive definite (SPD) matrix $A \in \mathbb{R}^{n \times n}$ (LAPACK: `dpotrfL`) using blocked algorithm 3 (also known as “right looking” or “greedy”). Figure 4.1 recapitulates this algorithm, which was previously detailed alongside algorithms 1 and 2 in Example 1.1 on Page 4. We focus on algorithm 3 because, as already seen in Example 1.2, it is the fastest among the three alternatives.

We perform our study on a SANDY BRIDGE-EP E5-2670 using OPENBLAS, and begin with single-threaded predictions for double-precision matrices of varying size (Section 4.3.1), then consider different block sizes (Sections 4.3.2 and 4.3.3), other data-types (Section 4.3.4), and finally multi-threaded BLAS kernels (Section 4.3.5).

4.3.1 Varying Problem Size

In our first analysis, we use only one of the SANDY BRIDGE’s 8 cores and vary the problem size between $n = 56$ and 4152 in steps of 64 while keeping the block size fixed at $b = 128$. Figure 4.2 shows the runtime and performance of predictions and measurements for this setup side-by-side. (Since the red line — at the top of the performance plots indicates the processor’s theoretical peak performance, such plots can also be interpreted as compute-bound efficiencies with 0% at the bottom and 100% at the top.) The predictions give a good idea of the algorithm behavior: While the runtime increases cubically with the

4.3 Accuracy Case Study: Cholesky Decomposition

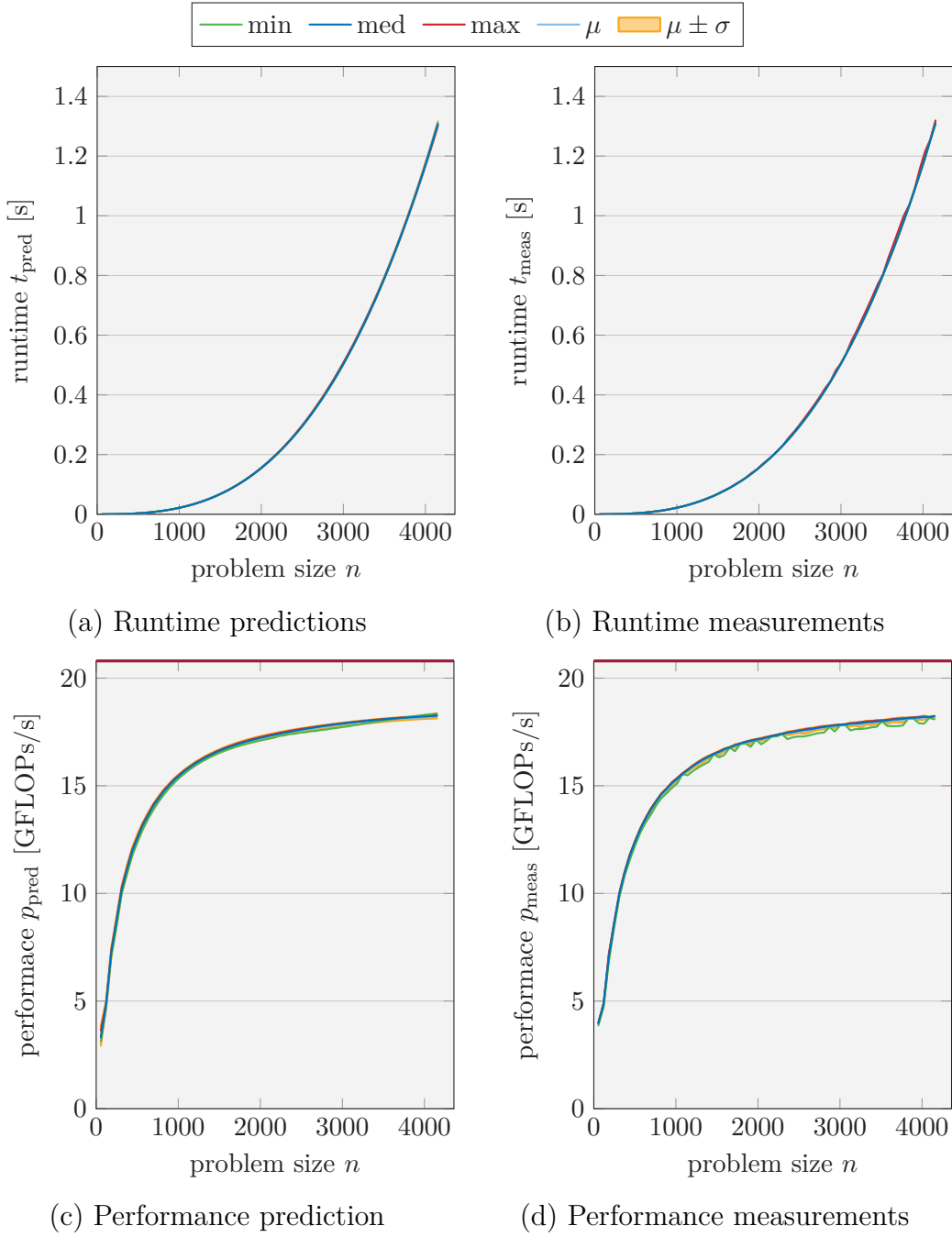


Figure 4.2: Measurements and predictions for the Cholesky decomposition.

(blocked algorithm 3, $b = 128$, SANDY BRIDGE-EP [E5-2670](#), 1 thread, OPENBLAS)

problem size n , the performance is low for small matrices and increases steadily towards 18 GFLOPs/s. At first sight, the predictions match the measurements well.

To further study the accuracy of our predictions, the top half of Figure 4.3 presents the prediction errors. As one might expect, Figure 4.3a indicates that with increasing problem size, the magnitude of the runtime prediction error increases for all summary statistics—most notably for the maximum (—). Since in contrast the performance prediction error (Figure 4.3b) is not affected by the decomposition’s cubic runtime, we instead observe the largest prediction errors for the smallest problem size $n = 56$. Furthermore, we find that the minimum performance prediction error (—) seems to alternate between two separate levels: one around 0 MFLOPs/s and one close to 200 MFLOPs/s. This behavior, which is also already somewhat visible in Figures 4.2d and 4.3a, is caused by measurement fluctuations as discussed in Section 2.1.2.3.

We gain more insights from the prediction errors when we compare it to the predicted quantities. For this purpose, the bottom half of Figure 4.3 presents the relative runtime and performance prediction errors. These relative errors for these metrics are almost identical up to a change in the sign—since the runtime is generally slightly underestimated, the performance is somewhat overestimated. Focusing on the runtime in Figure 4.3c, we notice that the average standard deviation ARE is 194.70% (—), which, as in Example 4.5, exceeds the error of the other prediction statistics by far. Furthermore, the previously addressed measurement fluctuations are also clearly visible in the maximum (—) as variations with a magnitude of 1.5%. The minimum (—), median (—), and mean (—) AREs on the other hand quickly fall below 2% for matrices larger than $n = 200$ and further below below 1% beyond $n \approx 1000$; across all chosen problem sizes, the average AREs for the minimum, median and mean runtime are, respectively, 0.78%, 0.91%, and 0.90%.

Among the eight metrics presented in Figures 4.2 and 4.3, we gained the most insight from 1) the performance prediction (Figure 4.2c), which gives a good idea of both the algorithm’s performance and efficiency, and 2) the relative runtime prediction error (Figure 4.3c), which provides not only an

4.3 Accuracy Case Study: Cholesky Decomposition

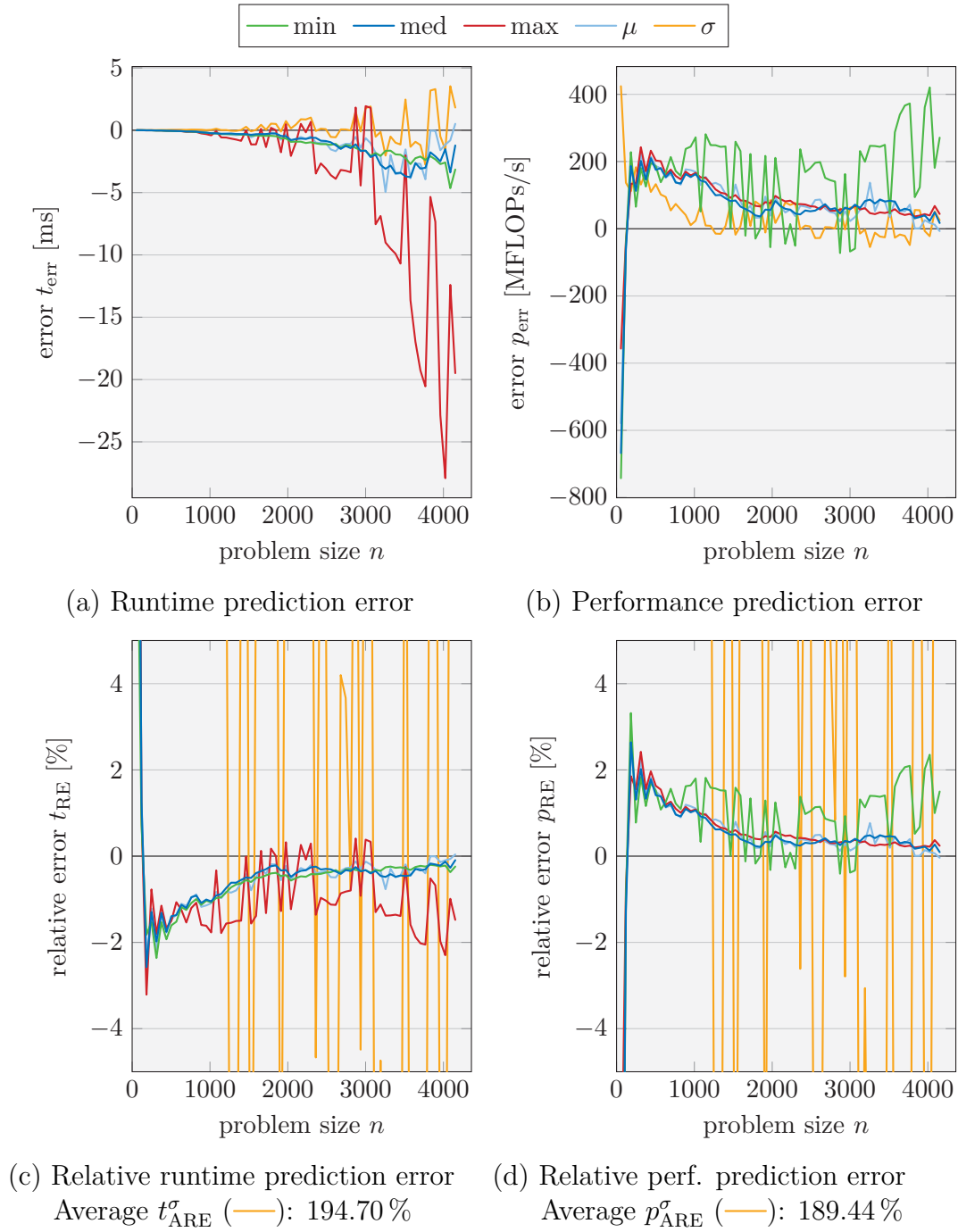


Figure 4.3: Prediction accuracy for the Cholesky decomposition.

(blocked algorithm 3, $b = 128$, SANDY BRIDGE-EP E5-2670, 1 thread, OPENBLAS)

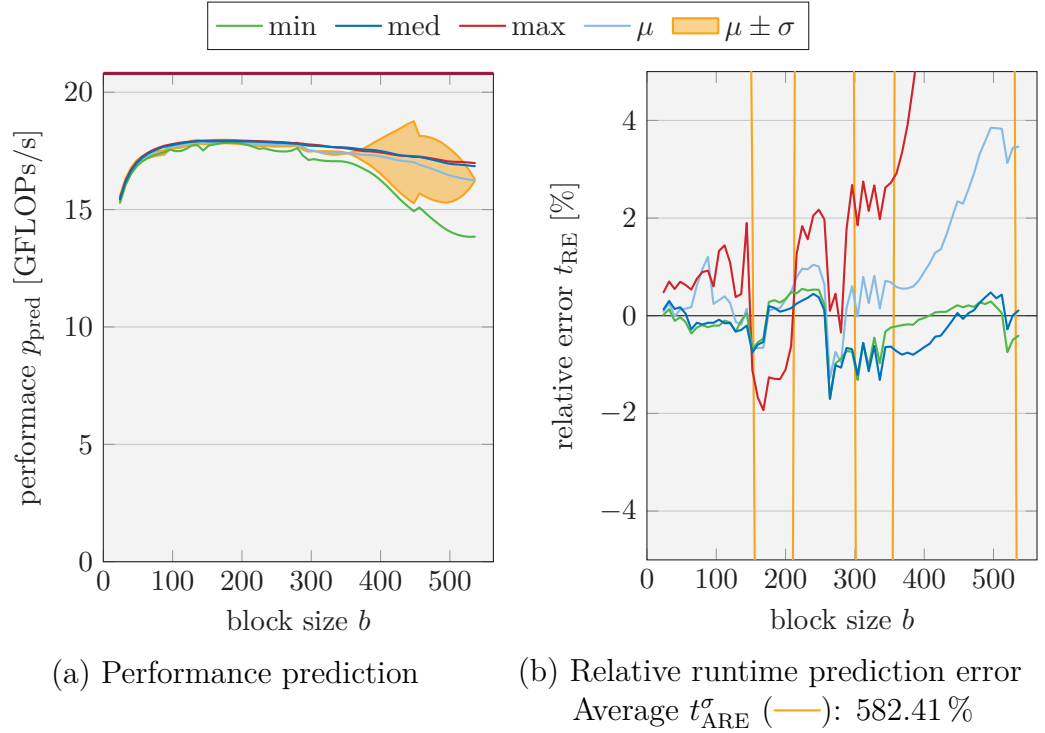


Figure 4.4: Predictions and prediction accuracy for the Cholesky decomposition with varying block size.

(blocked algorithm 3, $n = 3000$, SANDY BRIDGE-EP [E5-2670](#), 1 thread, OPENBLAS)

accuracy measure independent of the operation, the algorithm, and the actual performance, but also indicates whether the runtime is under- or overestimated. Hence, we use these two types of plots in our following analyses.

4.3.2 Varying Block Size

In our next analysis, we fix the problem size to $n = 3000$ and vary the block size between $b = 24$ and 536 in steps of 8. [Figure 4.4](#) presents the performance prediction and the relative runtime prediction error for this scenario using single-threaded OPENBLAS on the SANDY BRIDGE.

The performance prediction ([Figure 4.4a](#)) exhibits the typical trade-off for any blocked algorithm: While for both small and large block sizes the algorithm attains rather poor performance, in between it reaches up to 17.91 GFLOPs/s,

which corresponds to an efficiency of 85.10 %. The cause for this trade-off and the selection of block sizes are addressed in detail in [Section 4.6](#).

Compared to our previous performance predictions ([Figure 4.2c](#)), [Figure 4.4a](#) exhibits a far wider spread of the summary statistics for large block sizes. In particular, the predicted minimum performance (—) drops drastically, which immediately causes the mean performance (—) to decrease and an enormous increase in the predicted standard deviation (—).

The relative runtime prediction error ([Figure 4.4b](#)) indicates that the predicted performance fluctuations are not present in the performance measurements: The maximum and mean relative errors (— and —) increase drastically for large problem size, suggesting that the model generation was influenced by large outlier measurements. (A repetition of the generation process would likely encounter different outliers and distort these metrics statistics for other problem sizes.) The minimum (—) and median (—), on the other hand, are with few exceptions predicted within 1 %; their average prediction AREs are 0.36 % (minimum —) and 0.42 % (median —).

4.3.3 Varying Problem Size and Block Size

If we vary both the problem size n and the block size b , we can visualize the runtime prediction ARE as a set of heat-maps as shown in [Figure 4.5](#). Note that these plots are based on a total of 39 690 measurements of the algorithm’s runtime (65 problem sizes, \approx 65 block sizes, 10 repetitions) that took over 4 hours. The performance models for the kernels needed for the predictions (`dpotf2L`, `dtrsmRLTN`, and `dsyrkLN`), on the other hand, were generated in just under 10 minutes, produced our predictions in under 20 s.

The standard deviation ARE is once again too large to fit the chosen scale and is hence not shown. Furthermore, as already seen in [Figure 4.4](#), the maximum prediction becomes rather inaccurate for large n and b , which also has a negative impact on the mean prediction. On the other hand, both the minimum and median predictions are overall quite accurate with an average ARE of only 0.45 %.

4 Model-Based Predictions for Blocked Algorithms

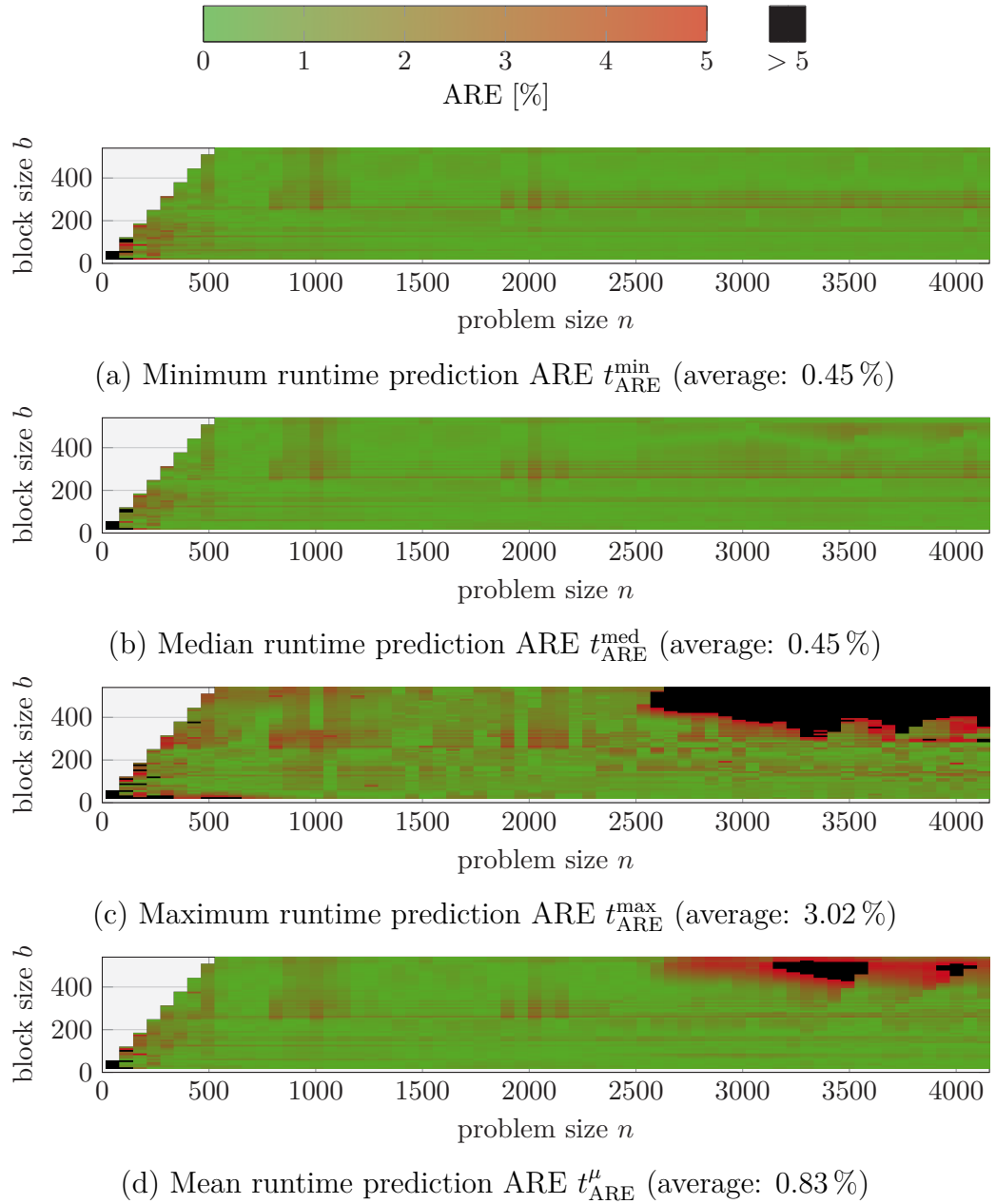


Figure 4.5: Prediction accuracy for the Cholesky decomposition.

Average t_{ARE}^{σ} : 346.87 %

(blocked algorithm 3, SANDY BRIDGE-EP [E5-2670](#), 1 thread, OPENBLAS)

4.3 Accuracy Case Study: Cholesky Decomposition

data-type	kernels		
single-precision real	<code>spotf2_L</code>	<code>strsm_{RLTN}</code>	<code>ssyrk_{LN}</code>
double-precision real	<code>dpotf2_L</code>	<code>dtrsm_{RLTN}</code>	<code>dsyrk_{LN}</code>
single-precision complex	<code>cpotf2_L</code>	<code>ctrsm_{RLTN}</code>	<code>cherk_{LN}</code>
double-precision complex	<code>zpotf2_L</code>	<code>ztrsm_{RLTN}</code>	<code>zherk_{LN}</code>

Table 4.2: Kernels in the Cholesky decomposition for different data-types.

Since in the following we compare multiple alternative algorithms and hardware/software setups, we limit our focus to a single statistic. While in the previous analysis the runtime minimum or median were predicted with equivalent accuracy, in practice the expected performance is better represented by the median runtime.¹ Hence, from now on we use the *relative median runtime prediction error* $t_{\text{RE}}^{\text{med}}$ as our *prediction accuracy measure*.

4.3.4 Other Data-Types

So far, we have considered the Cholesky decomposition of real double-precision matrices; however, the same algorithm is also applicable to other data-types. For the four de-facto standard numerical data-types (real and complex² floating-point numbers in single- and double-precision) Table 4.2 summarizes the algorithm’s BLAS and LAPACK kernels, and Figure 4.6 presents our model’s performance predictions and their accuracy. (For each data-type, we generated a separate set of performance models.)

In the performance predictions (Figure 4.6a), we observe that the real double-precision version (—) is most efficient (with respect to its theoretical peak performance); this was to be expected because OPENBLAS is most optimized for this data-type. In contrast, it is somewhat surprising that, while single-precision complex (—) is noticeably more performant than single-precision real (—), double-precision complex (—) does not exceed an efficiency of 50 %.

¹ In scenarios other than our considered single-node computations different measures might be preferable; e.g., the 90th percentile runtime.

² For the complex cases, the Cholesky decomposition is of the form $LL^H := A$, where A must be Hermitian positive definite (HPD).

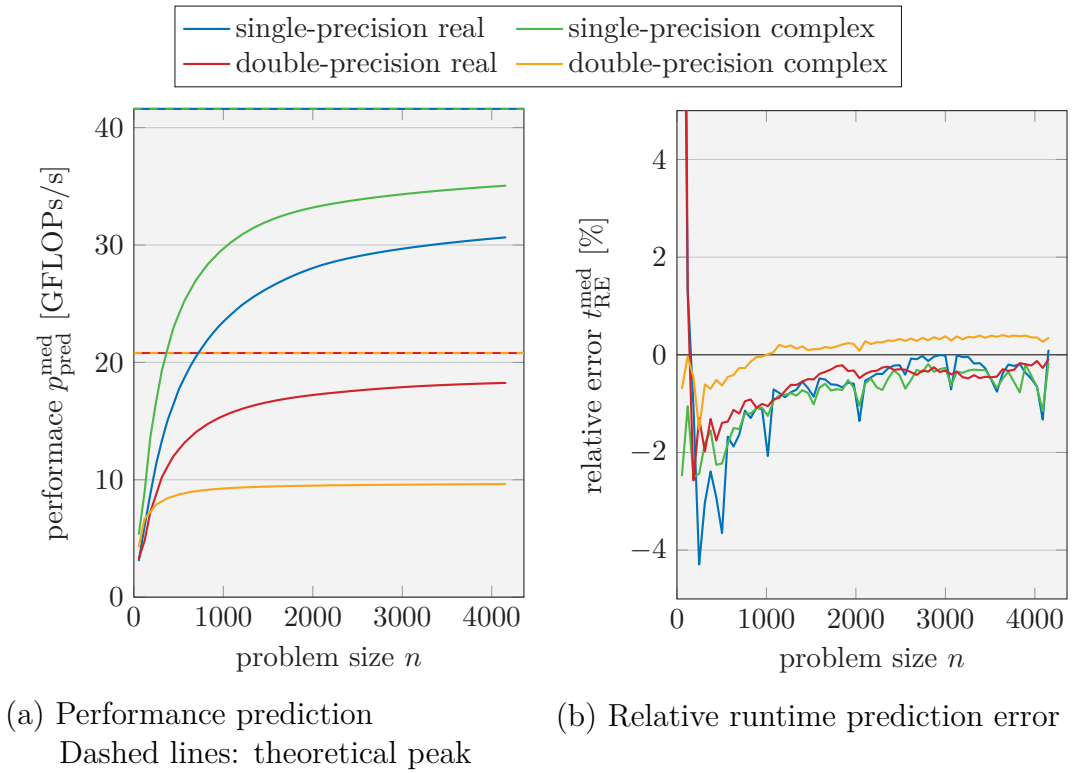


Figure 4.6: Predictions and prediction accuracy for the Cholesky decomposition with different data-types.

(algorithm 3, $b = 128$, SANDY BRIDGE-EP E5-2670, 1 thread, OPENBLAS, median)

Although the algorithm’s performance for the four data-types differs significantly, [Figure 4.6a](#) reveals that our models predict the runtime for all of them equally well. Moreover, for the in comparison inefficient double-precision complex variant (—), the prediction is already notably accurate small problem sizes below $n = 1000$.

With equally accurate predictions demonstrated for four data-types, we will in the following focus on real operations in double-precision.

4.3.5 Multi-Threaded BLAS

Finally, we consider how multi-threading (through OPENBLAS) impacts the algorithm’s performance and our predictions’ accuracy. For this purpose, [Fig-](#)

4.3 Accuracy Case Study: Cholesky Decomposition

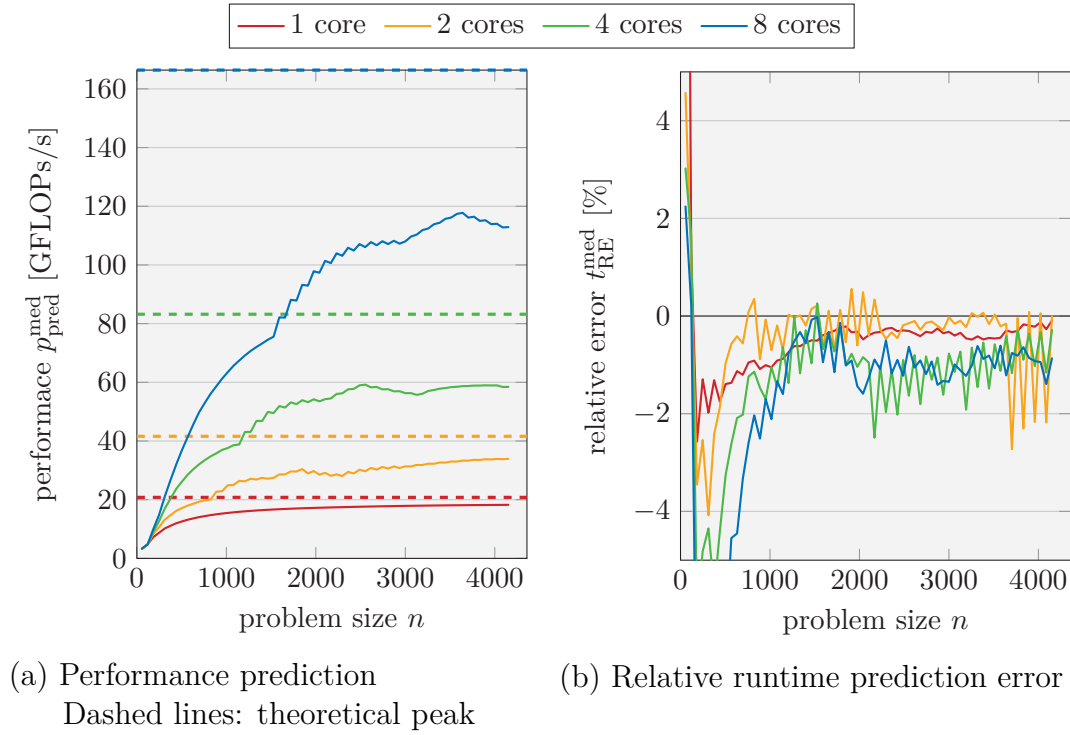


Figure 4.7: Predictions and prediction accuracy for the Cholesky decomposition with multi-threaded OPENBLAS.

(algorithm 3, $b = 128$, SANDY BRIDGE-EP [E5-2670](#), median)

Figure 4.7 presents the predicted performance of the Cholesky decomposition and the prediction accuracy with 1, 2, 4, and 8 threads on the 8-core SANDY BRIDGE. (For each of these four levels of parallelism, a separate set of performance models was generated.)

The predictions show that, while the performance grows with the number of threads, the efficiency decreases from 87.74% with one thread to a maximum of 70.78% with eight threads. Furthermore, the performance curves become less smooth with increased parallelism.

Considering our prediction's accuracy, we notice that for small problem sizes below $n = 500$, the prediction ARE increases significantly when more threads are added. Beyond this point however, the prediction for 1 (—) and 2 threads (—) are both highly accurate with an average ARE of 0.46%;

the predictions for 4 (—) and 8 threads (—) are slightly less accurate and the AREs fluctuate around 1%. Note that the large fluctuations within the ARE for the multi-threaded algorithms are caused by the combination of the block size $b = 128$ and the chosen problem sizes in steps of 64. While with 8 threads (—) these fluctuations are represented by our predictions to some degree, with 2 (—) and 4 threads (—), they are most striking for large problem sizes, where our models do not predict such fluctuations.

4.3.6 Summary

We studied the blocked Cholesky decomposition algorithm 3 on a SANDY BRIDGE-EP E5-2670 using OPENBLAS with varying problem and block sizes, data-types, and kernel parallelism. We analyzed this algorithm’s measured and predicted runtime and performance to evaluate the accuracy of our predictions, and selected the relative median runtime prediction error $t_{\text{RE}}^{\text{med}}$ as our primary accuracy measure.

4.4 Accuracy Study:

Blocked LAPACK Algorithms

We now extend our analysis from the previous case study to a larger group of algorithms and a wider range of hardware and software setups. We consider six of LAPACK’s blocked algorithms:

dlauum_ Lower-triangular matrix multiplication with its transpose:

$$\boxed{A} := \begin{array}{c} \diagup L^T \\ \diagdown L \end{array}$$

with $\begin{array}{c} \diagup L \\ \diagdown \end{array} \in \mathbb{R}^{n \times n}$ and $\boxed{A} \in \mathbb{R}^{n \times n}$ symmetric. The algorithm, outlined in [Figure 4.8a](#), overwrites $\begin{array}{c} \diagup L \\ \diagdown \end{array}$ with \boxed{A} in lower-triangular storage.

dlauum arises as part of the inversion of symmetric positive definite (SPD) matrices ($\boxed{A} := \boxed{A}^{-1}$).

traverse A along \searrow :
 dtrmm_{LLTN}: $A_{10} := A_{11}^T A_{10}$
 dlauu2_L: $A_{11} := A_{11} A_{11}^T$
 dgemm_{TN}: $A_{10} := A_{10} + A_{21}^T A_{20}$
 dsyrk_{LT}: $A_{11} := A_{11} + A_{21}^T A_{21}$

(a) $\text{dlauum}_L: A := A^T A$

traverse A and L along \searrow :
 dsygs2_{1L}: $A_{11} := L_{11}^{-1} A_{11} L_{11}^{-T}$
 dtrsm_{RLTN}: $A_{21} := A_{21} L_{11}^{-T}$
 dsymm_{RL}: $A_{21} := A_{21} - \frac{1}{2} L_{21} A_{11}$
 dsyr2k_{LN}: $A_{22} := A_{22} - A_{21} L_{21}^T - L_{21} A_{21}^T$
 dsymm_{RL}: $A_{21} := A_{21} - \frac{1}{2} L_{21} A_{11}$
 dtrsm_{LLNN}: $A_{21} := L_{22}^{-1} A_{21}$

(b) $\text{dsygst}_{1L}: A := L^{-1} A L^{-T}$

traverse A along \nwarrow :
 dtrmm_{LLNN}: $A_{21} := A_{22} A_{21}$
 dtrsm_{RLNN}: $A_{21} := -A_{21} A_{11}^{-1}$
 dtrti2_{LN}: $A_{11} := A_{11}^{-1}$

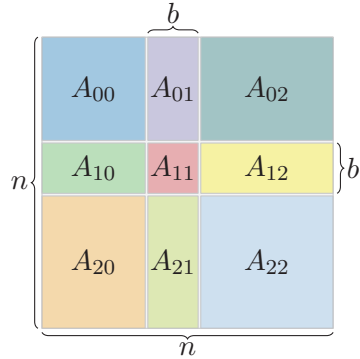
(c) $\text{dtrtri}_{LN}: A := A^{-1}$

traverse A along \searrow :
 dsyrk_{LN}: $A_{11} := A_{11} - A_{10} A_{10}^T$
 dpotf2_L: $A_{11} A_{11}^T := A_{11}$
 dgemm_{NT}: $A_{21} := A_{21} - A_{20} A_{10}^T$
 dtrsm_{RLTN}: $A_{21} := A_{21} A_{11}^{-1}$

(d) $\text{dpotrf}_L: A A^T := A$

$P := I$
traverse A along \searrow :
 dgetf2: $A_{11} A_{11} := A_{11}$, update P
 dlaswp: apply P to A_{10}
 dlaswp: apply P to A_{20}
 dlaswp: apply P to A_{12}
 dlaswp: apply P to A_{22}
 dtrsm_{LLNU}: $A_{12} := A_{11} A_{12}$
 dgemm_{NN}: $A_{22} := A_{22} - A_{21} A_{12}$

(e) $\text{dgetrf}: P L U := A$



(f) Matrix partitioning

Figure 4.8: LAPACK's blocked algorithms for dlauum_L , dsygst_{1L} , dtrtri_{LN} , dpotrf_L , and dgetrf .

dsygst_{1L} Two-sided symmetric lower-triangular linear system solve³:

$$\boxed{A} := \begin{array}{c} \diagdown \\ L^{-1} \end{array} \boxed{A} \begin{array}{c} \diagup \\ L^{-T} \end{array}$$

with $\begin{array}{c} \diagdown \\ L \end{array} \in \mathbb{R}^{n \times n}$ and $\boxed{A} \in \mathbb{R}^{n \times n}$ symmetric in lower-triangular storage. The algorithm is outlined in [Figure 4.8b](#).

dsygst is used to reduce generalized SPD eigenvalue problems (e.g., $\boxed{A}x = \lambda \boxed{B}x$) to the standard form ($\boxed{A}x = \lambda x$).

dtrtri_{LN} Inversion of a lower-triangular matrix:

$$\begin{array}{c} \diagdown \\ A \end{array} := \begin{array}{c} \diagdown \\ A^{-1} \end{array}$$

with $\begin{array}{c} \diagdown \\ A \end{array} \in \mathbb{R}^{n \times n}$. The algorithm is outlined in [Figure 4.8c](#).

dtrtri is a building block for the inversion of general and SPD matrices, which are used when, instead of the solution of a linear system, the actual numeric entries in the inverse matrix are required.

dpotrf_L Lower-triangular Cholesky decomposition:

$$\begin{array}{c} \diagdown \\ L \end{array} \begin{array}{c} \diagup \\ L^T \end{array} := \boxed{A}$$

with $\boxed{A} \in \mathbb{R}^{n \times n}$ SPD in lower-triangular storage and $\begin{array}{c} \diagdown \\ L \end{array} \in \mathbb{R}^{n \times n}$. The algorithm, outlined in [Figure 4.8d](#), overwrites \boxed{A} with $\begin{array}{c} \diagdown \\ L \end{array}$.

dpotrf is central to many operations on SPD matrices, for instance: inversion, solution of linear systems ($x := \boxed{A}^{-1}x$), and reduction of generalized eigenvalue problems to standard form.

dgetrf LU decomposition with partial pivoting:

$$\boxed{P} \begin{array}{c} \diagdown \\ L \end{array} \begin{array}{c} \diagup \\ U \end{array} := \boxed{A}$$

³ **dsygst**'s first flag argument indicates whether (1) L^{-1} or (2) L is applied.

4.4 Accuracy Study: Blocked LAPACK Algorithms

with $A \in \mathbb{R}^{m \times n}$, $L \in \mathbb{R}^{m \times \min(m,n)}$ unit triangular, and $U \in \mathbb{R}^{\min(m,n) \times n}$. The algorithm, outlined in Figure 4.8e, overwrites A with L and U , and returns P as a permutation vector.

`dgetrf` is used to solve linear systems and invert general matrices.

`dgeqrf` QR decomposition:

$$Q \ R := A$$

with $A \in \mathbb{R}^{m \times n}$, $Q \in \mathbb{R}^{m \times \min(m,n)}$, and $R \in \mathbb{R}^{\min(m,n) \times n}$. The algorithm, outlined in Figure 4.9, overwrites A 's upper-triangular (or -trapezoidal) part with R , and represents Q as a product of elementary reflectors stored in A 's strictly lower-triangular (or -trapezoidal) part and a vector of scalar factors τ .

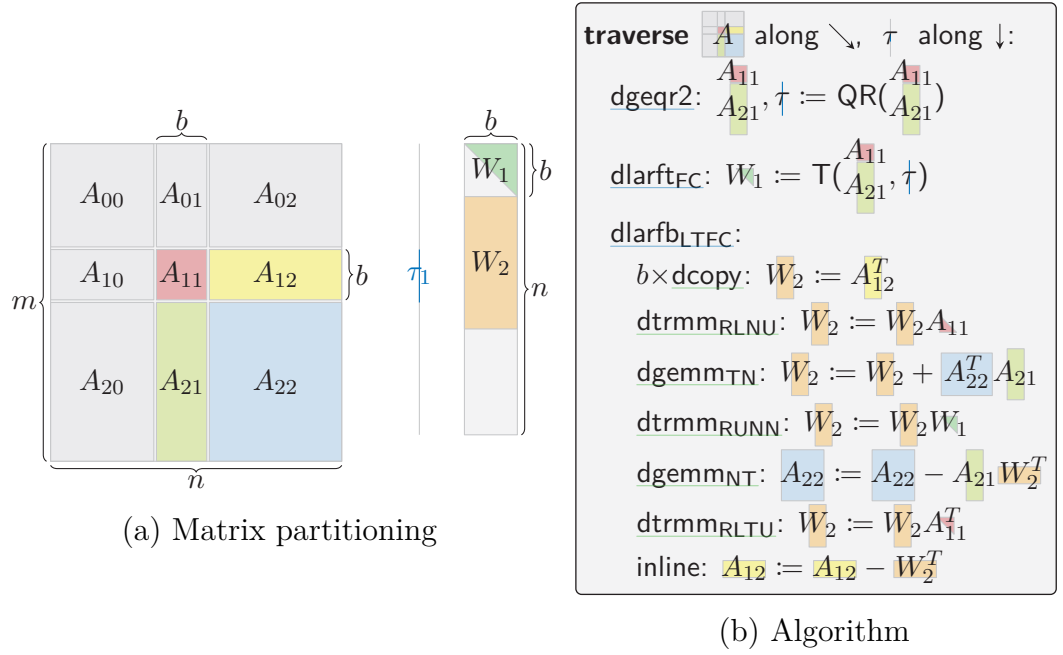
`dgeqrf` is used in several eigensolvers ($Q \ \Lambda \ Q^{-1} := A$), the singular-value decomposition ($U \ \Sigma \ V^T := A$), and least-squares solvers ($X := \arg \min \|B - A X\|$).

For `dgetrf` and `dgeqrf`, we consider the square case with $m = n$.

We study a total of six hardware and software setups: An 8-core SANDY BRIDGE-EP E5-2670 and a 12-core HASWELL-EP E5-2680 v3 with OPEN-BLAS, BLIS, and MKL. We consider both the single-threaded case and the scenario where all processor cores are used by the BLAS implementation (with the exception of BLIS, which did not offer a user-friendly threading model at the time of writing). For all of these operations, we both predict and measure the runtime for problem sizes between $n = 56$ and 4152 in steps of 64.

4.4.1 Single-Threaded BLAS

We begin with a study of the single-threaded prediction accuracy with LAPACK's default block size ($b = 64$, except for `dgeqrf` with $b = 32$). While these are generally sub-optimal configurations and often even sub-optimal algorithms for the performed operations, this configuration is unfortunately still encountered frequently in application codes that use the reference LAPACK

Figure 4.9: LAPACK's blocked algorithm for dgeqrf.

	SANDY BRIDGE-EP E5-2670			HASWELL-EP E5-2680 v3			average
	OPENBLAS	BLIS	MKL	OPENBLAS	BLIS	MKL	
	(—)	(—)	(—)	(—)	(—)	(—)	
<u>dlauum</u> _L	1.23 %	2.70 %	1.40 %	0.92 %	0.75 %	2.19 %	1.53 %
<u>dsygst</u> _{1L}	1.05 %	2.05 %	3.31 %	3.58 %	2.44 %	3.35 %	2.63 %
<u>dtrtri</u> _{LN}	0.71 %	2.02 %	1.31 %	2.09 %	1.67 %	1.69 %	1.58 %
<u>dpotrf</u> _L	1.44 %	1.03 %	1.44 %	2.05 %	1.52 %	2.44 %	1.65 %
<u>dgetrf</u>	1.01 %	0.96 %	0.80 %	1.13 %	1.63 %	1.67 %	1.20 %
<u>dgeqrf</u>	1.85 %	2.05 %	3.55 %	3.64 %	3.93 %	2.22 %	2.87 %
average	1.22 %	1.80 %	1.97 %	2.24 %	1.99 %	2.26 %	1.91 %

Table 4.3: Single-threaded runtime prediction ARE $t_{\text{ARE}}^{\text{med}}$ for blocked LAPACK algorithms averaged across problem sizes.(n = 56, ..., 4152 in steps of 64; b = 64 except dgeqrf: b = 32)

implementation. As such, it forms a quite canonical reference for the evaluation of our predictions.

Figure 4.10 presents the relative runtime prediction error $t_{\text{RE}}^{\text{med}}$ for this scenario.

4.4 Accuracy Study: Blocked LAPACK Algorithms

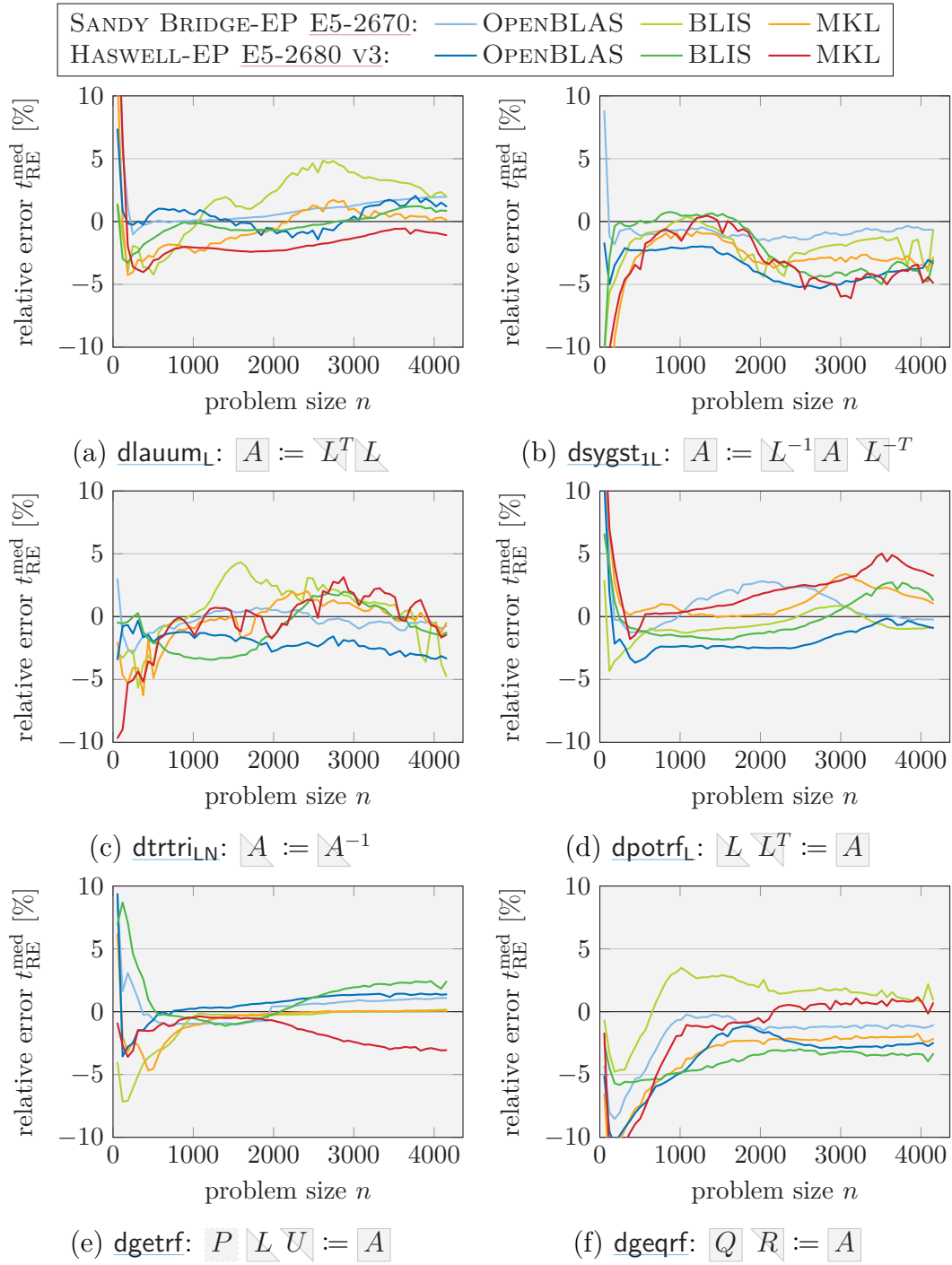


Figure 4.10: Single-threaded prediction accuracy for LAPACK algorithms.

($b = 64$, except dgeqrf : $b = 32$)

For all algorithms and setups, our predictions are mostly within 5 % of the measured runtime, and in many situations considerably closer. The runtime prediction ARE averaged across all problem sizes for each routine and setup is summarized in [Table 4.3](#): It ranges from 0.71 % to 3.93 %, and its average and median are, respectively, 1.91 % and 1.69 %. Overall, the predictions are slightly more accurate on the SANDY BRIDGE-EP [E5-2670](#) (average $t_{\text{ARE}}^{\text{med}} = 1.66$ %) with the lowest average $t_{\text{ARE}}^{\text{med}} = 1.22$ % for OPENBLAS ([—](#)); on the HASWELL-EP [E5-2680 v3](#) (average $t_{\text{ARE}}^{\text{med}} = 2.16$ %), the predictions are least accurate for MKL ([—](#)) with an average of $t_{\text{ARE}}^{\text{med}} = 2.26$ %.

Most routines are predicted equally well (with an average $t_{\text{ARE}}^{\text{med}}$ around 1.5 %) with two exceptions: [dsygst1l](#) (average $t_{\text{ARE}}^{\text{med}} = 2.63$ %) and [dgeqrf](#) (average $t_{\text{ARE}}^{\text{med}} = 2.87$ %).

- For the two-sided linear system solver [dsygst1l](#), [Figure 4.10b](#) reveals that for most setups, the predictions consistently underestimate the algorithm runtime for large problem sizes n .

A quick calculation shows that this effect is related to the size of the last-level cache (L3): On the HASWELL, the problem emerges beyond $n \approx 2000$ at which point the two operands [A](#) (symmetric in lower-triangular storage) and [L](#) take up $2 \times \frac{2000^2}{2}$ doubles ≈ 30.52 MiB—slightly more than the L3 cache of 30 MiB. On the SANDY BRIDGE with 20 MiB of L3 cache, the effect is accordingly already visible beyond $n \approx 1600$.

The cause for the underestimation of large problems is as follows: Our models are based on repeated kernel measurements, which operate on cached (“warm”) data as long as all of the kernel’s arguments fit in the cache. However, each traversal step of [dsygst1l](#) ([Figure 4.8b](#)) uses two separate kernels (namely [dsyr2k_{LN}](#) and [dtrsm_{LLNN}](#)) that operate on the trailing parts of [A](#) and [L](#)—since these do not fit in the cache simultaneously, they are mutually evicted by these kernels, and hence have to be loaded from main memory repeatedly (“cold” data). To summarize, our models estimate fast operations on cached data, while in the algorithm the operations are slower due to cache misses.

A more detailed study of caching effects within blocked algorithms and attempts to account for them are presented in [Chapter 5](#).

Note that only `dsygst1L` is affected by caching effects on this scale because all other routines involve only one dense operand.

- For the QR decomposition `dgeqrf`, [Figure 4.10f](#) reports that the runtime for almost all setups is consistently underestimated—especially for small problems.

The cause is the transposed matrix copy and addition (see [Figure 4.9](#)), which account for about 4 % of the runtime for small problems ($n \approx 250$) and 1 % for large problems ($n \approx 4000$): The copy, performed by a sequence of $b = 32$ `dcopy`s, is underestimated by $2\times$ to $7\times$ because our models do not account for caching effects; the addition, which inlined as two nested loops, is not accounted for at all.

4.4.2 Multi-Threaded BLAS

We study the multi-threaded prediction accuracy for the same six LAPACK algorithms using all available cores of the processors, i.e., 8 threads on the SANDY BRIDGE-EP [E5-2670](#) and 12 threads on the HASWELL-EP [E5-2680 v3](#). In contrast to the single-threaded predictions, we use a block size of $b = 128$ for all algorithms—while this configuration is certainly not optimal for all algorithms and problem sizes, it generally yields better performance than LAPACK’s default values.

[Figure 4.11](#) presents the relative runtime prediction errors $t_{\text{RE}}^{\text{med}}$ for this scenario, and [Table 4.4](#) summarizes their averaged AREs $t_{\text{ARE}}^{\text{med}}$. Compared to the single-threaded case, the prediction errors are across the board around $2.5\times$ larger with a total average of $t_{\text{ARE}}^{\text{med}} = 4.85\%$. The predictions are roughly equally accurate across the two architectures and the two BLAS implementations.

Considering [Figure 4.11](#), we note fluctuation patterns in the prediction errors by up to 10 %, most notably for `dsygst1L` and `dtrtri1L` using MKL on the

4 Model-Based Predictions for Blocked Algorithms

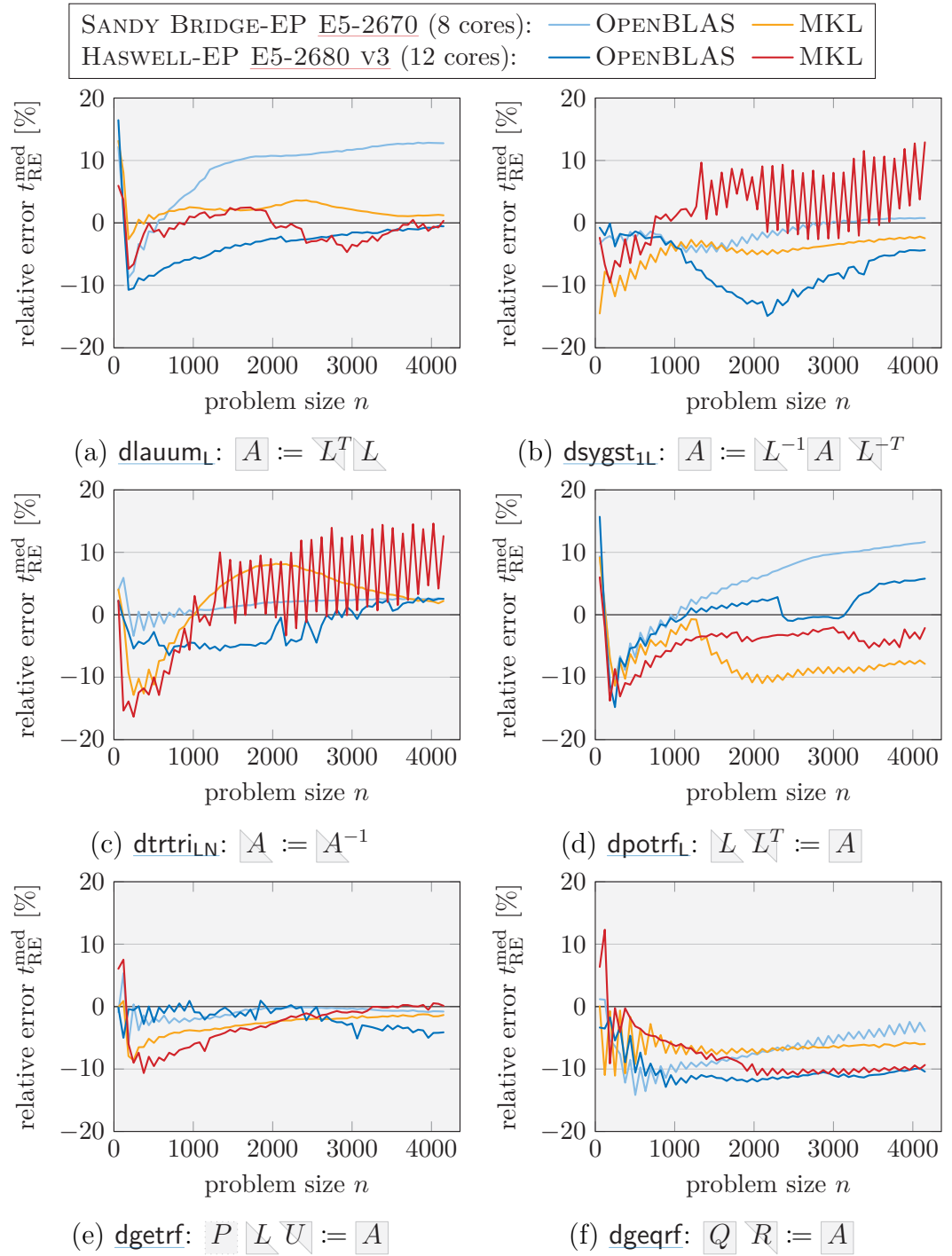


Figure 4.11: Multi-threaded prediction accuracy for LAPACK algorithms.

($b = 128$)

4.4 Accuracy Study: Blocked LAPACK Algorithms

	SANDY BRIDGE-EP E5-2670 OPENBLAS (—)	MKL (—)	HASWELL-EP E5-2680 v3 OPENBLAS (—)	MKL (—)	average
<u>dlauum_L</u>	9.42 %	2.29 %	3.73 %	1.93 %	4.34 %
<u>dsygst_{1L}</u>	1.83 %	4.55 %	7.17 %	5.03 %	4.65 %
<u>dtrtri_{LN}</u>	1.91 %	5.28 %	3.18 %	7.05 %	4.35 %
<u>dpotrf_L</u>	6.89 %	7.46 %	3.00 %	4.65 %	5.50 %
<u>dgetrf</u>	1.07 %	2.81 %	1.87 %	3.28 %	2.26 %
<u>dgeqrf</u>	6.89 %	6.37 %	10.32 %	8.42 %	8.00 %
average	4.67 %	4.79 %	4.88 %	5.06 %	4.85 %

Table 4.4: Multi-threaded runtime prediction ARE $t_{\text{ARE}}^{\text{med}}$ for blocked LAPACK algorithms averaged across problem sizes.

($n = 56, \dots, 4152$ in steps of 64, $b = 128$, SANDY BRIDGE: 8 cores, HASWELL: 12 cores)

HASWELL (—). As observed in Section 4.3.5, these fluctuations are an artefact of the block size $b = 128$ interacting with the considered problem sizes in steps of 64: Between consecutive problem sizes, the remaining matrix portions in the last step of the matrix traversal alternate between widths 56 and 120.

As in the single-threaded case, the QR decomposition’s runtime is underestimated by on average 8.00 %, due to the dcopys and the inlined matrix addition. Since especially the latter cannot make any use of the multi-threaded parallelism, their impact increases significantly with the number of available cores.

Furthermore, several individual algorithms and setups are consistently under- or overestimated: e.g., OPENBLAS on the SANDY BRIDGE-EP E5-2670 (—) for dlauum_L and dpotrf_L. These problems arise from the multi-threaded implementations of dgemm, whose irregular performance is not well represented in our models: Since BLAS implementations distribute computations among threads along a certain dimension of the operation, for small dimension (such as the block size), only a subset of the available threads is used. When the small dimension is increased, more threads are activated and the performance increases suddenly.

4.4.3 Summary

This section has shown that across experiments on two processor architectures, three BLAS implementations, and six blocked LAPACK algorithms, our models yield accurate predictions that are on average within 1.91 % (single-threaded) and 4.85 % (multi-threaded) of reference measurements. Encouraged by these accuracy results, the following sections use performance predictions to target our main goals of algorithm selection and block-size optimization.

4.5 Algorithm Selection

This section uses model-based predictions to determine which of several alternative blocked algorithms for the same operation is the fastest. To confirm the correctness of our predictions' selections on a HASWELL-EP [E5-2680 v3](#) using OPENBLAS, we compare them to the optimal algorithms identified by time-consuming empirical measurements.

[Section 4.5.1](#) revisits the Cholesky decomposition with only three alternative blocked algorithms, [Section 4.5.2](#) considers the inversion of a triangular matrix with eight alternatives, and [Section 4.5.3](#) addresses the solution of the triangular Sylvester equation with a total of 64 algorithms.

4.5.1 Cholesky Decomposition

The *three blocked algorithms* for the lower-triangular Cholesky decomposition

$$\begin{array}{|c|} \hline L \\ \hline \end{array} \begin{array}{|c|} \hline L^T \\ \hline \end{array} := \begin{array}{|c|} \hline A \\ \hline \end{array}$$

of a symmetric positive definite matrix $\begin{array}{|c|} \hline A \\ \hline \end{array} \in \mathbb{R}^{n \times n}$ were introduced in [Example 1.1](#) on [Page 4](#), and [Section 4.3](#) studied algorithm 3 in detail.

[Figure 4.12](#) presents the performance predictions and measurements for the three algorithms with problem sizes $n = 56, \dots, 4152$ in steps of 64. For both the single- and multi-threaded setup, the predictions accurately indicate that algorithm 3 ([—](#)) is the fastest among the three alternatives. The differences

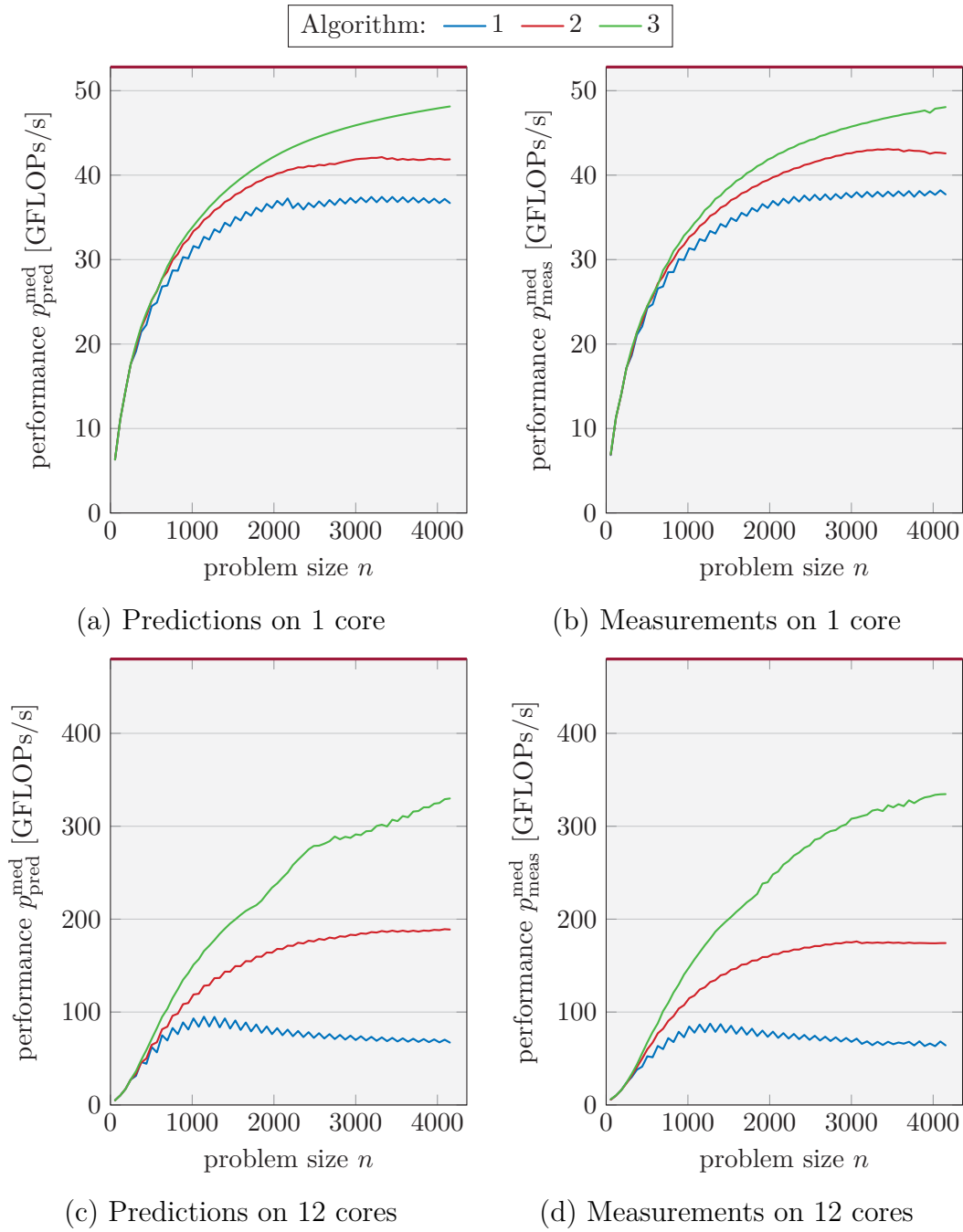


Figure 4.12: Performance measurements and predictions for the blocked Cholesky decomposition algorithms in lower-triangular storage.

($b = 128$, HASWELL-EP [E5-2680 v3](#), OPENBLAS)

in performance among the three algorithms is enormous: On 1 and 12 cores, algorithm 3 (—) is faster than algorithm 1 (—) by, respectively, 31.17% and 391.16%.

Although our study reveals that algorithm 3 (—) is the fastest among the three alternatives, LAPACK uses the suboptimal algorithm 2 (—) in its `dpotrfL`.

Note that while the reference performance measurements (Figures 4.12a and 4.12c) together took around 1 minute, our prediction identified the fastest algorithm in just over 0.5s—over 100× faster. Since for these predictions we represented and evaluated our models in PYTHON, we expect that using another storage format and evaluation system (e.g., in C/C++) would further increased the prediction speed by one or two orders of magnitude.

4.5.2 Triangular Inversion

Figure 4.13 presents the *eight blocked algorithms* for the inversion of a lower-triangular matrix

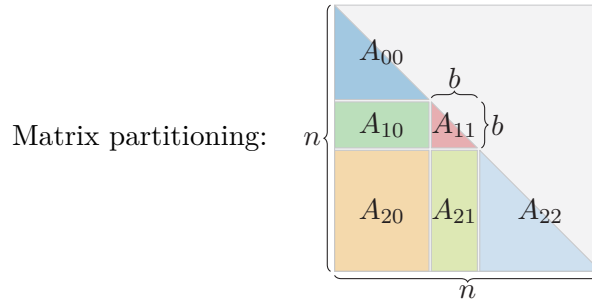
$$\triangleleft A := \triangleleft A^{-1}$$

with $A \in \mathbb{R}^{n \times n}$ non-singular. Note that algorithms 5 through 8 are the mirrors of algorithms 1 through 4 with the opposite traversal direction—↖ instead of ↘. Furthermore, algorithms 4 and 8 not only perform around 3× more FLOPs than required, but are also numerically unstable.⁴ Note that LAPACK’s `dtrtriLN` implements algorithm 5 with a default block size of $b = 64$.

Figure 4.14 presents the performance predictions and measurements for the eight algorithms for problem sizes between $n = 56$ and 4152 in steps of 6 on a HASWELL-EP E5-2680 v3 using OPENBLAS.

For the single-threaded case, the predictions correctly indicate that for different problem sizes different algorithms attain the best performance: While

⁴ Further six algorithms can be obtained from algorithms 1 to 3 and 5 to 6, by swapping the inversion of the diagonal A_{11} with the preceding `dtrsmRLNN` and turning the latter into a `dtrmmRLNN`; however, the resulting algorithms are also numerically unstable and thus not further discussed. For further details on the numerical stability of triangular inversion see [41].



traverse A along \searrow :

$\text{dtrmm}_{\text{RLNN}}: A_{10} := A_{10} A_{00}$

$\text{dtrsm}_{\text{LLNN}}: A_{10} := -A_{11}^{-1} A_{10}$

$\text{dtrti2}_{\text{LN}}: A_{11} := A_{11}^{-1}$

(a) Algorithm 1

traverse A along \searrow :

$\text{dtrsm}_{\text{LLNN}}: A_{21} := A_{22}^{-1} A_{21}$

$\text{dtrsm}_{\text{RLNN}}: A_{21} := -A_{21} A_{11}^{-1}$

$\text{dtrti2}_{\text{LN}}: A_{11} := A_{11}^{-1}$

(b) Algorithm 2

traverse A along \searrow :

$\text{dtrsm}_{\text{RLNN}}: A_{21} := -A_{21} A_{11}^{-1}$

$\text{dgemm}_{\text{NN}}: A_{20} := A_{20} + A_{21} A_{10}$

$\text{dtrsm}_{\text{LLNN}}: A_{10} := A_{11}^{-1} A_{10}$

$\text{dtrti2}_{\text{LN}}: A_{11} := A_{11}^{-1}$

(c) Algorithm 3

traverse A along \searrow :

$\text{dtrsm}_{\text{LLNN}}: A_{21} := -A_{22}^{-1} A_{21}$

$\text{dgemm}_{\text{NN}}: A_{20} := A_{20} - A_{21} A_{10}$

$\text{dtrmm}_{\text{RLNN}}: A_{10} := A_{10} A_{00}$

$\text{dtrti2}_{\text{LN}}: A_{11} := A_{11}^{-1}$

(d) Algorithm 4 (unstable)

traverse A along \nwarrow :

$\text{dtrmm}_{\text{LLNN}}: A_{21} := A_{22} A_{21}$

$\text{dtrsm}_{\text{RLNN}}: A_{21} := -A_{21} A_{11}^{-1}$

$\text{dtrti2}_{\text{LN}}: A_{11} := A_{11}^{-1}$

(e) Algorithm 5 (LAPACK)

traverse A along \nwarrow :

$\text{dtrsm}_{\text{RLNN}}: A_{10} := A_{10} A_{00}^{-1}$

$\text{dtrsm}_{\text{LLNN}}: A_{10} := -A_{11}^{-1} A_{10}$

$\text{dtrti2}_{\text{LN}}: A_{11} := A_{11}^{-1}$

(f) Algorithm 6

traverse A along \nwarrow :

$\text{dtrsm}_{\text{LLNN}}: A_{10} := -A_{11}^{-1} A_{10}$

$\text{dgemm}_{\text{NN}}: A_{20} := A_{20} + A_{21} A_{10}$

$\text{dtrsm}_{\text{RLNN}}: A_{21} := A_{21} A_{11}^{-1}$

$\text{dtrti2}_{\text{LN}}: A_{11} := A_{11}^{-1}$

(g) Algorithm 7

traverse A along \nwarrow :

$\text{dtrsm}_{\text{RLNN}}: A_{10} := -A_{10} A_{00}^{-1}$

$\text{dgemm}_{\text{NN}}: A_{20} := A_{20} - A_{21} A_{10}$

$\text{dtrmm}_{\text{LLNN}}: A_{21} := A_{22} A_{21}$

$\text{dtrti2}_{\text{LN}}: A_{11} := A_{11}^{-1}$

(h) Algorithm 8 (unstable)

Figure 4.13: Blocked algorithms for the inversion of a lower-triangular matrix.

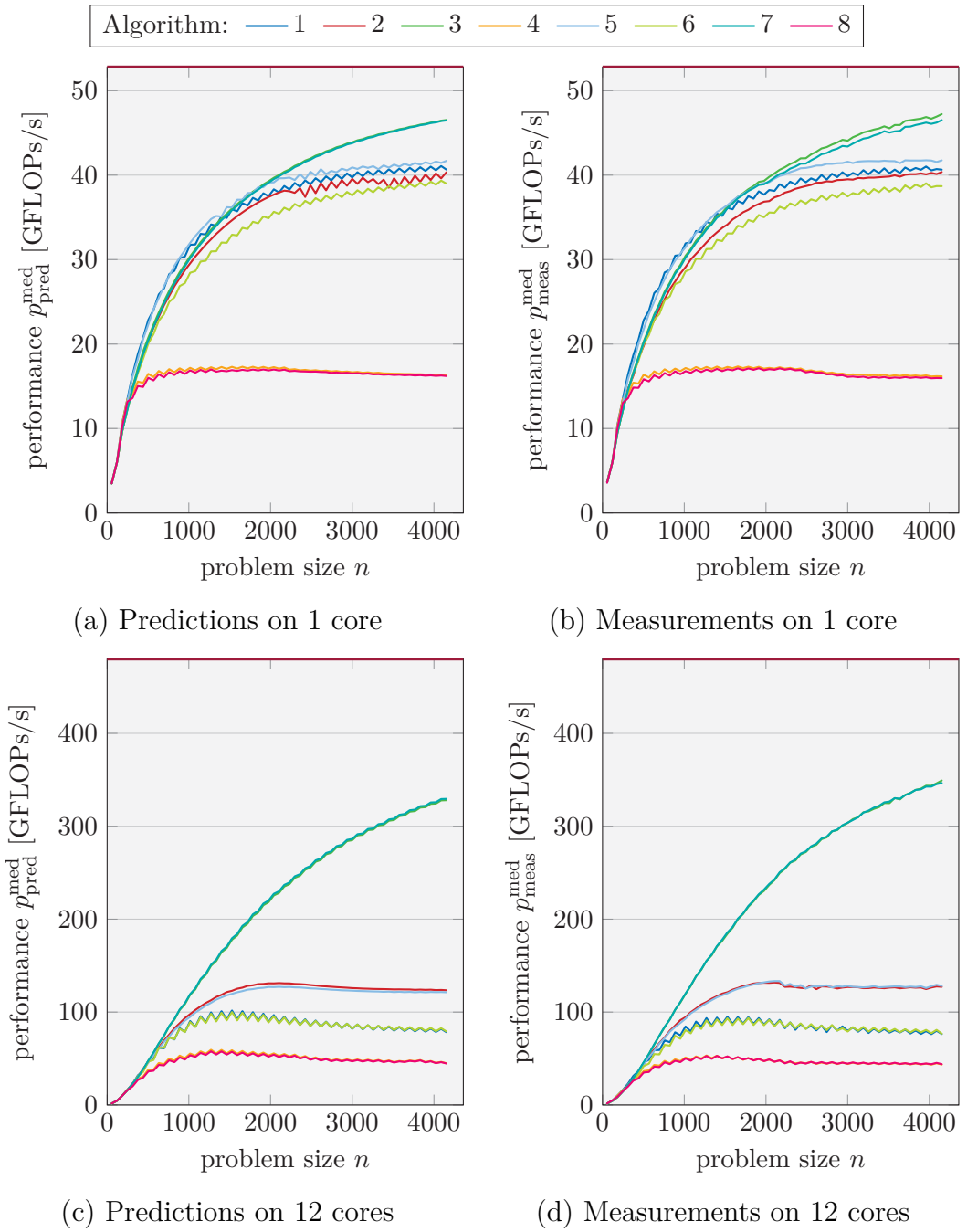


Figure 4.14: Performance measurements and predictions for the eight blocked lower-triangular inversion algorithms.

($b = 128$, HASWELL-EP [E5-2680 v3](#), OPENBLAS)

for small matrices algorithms 1 (—) and 5 (—) are faster than the third-fastest by up to 12.80 %, beyond $n \approx 1500$, algorithms 3 (—) and 7 (—) take the lead over algorithm 5 (—) in the third place by up to 13.16 % and growing. However, the predictions cannot differentiate which of the two algorithms is actually the fastest; e.g., for larger matrices, algorithm 3 (—) is up to 1.53 % faster than algorithm 7 (—).

Using all of the HASWELL's 12 cores, the predictions clearly and correctly identify that algorithms 3 (—) and 7 (—) attain the same performance, which is up to $2.73\times$ higher than the third-fastest algorithm an increasing. Furthermore, the predictions confirm that algorithms 4 (—) and 8 (—) are indeed considerably slower than all alternatives — by up to $2.96\times$ on 1 core and $7.95\times$ on 12 cores.

In summary, although our predictions in some cases cannot differentiate between algorithms with nearly identical performance, they reliably distinguish and rank algorithms with different performance.

4.5.3 Sylvester Equation Solver

The triangular⁵ Sylvester equation

$$\begin{matrix} \diagdown & \boxed{A} & \end{matrix} \boxed{X} + \boxed{X} \begin{matrix} \diagdown & \boxed{B} & \end{matrix} = \boxed{C} \quad ,$$

with $\begin{matrix} \diagdown & \boxed{A} & \end{matrix} \in \mathbb{R}^{m \times m}$, $\begin{matrix} \diagdown & \boxed{B} & \end{matrix} \in \mathbb{R}^{n \times n}$, and $\boxed{C}, \boxed{X} \in \mathbb{R}^{m \times n}$, to be solved for \boxed{X} , is commonly used in control theory and to estimate the condition numbers of eigenvalue problems. Its solution is typically implemented in-place with the \boxed{X}

⁵ The general Sylvester equation with full A and B can be reduced to this case by means of the Schur decomposition [20], which, however, results in only quasi-triangular matrices that may contain full 2×2 diagonal blocks, i.e., individual non-zero elements on the first sub-diagonal. Since each 2×2 -blocks is processed as one element, it cannot be split across sub-matrices in a blocked matrix-traversal. The resulting technical implications affect neither a blocked algorithm's structure at larger nor its performance, and we thus avoid such technicalities and assume upper-triangular A and B .

overwriting C ; LAPACK’s provides the operation in the form of the purely unblocked `dtrsylNN1`.⁶

4.5.3.1 Algorithms

The solution to the triangular Sylvester equation is computed by traversing C from the bottom left to the top right. However, in contrast to the previous operations, this traversal does not need to follow C ’s diagonal; in fact C can be traversed in various different ways: Two algorithms traverse C vertically, two horizontally (using 3×1 and 1×3 partitions), and 14 diagonally (exposing 3×3 sub-matrices), making a total of 18 algorithms. Furthermore, as detailed in the following, the Sylvester equation requires two layers of blocked algorithms, resulting in a total of 64 “complete” algorithms.

Figure 4.15 presents the four algorithms that traverse C vertically or horizontally, thereby exposing 3×1 or 1×3 sub-matrices; each of these algorithms consists of one call to `dgemmNN` and the solution of a sub-problem (another triangular Sylvester equation). To obtain a “complete” algorithm, two of these algorithms with orthogonal traversals are combined—the first traverses the full C and invokes the second to solve sub-problem in each iteration; the second, in turn, solves its small $b \times b$ sub-problem using LAPACK’s unblocked `dtrsylNN1`. E.g., one can use algorithm $m1$ to traverse C vertically and in each step apply algorithm $n2$ to traverse the middle panel C_1 horizontally. We call the resulting “complete” algorithm $m1n2$, and see that eight such combinations are possible: $m1n1$, $m1n2$, $m2n1$, $m2n2$, $n1m1$, $n1m2$, $n2m1$, and $n2m2$. Note that in principle the block sizes for the two layered blocked algorithms can be chosen independently; however, we limit our study to a single block size for both layers.

Beyond the combination of the vertically and horizontally traversing algorithms above, an additional 14 algorithms traverse the matrix diagonally (with potentially different block sizes b_m and b_n for dimensions m and n), and operate

⁶ `dtrsyl`’s first two flag arguments indicate transpositions of A and B , and the third allows to turn the operation’s left-hand-side sum into a difference.

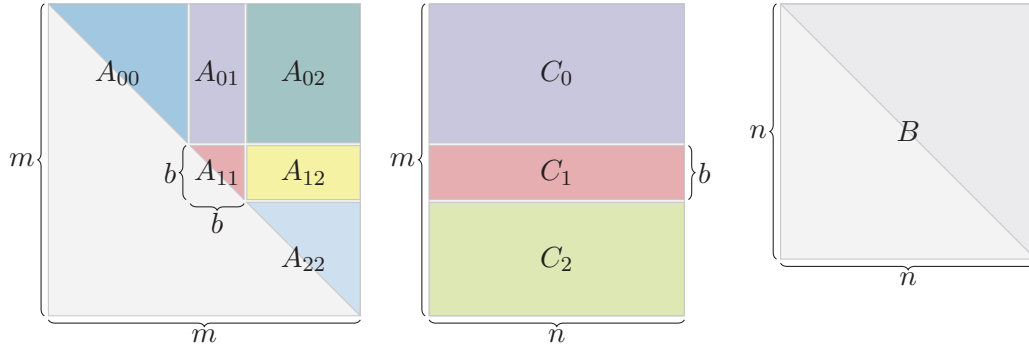
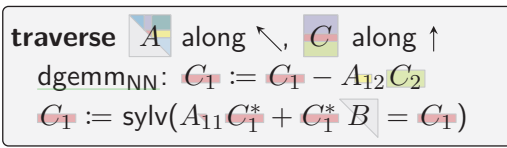
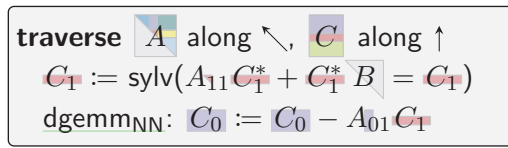
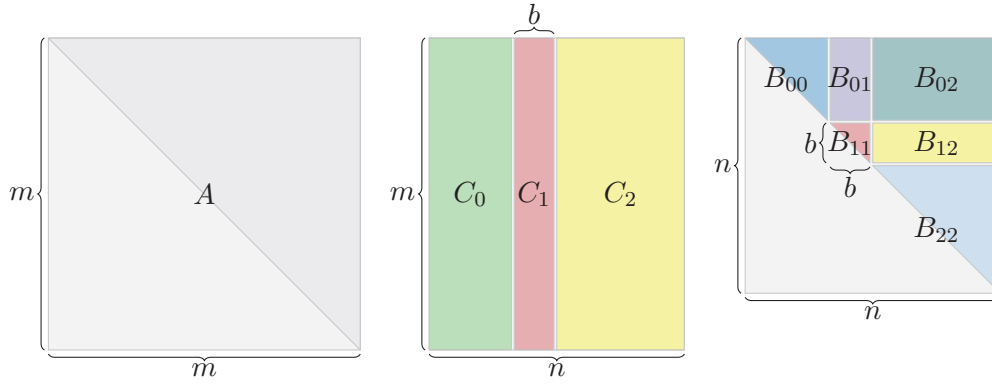
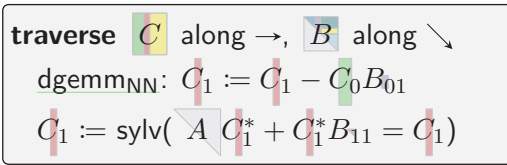
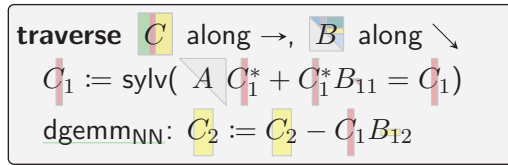
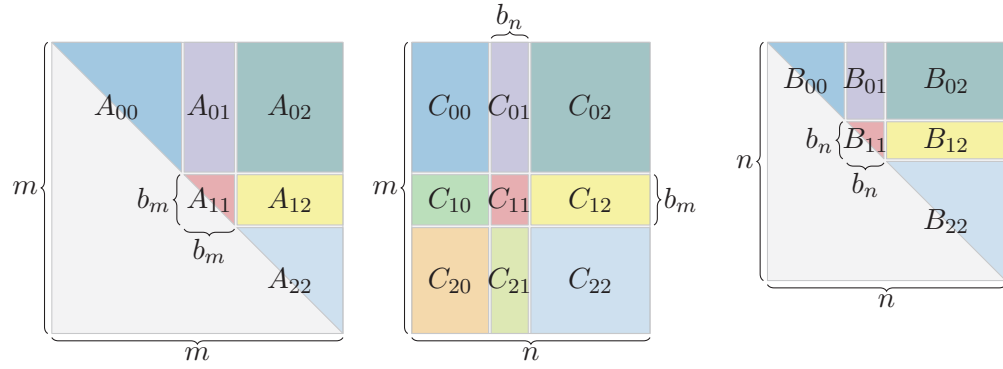

 (a) Vertical traversal of C : 3×1 matrix partitioning

 (b) Algorithm $m1$

 (c) Algorithm $m2$

 (d) Horizontal traversal of C : 1×3 matrix partitioning

 (e) Algorithm $n1$

 (f) Algorithm $n2$

 Figure 4.15: Blocked algorithms solving the triangular Sylvester equation with 1×3 and 3×1 matrix partitionings.

 (Output X overwrites input C .)


 (a) 3×3 matrix partitioning

traverse A along \swarrow , C along \nearrow , B along \searrow :
 dgemm_{NN}: $C_{10} := C_{10} - A_{12}C_{21}$
 $C_{10} := \text{sylv}(A_{11}C_{10}^* + C_{10}^*B_{00} = C_{10})$
 dgemm_{NN}: $C_{21} := C_{21} - C_{20}B_{01}$
 $C_{21} := \text{sylv}(A_{22}C_{21}^* + C_{21}^*B_{11} = C_{21})$
 dgemm_{NN}: $C_{11} := C_{11} - C_{10}B_{01}$
 dgemm_{NN}: $C_{11} := C_{11} - A_{12}C_{21}$
 dtrsyl_{NN1}: $C_{11} := \text{sylv}(A_{11}C_{11}^* + C_{11}^*B_{11} = C_{11})$

(b) Algorithm 1

traverse A along \swarrow , C along \nearrow , B along \searrow :
 $C_{21} := \text{sylv}(A_{22}C_{21}^* + C_{21}^*B_{11} = C_{21})$
 dgemm_{NN}: $C_{11} := C_{11} - A_{12}C_{21}$
 dtrsyl_{NN1}: $C_{11} := \text{sylv}(A_{11}C_{11}^* + C_{11}^*B_{11} = C_{11})$
 dgemm_{NN}: $C_{01} := C_{01} - A_{02}C_{21}$
 dgemm_{NN}: $C_{01} := C_{01} - A_{01}C_{11}$
 $C_{01} := \text{sylv}(A_{00}C_{01}^* + C_{01}^*B_{11} = C_{01})$
 dgemm_{NN}: $C_{02} := C_{02} - C_{01}B_{12}$
 dgemm_{NN}: $C_{12} := C_{12} - C_{11}B_{12}$
 dgemm_{NN}: $C_{22} := C_{22} - C_{21}B_{12}$

(c) Algorithm 10

 Figure 4.16: Sample of blocked algorithms solving the triangular Sylvester equation with 3×3 matrix partitionings.

 (Output X overwrites input C .)

on a set of 3×3 sub-matrices in each iteration; Figure 4.16 presents a sample of two of these algorithms (all 14 algorithms are found in LIBFLAME [112]). Each algorithm consists of a sequence of `dgemmNN`s and three solutions of sub-problems that are also triangular Sylvester equations. While the sub-problem involving B_{11} of size $b_m \times b_n$ is directly solved by the unblocked `dtrsylNN1`, the other two involve potentially large yet thin panels of C . Complete algorithms are constructed by solving each of these sub problems with an appropriate vertical or horizontal traversal algorithm.⁷ Since each of the 14 algorithms has two such sub-problems, for each of which we can choose from two algorithms, we end up with a total of $14 \cdot 2 \cdot 2 = 56$ possible combinations. Together with the eight combinations of only vertical and horizontal traversal algorithms, this results in a grand total of 64 different “complete” blocked algorithms.

4.5.3.2 Algorithm Selection

Figure 4.17 presents performance predictions and measurements for the Sylvester equation solver for problem sizes between $n = 56$ and 4152 in steps of 64 and block size $b = 64$ on a HASWELL-EP E5-2680 v3 using OPENBLAS. Since the executions for this setup take between 40 minutes and 2 hours for each algorithm, we only measured the eight algorithms based exclusively on orthogonal matrix traversals. Our predictions, which are generated up to $1500\times$ faster at roughly 5 s per algorithm, indicate that in terms of performance these eight algorithms are evenly spread across the entire 64 “complete” algorithms.

For the single-threaded scenario, the predictions in Figure 4.17a suggest that algorithms $n2m2$ (—) and $m1n1$ (—) are, respectively, the fastest and slowest, and differ in performance by 9.99 %. The measurements in Figure 4.17b confirm that, while algorithm $n2m2$ (—) is indeed the fastest, algorithm $n1m1$ (—) is the slowest. While the performance of algorithms $m1n1$ (—) and $n1m1$ (—) is predicted to be almost identical, the measurements show that $m1n1$ (—) is in fact up to 3.00 % faster than $n1m1$ (—). Furthermore, while the remaining

⁷ Setting one of the block sizes of a diagonally traversing algorithm to the corresponding matrix size results in one of the vertical or horizontal traversal algorithms.

4 Model-Based Predictions for Blocked Algorithms

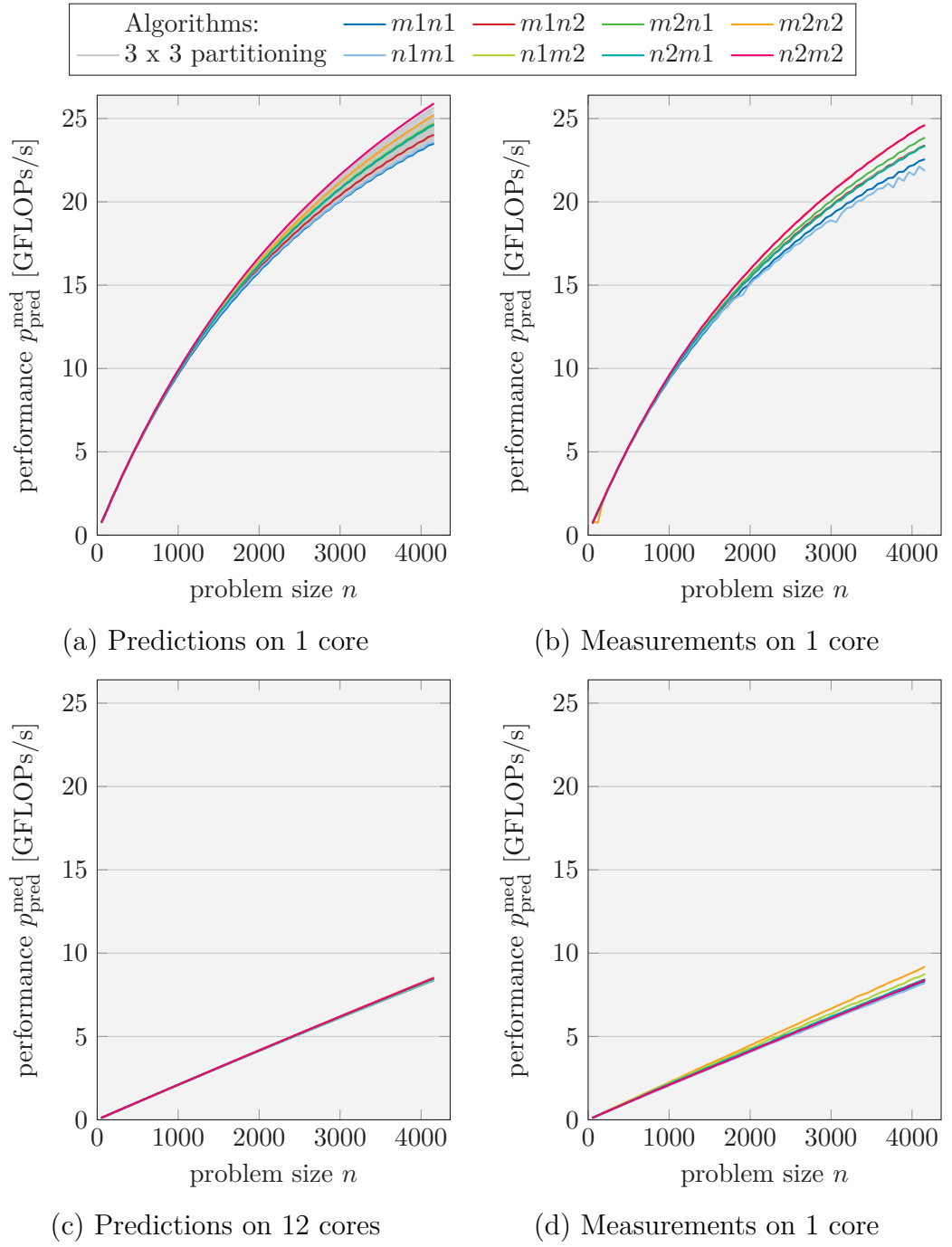


Figure 4.17: Performance predictions and measurements for the blocked triangular Sylvester equation solvers.

($b = 128$, HASWELL-EP [E5-2680 v3](#), OPENBLAS)

algorithms are correctly placed between the fastest and the slowest, they are not accurately ranked.

The predictions and measurements for the multi-threaded scenario in [Figures 4.17c](#) and [4.17d](#) are at first sight surprising: Compared to the single-threaded case the attained performance is considerably lower. For matrices of size $n = 4000$, the algorithms reach roughly 8 GFLOPs/s, which corresponds to merely 1.67 % of the processor’s 12-core peak performance of 480 GFLOPs/s (without TURBO BOOST). An analysis revealed that the source of the drastic increase in runtime is the BLAS Level 1 kernel `dswap`, which the unblocked `dtrsylNN1`⁸ uses to swap two vectors of length 4: Although the workload for this operation is tiny, with multiple threads OPENBLAS (version 0.2.15) activates its parallelisation, which for a copy operation on only 64 bytes introduces a overhead of over 200× the kernel’s single-threaded runtime. (The problem was subsequently fixed in OPENBLAS version 0.2.16 (March 2016) and is not present in MKL.)

While the multi-threaded predictions for all 64 algorithms indicate virtually identical performance and thus do not allow a meaningful performance ranking, they support the crucial revelation that using OPENBLAS 0.2.15 the triangular Sylvester equation is solved considerably faster on a single core than on 12 cores without exception.

4.5.4 Summary

We evaluated performance predictions for blocked algorithms as a means to select the fastest algorithm from a set of mathematically equivalent alternatives. We considered three operations with an increasing number of algorithms and found our predictions to rank the algorithms with great precision, thereby correctly identifying the fastest algorithm(s) in all cases. We also noted that using our model-based predictions instead of empirical measurements speeds up the identification process by two to three orders of magnitude.

⁸ Technically within `dlasy2`, which is called from `dtrsylNN1`.

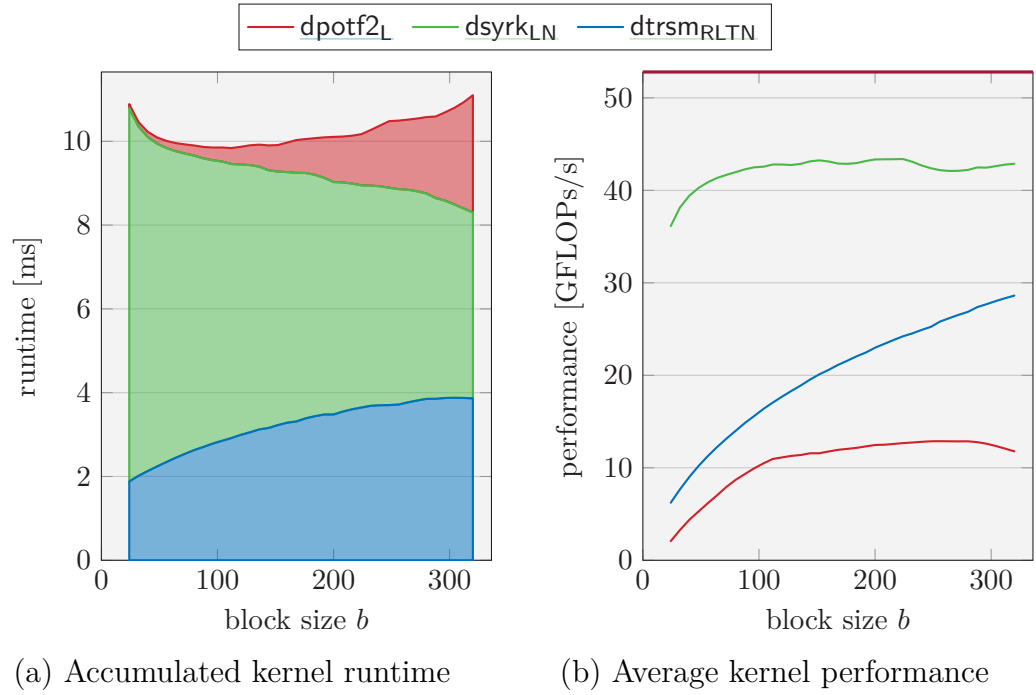


Figure 4.18: Breakdown of the blocked Cholesky decomposition algorithm 3 in terms of kernel runtime and performance.

($n = 1000$, HASWELL-EP E5-2680 v3, 1 threads, OPENBLAS, predictions)

4.6 Block Size Optimization

We now turn to our second goal for blocked algorithms: Using model-based performance predictions to optimize the algorithmic block size b .

To understand how the block size influences an algorithm's performance, recall that it determines the shape of the sub-matrices exposed in each traversal step—most notably the width of matrix panels such as A_{10} and A_{21} and the size of the square diagonal block A_{11} (see [Example 1.1](#) on 4). It hence incurs a trade-off between an increase in performance of BLAS Level 3 kernels for larger operations and a shift of computational workload to the comparatively inefficient of unblocked LAPACK kernel.

Example 4.6: Block size trade-off

We study how the kernels within the blocked Cholesky decomposition

algorithm 3 (Figure 1.1d on Page 5) contribute to its runtime for a problem of size $n = 1000$ and varying block size b on a HASWELL-EP E5-2680 v3 using single-threaded OPENBLAS. For this setup Figure 4.18 presents model-based performance estimates of (a) how much of the algorithm's runtime is spent in the kernels `dpotf2LN`, `dtrsmRLTN` and `dsyrkLN`, and (b) these kernels' average performance.

For small block sizes b , the arithmetic intensity of `dtrsmRLTN` and `dsyrkLN` is so low that they are effectively bandwidth-bound, and thus fairly inefficient. As b increases, the operands of all three kernels grow in size and so does their performance (Figure 4.18b): `dsyrkLN` (—) plateaus near 43 GFLOPs/s around $b = 100$, and while `dtrsmRLTN`'s efficiency (—) steadily rises towards that of `dsyrkLN` (—), `dpotf2LN` (—) approaches its peak of only 12 GFLOPs/s around $b = 175$. On the other hand, with increasing b more and more computation is shifted from the BLAS Level 3 routines to the inefficient `dpotf2LN` (—); beyond $b = 112$ this kernel's low performance causes the overall runtime to increase (Figure 4.18a).

In the following analysis of our model-based performance predictions, we once more consider the lower-triangular Cholesky decomposition and the inversion of a lower-triangular matrix in, respectively, Sections 4.6.1 and 4.6.2, and study three of LAPACK's blocked algorithms in Section 4.6.3.

4.6.1 Cholesky Decomposition

We revisit the Cholesky decomposition with blocked algorithm 3 (Figure 1.1d on Page 5), which Section 4.5.1 identified as the fastest. Figure 4.19 presents the algorithm's performance predictions and measurements for problem sizes $n = 1000, 2000, 3000$, and 4000 on a HASWELL-EP E5-2680 v3 using single- and multi-threaded OPENBLAS; it highlights the predicted and empirical optimal block sizes b_{pred} and b_{opt} .

In the single-threaded case, the predicted optimal block sizes b_{pred} are identical to the empirical optima b_{opt} for $n = 2000$ (—), 3000 (—), and 4000 (—). For

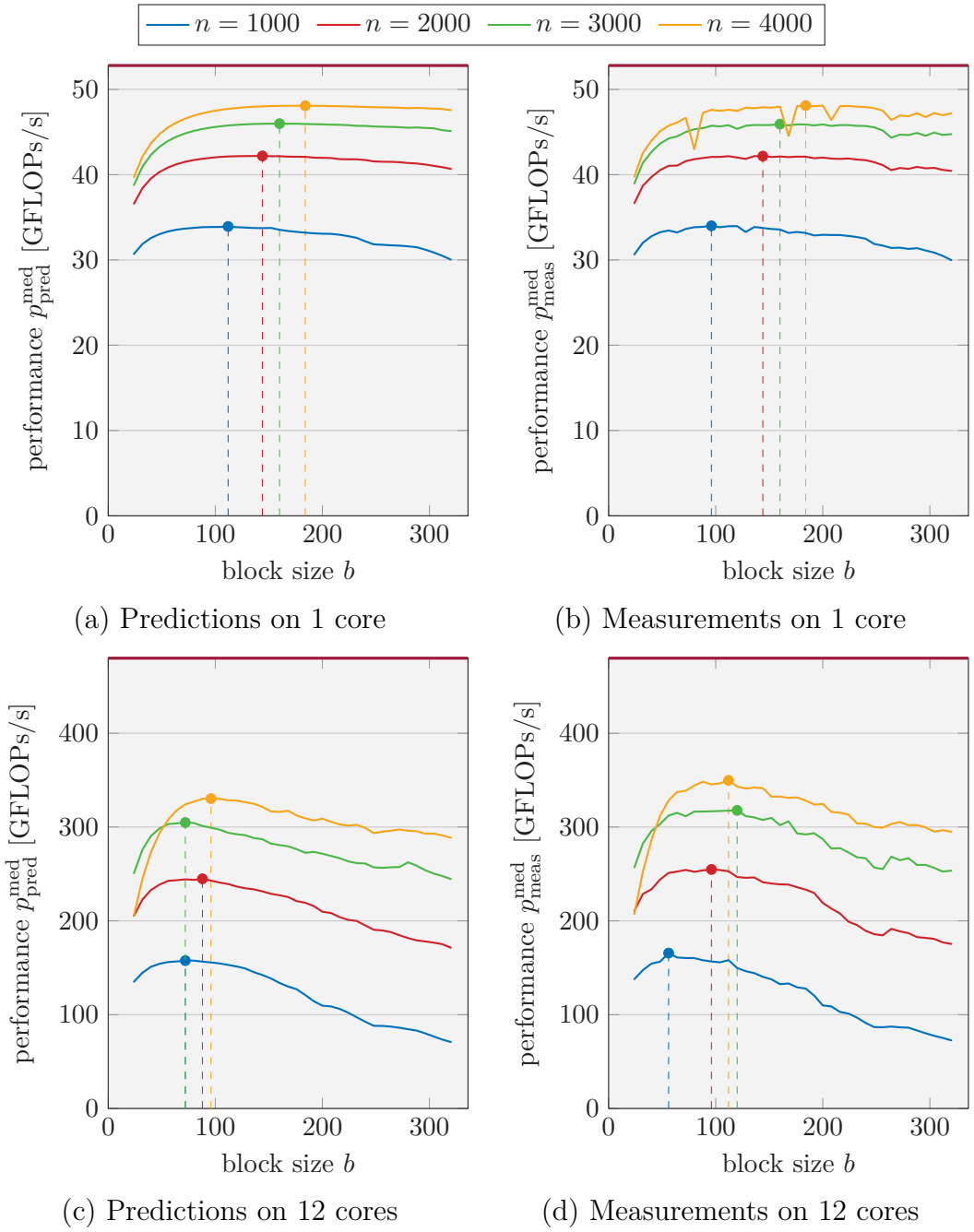


Figure 4.19: Model-based block size optimization and empirical optima for the Cholesky decomposition algorithm 3.

(HASWELL-EP [E5-2680 v3](#), OPENBLAS)

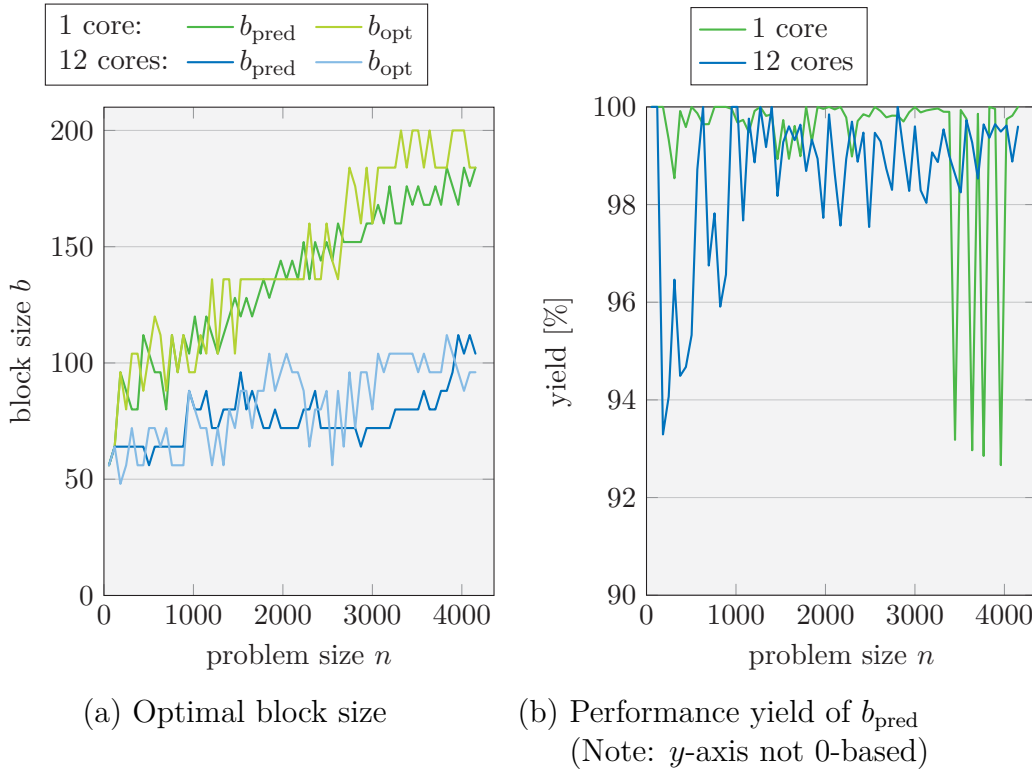


Figure 4.20: Predicted and empirical optimal block sizes and prediction yields for the Cholesky decomposition algorithm 3.

(HASWELL-EP E5-2680 v3, OPENBLAS)

$n = 1000$ (—) the predicted optimum $b_{\text{pred}} = 112$ is larger than empirical $b_{\text{opt}} = 96$, but choosing $b = 112$ nonetheless yields 99.92 % of the optimal performance.

In the multi-threaded case, the performance predictions do not match the measurements quite as well, and none of the predicted b_{pred} match the empirical b_{opt} . However, with $b = b_{\text{pred}}$ the algorithm still reaches on average 98.52 % of the optimal performance.

We expand our study to a wider range of problem sizes between $n = 56$ and 4152 in steps of 64 and analyze both how closely our predicted b_{pred} match the empirical b_{opt} and how much of the optimal performance $p_{\text{meas}}^{\text{med}}(b_{\text{opt}})$ the algorithm attains with b_{pred} . We referred to this ratio as b_{pred} 's *performance*

yield:

$$\text{yield} \stackrel{\text{def}}{=} \frac{p_{\text{meas}}^{\text{med}}(b_{\text{pred}})}{p_{\text{meas}}^{\text{med}}(b_{\text{opt}})} = \frac{t_{\text{meas}}^{\text{med}}(b_{\text{opt}})}{t_{\text{meas}}^{\text{med}}(b_{\text{pred}})} . \quad (4.7)$$

Figure 4.20a confirms that b_{pred} matches b_{opt} slightly better on one core (—, —) than on 12 cores (—, —). This is also reflected in its performance yield presented in Figure 4.20b: On 1 core, the average yield is 99.35% while on 12 cores it is slightly lower at 98.57%.

Note that for this study we measured the runtime of the algorithm 10 times for each considered problem size n and block size b , which took almost 2 hours in the single-threaded case and around 20 minutes with 12 threads—in contrast the predictions for the same range of sizes were obtained in under a minute in both cases.

4.6.2 Triangular Inversion

we repeat the above study for the inversion of a lower-triangular matrix with blocked algorithm 3 (Figure 4.13c), which was shown to be the fastest for large problem sizes in Section 4.5.1. Figure 4.21 presents (a) the predicted and empirical optimal block sizes b_{pred} and b_{opt} using single- and multi-threaded OPENBLAS, and (b) b_{pred} 's performance yields.

Figure 4.21a shows that, in contrast to the Cholesky decomposition (Figure 4.20a), the optimal block sizes for the single- and multi-threaded inversion of a lower-triangular matrix are fairly similar, yet slightly lower in the single-threaded case. However, the empirical b_{opt} exhibits a behavior not well represented by the predicted b_{pred} : Beyond $n \approx 2000$, the multi-threaded b_{opt} (—) assumes only two values—96 and 192—while our prediction b_{pred} indicates a more gradual transition. (On 1 core the effect is similar with $b_{\text{opt}} = 96$ for almost all problem sizes beyond $n \approx 1700$.) The cause for this problem is that our models only poorly represent certain spikes in the performance of the multi-threaded `dsyrkLN` implementation at the optimal block sizes.

The sub-optimal choices of block sizes are reflected in the prediction yields: Figure 4.21b shows that, while on a single core the yield is almost ideal

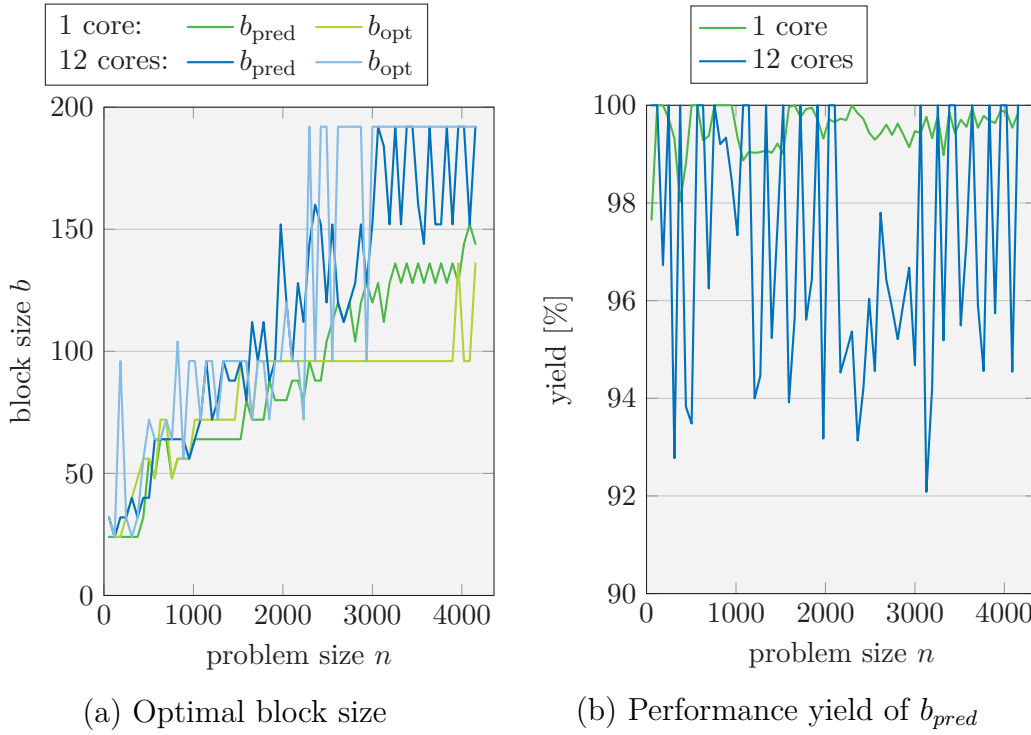


Figure 4.21: Predicted and empirical optimal block sizes and prediction yields for the inversion of a lower-triangular matrix algorithm 3.

(HASWELL-EP E5-2680 v3, OPENBLAS)

at 99.53 %, on 12 cores, it drops notably—especially for larger problems—averaging 97.13 %.

4.6.3 LAPACK Algorithms

To conclude our study on block size optimization we consider three of LAPACK’s blocked algorithms on square matrices:

- `dsygst1L`: $\begin{bmatrix} A \end{bmatrix} := \begin{bmatrix} L^{-1} A \end{bmatrix} \begin{bmatrix} L^{-T} \end{bmatrix}$ (Figure 4.8b),
- `dgetrf`: $\begin{bmatrix} P \end{bmatrix} \begin{bmatrix} L \end{bmatrix} \begin{bmatrix} U \end{bmatrix} := \begin{bmatrix} A \end{bmatrix}$ (Figure 4.8e), and
- `dgeqrf`: $\begin{bmatrix} Q \end{bmatrix} \begin{bmatrix} R \end{bmatrix} := \begin{bmatrix} A \end{bmatrix}$ (Figure 4.9).

4 Model-Based Predictions for Blocked Algorithms

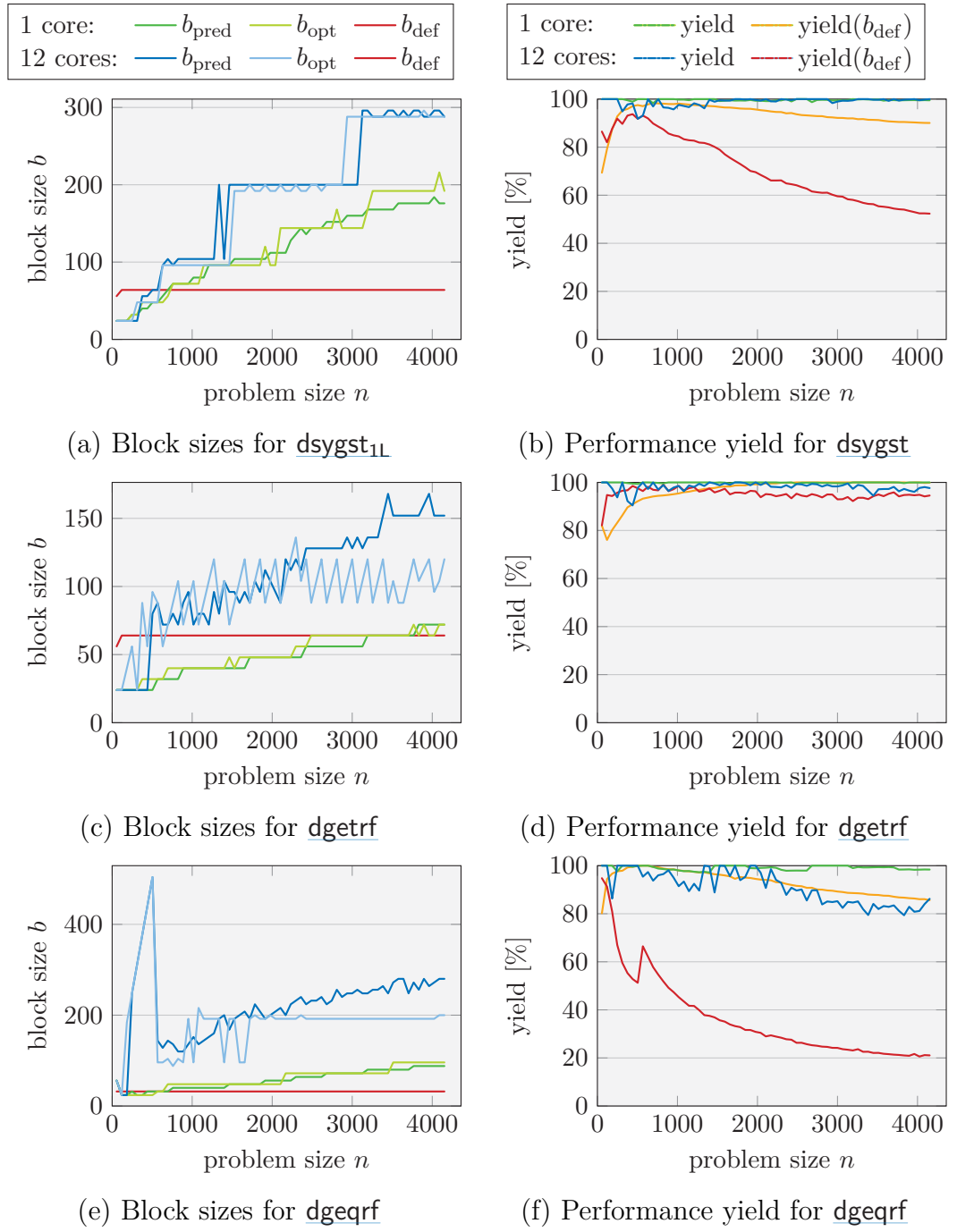


Figure 4.22: Predicted and empirical optimal block sizes and prediction yields for dsygst_{1L}, dgetrf, and dgeqrf.

(HASWELL-EP E5-2680 v3, OPENBLAS)








			<u>dsygst_{1L}</u>	<u>dgetrf</u>	<u>dgeqrf</u>
1 core	yield	()	99.64 %	99.90 %	99.05 %
	yield(b_{def})	()	93.64 %	96.92 %	92.92 %
	improvement		6.70 %	3.41 %	6.89 %
12 cores	yield	()	98.93 %	98.10 %	90.98 %
	yield(b_{def})	()	70.76 %	95.23 %	36.23 %
	improvement		45.18 %	3.08 %	189.64 %






Table 4.5: Average performance yields and improvement over LAPACK for dsygst_{1L}, dgetrf, and dgeqrf.

($n = 56, \dots, 4152$ in steps of 64, $b_{\text{def}} = 64$, except dgeqrf: $b_{\text{def}} = 32$, HASWELL-EP E5-2680 v3, OPENBLAS)

For these routines, the left half of Figure 4.22 presents the predicted and empirical optimal block sizes b_{pred} and b_{opt} , as well as LAPACK's default block size b_{def} (dsygst_{1L}, dgetrf: 64; dgeqrf: 32); and the right half shows the performance yields for both b_{pred} and b_{def} . Furthermore, Table 4.5 summarizes the yields for these block sizes averaged across the chosen problem sizes.

For the single-threaded operations our predicted optimal block sizes b_{pred} () match the empirical optimum b_{opt} () quite well, resulting in an average performance yield () of 99.53 %. For both dsygst_{1L} and dgeqrf, the optimal block size quickly exceed LAPACK's default values, leading to an improved performance of roughly 10 % (dsygst_{1L}) and 15 % (dgeqrf). For dgetrf on the other hand, LAPACK's $b_{\text{def}} = 64$ is actually ideal between $n = 2500$ and 3700, meaning our predicted b_{pred} only yields improvements for smaller problem sizes.

In the multi-threaded case, the optimal block sizes are across the board larger.

- For dsygst_{1L}, b_{opt} () is correctly predicted () to jump from ≈ 100 to ≈ 200 around $n = 1500$, and next to ≈ 290 at $n \approx 3000$. These predictions yield 98.93 % () of the optimal performance, which is an average 45.18 % improvement over LAPACK's $b_{\text{def}} = 64$ ()—reaching up to ≈ 90 % for $n \approx 4000$.
- For dgetrf, the optimal block size b_{opt} () fluctuates constantly with a

magnitude of 32, which is not represented by the prediction b_{pred} (—). However, the general trend is captured fairly well up to $n \approx 2500$, after which b_{opt} stagnates, while b_{pred} increases further. As a result, the performance yield (—) decreases slightly beyond $n = 3000$, yet retains a high average of 98.10 %. Since `dgetrf` is generally less sensitive to the block size, LAPACK’s $b_{\text{def}} = 64$ also yields above 95 % (—) of the optimal performance.

- For `dgeqrf`, the unblocked `dgeqr2` is faster for small problem sizes than the blocked algorithm with any block size, which translates to $b_{\text{opt}} = n$;⁹ this behavior is for the considered block sizes up to $b = 536$ correctly predicted. The trend of b_{def} ’s yield in Figure 4.22f suggests that `dgeqr2` may continue to be faster than `dgeqrf` until $n \approx 1000$. Beyond this point, b_{opt} (—) jumps between ≈ 100 and ≈ 200 until $n \approx 2000$, after which it remains around $b_{\text{opt}} = 200$. Since our predicted b_{pred} (—) indicates a smoother increase beyond $n = 2000$, the performance yield (—) eventually drops to ≈ 84 %. Compared to the yield of LAPACK’s $b_{\text{def}} = 32$ (—), however, this is still a major improvement of up to $4\times$, averaging 189.64 %.

In summary, our model-based predictions of the optimal block size show varying degrees of accuracy, yet consistently provide performance improvements over LAPACK’s default block sizes by up to 300 %. Note that while the measurements for the above study took in total almost 4 days, all corresponding predictions were obtained from our models in under 25 minutes. While choosing a coarser set of samples (i.e., fewer problem and block sizes) for the empirical optimization might reduce its runtime to below 10 hours, our predictions, to which the same reduction can be applied, would still provide a significant speedup. By porting our currently PYTHON-based models to other formats to be evaluated in a faster language (e.g., in C/C++), we expect that this prediction time can be reduce mere seconds.

⁹ To leverage the performance of optimized BLAS Level 3 through a blocked algorithm, `dgeqrf` performs $O(bn^2)$ FLOPs more than `dgeqr2`. The fact that `dgeqr2` is faster than `dgeqrf` not only for small problems indicates that these extra FLOPs are not easily amortized in the multi-threaded scenario.

4.7 Summary

This chapter presented performance predictions for blocked algorithms based on the performance models described in [Chapter 3](#). These predictions were found to closely match the measured performance of blocked LAPACK algorithms in a variety of setups. They allow us to solve two important problems without any algorithm executions:

- We can rank alternative blocked algorithms according to their performance, and thereby identify the fastest algorithm for various operations.
- We can select near-optimal block sizes that lead algorithms to within a few percent of their empirically optimal performance; they are often an enormous improvement over LAPACK's default block sizes.

Since our models predict algorithm executions two to three orders of magnitude faster than corresponding empirical measurements, they make previously disproportionately time-consuming optimization processes feasible.

5 Cache Modeling and Prediction

The previous chapter introduced the concept of model-based performance predictions for dense linear algebra algorithms. While such predictions are accurate for many scenarios, we observed a degradation in accuracy for operands larger than the processor’s last-level cache. This chapter analyzes such caching effects and explores how they can be accounted for in predictions.

[Section 5.1](#) presents a case study on LAPACK’s blocked QR decomposition `dgeqrf` on a HARPertown [E5450](#) using OPENBLAS, and details efforts to accurately estimate the runtime of the kernel invocations within `dgeqrf` by combining isolated in- and out-of-cache timings. Next, [Section 5.2](#) applies the developed approach to two further LAPACK algorithms. Finally, [Section 5.3](#) attempts to employ the same concepts on more recent hardware, and reveals limitations to how well isolated kernel timings can predict an algorithm’s total runtime.

Publication

The work presented in this chapter—in particular [Sections 5.1](#) and [5.2](#)—is in parts based on research previously published in:

- [5] Elmar Peise and Paolo Bientinesi. “A Study on the Influence of Caching: Sequences of Dense Linear Algebra Kernels”. In: *High Performance Computing for Computational Science – VECPAR 2014: 11th International Conference*. Volume 8969. Lecture Notes in Computer Science. Springer International Publishing, Apr. 2015, pages 245–258. DOI: [10.1007/978-3-319-17353-5_21](https://doi.org/10.1007/978-3-319-17353-5_21).

5.1 Case Study: QR Decomposition on a HARPertown E5450

We focus on a specific, yet exemplary algorithm and setup: We analyze the performance of LAPACK’s QR decomposition `dgeqrf`

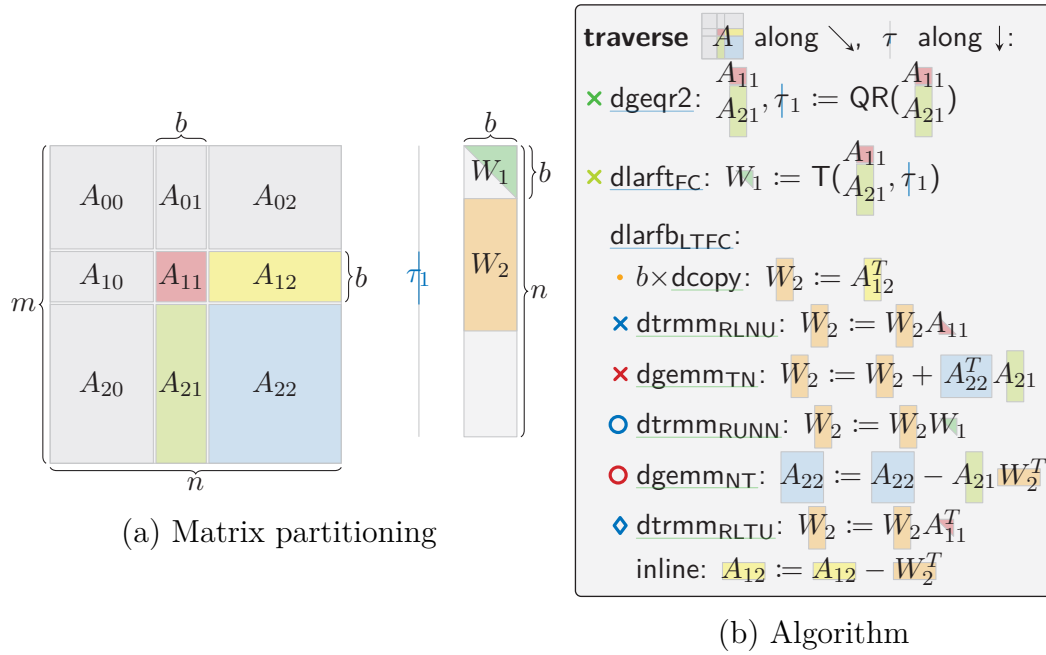
$$\boxed{Q} \boxed{R} := \boxed{A}$$

of a square matrix $\boxed{A} \in \mathbb{R}^{1568 \times 1568}$ with LAPACK’s default block size $b = 32$ on a HARPertown E5450 using single-threaded OPENBLAS—with a size of about 18 MiB, \boxed{A} exceeds the processor’s last-level cache (L2) of 6 MiB per 2 cores.

In the following, [Section 5.1.1](#) presents the blocked algorithm behind `dgeqrf` and instrumentation-based in-algorithm timings that serve as the reference for our per-kernel runtime predictions. Next, [Section 5.1.2](#) measures the runtime of each kernel invocation in isolation with cache preconditions, and establishes in- and out-of-cache timings as, respectively, lower and upper bounds on the in-algorithm timings. [Section 5.1.3](#) combines the in- and out-of-cache timings to estimate the in-algorithm timing by tracking which parts of the kernel operands reside in the processor’s L2 cache prior to the invocation. Finally, [Section 5.1.4](#) expands the introduced methodology beyond the initially considered instance of the blocked QR decomposition towards other scenarios on the HARPertown E5450, including other matrix and block sizes, BLAS implementations, and kernel parallelism.

5.1.1 Timing Kernels in LAPACK’s `dgeqrf`

[Figure 4.9](#) outlines the blocked algorithm employed by LAPACK’s QR decomposition `dgeqrf`. The algorithm overwrites \boxed{A} ’s upper-triangular part with \boxed{R} , and stores \boxed{Q} as the combination of 1) a series of elementary reflectors in \boxed{A} ’s strictly lower-triangular portion, and 2) a separate output vector of scalar


 Figure 5.1: LAPACK's blocked algorithm for `dgeqrf`.

The routine markers (✕, ♦, etc.) are references for following plots.

factors τ . It furthermore requires an auxiliary matrix $W \in \mathbb{R}^{n \times b}$ for temporary data.

`dgeqrf` itself invokes only three routines: the unblocked QR decomposition `dgeqr2`, the formation of the triangular block reflector T (stored in W_1) through the unblocked `dlarftFC`,¹ and the application of the block reflector through `dlarfbLTFC`. The latter in turn is implemented largely in terms of the BLAS Level 3 kernels `dtrmm` and `dgemm`; it furthermore performs a transposed matrix copy through a series of b `dcopys`, and an inlined transposed matrix subtraction.²

To measure the runtime of the kernels within the QR decomposition—henceforth called *in-algorithm timings*—we manually instrument `dgeqrf` and `dlarftFC`, and collect timestamps (through the x86 instruction `rdtsc`) between kernel invocations. For the studied algorithm execution, Figure 5.2a presents

¹ The flags `direct = F` and `storev = C` indicate that the reflectors are stored in *forward* order and as *column* vectors.

² A series of b `daxpys` would likely be more efficient.

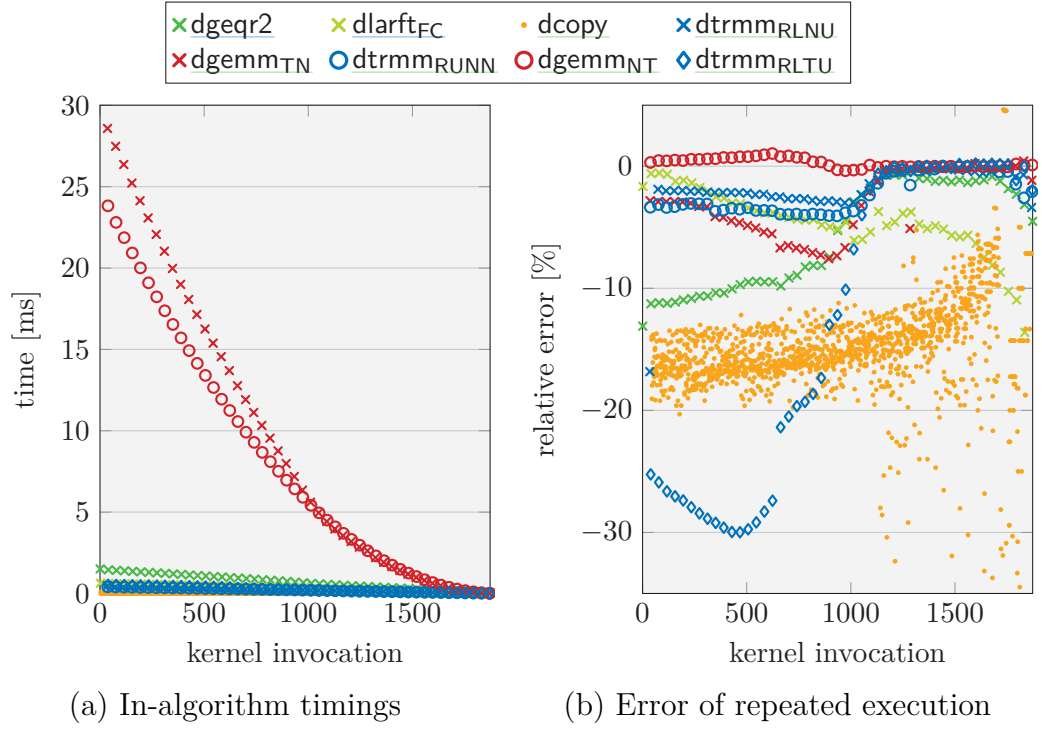


Figure 5.2: In-algorithm timings and error of repeated execution timings with respect to these for the 1873 kernel invocations within `dgeqrf`.

($n = 1568$, $b = 32$, HARPERTOWN E5450, 1 thread, OPENBLAS, median of 100 repetitions)

the in-algorithm timings computed from these timestamps: The x -axis enumerates the 1873 kernel invocations,³ for each of which one data-point presents the kernel runtime. The total execution time (946.68 ms) is dominated by the two `dgemms` (`x`, `o`); although the size of their operands is the same, their runtimes differ significantly. Our ultimate goal is to develop a strategy to accurately predict the runtimes for all kernel invocations without executing `dgeqrf` itself.

5.1.2 Cache-Aware Timings

We begin to predict the in-algorithm timings with an elementary setup: *repeated execution* of the kernels in isolation. In these executions, which are performed

³ $n/b - 1 = 1568/32 - 1 = 48$ traversal steps à 39 kernels (`dgeqr2`, `dlarfb`, $b = 32$ `dcopys`, 3 `dtrmms`, and 2 `dgemms`) and 1 final `dgeqr2`: $48 \times 39 + 1 = 1873$.

5.1 Case Study: QR Decomposition on a *HARPERTOWN E5450*

one right after the other without any modifications to the data, we use the same flags and matrix sizes as within `dgeqrf` and well separated memory locations as operands. Figure 5.2b shows the relative runtime error for the median of 100 such independent repetitions with respect to the in-algorithm timings. While the relative error for `dcopy` (•) is rather large, the total contribution of its 1536 invocations to the algorithm’s runtime is below 1 %. Not considering these `dcopys`, the absolute relative error of the repeated execution runtime estimates relative to the in-algorithm timings averaged across all kernel invocations—in the following simply referred to as *error*—is 4.42 %.

For most routines and especially for `dtrmmRLTU` (◊) and `dgeqr2` (✕), the repeated execution timings underestimates the in-algorithm timings for the first 1000 kernel invocations. More surprisingly however, `dgemmNT` (○) is even overestimated—it is faster within `dgeqrf`.

The change around the 1000th kernel invocation in Figure 5.2b is directly linked to the cache: While traversing the matrix, `dgeqrf` only operates on A ’s bottom-right quadrant, which becomes smaller in step, and beyond invocation 1000 fits in the L2 cache. As a result, the subsequent runtime measurements of repeated executions show only minimal differences with respect to the in-algorithm timings. This confirms caching as the cause of the discrepancies.

To better understand the scope of this influence we manipulate the cache locality of the kernel operands in our isolated executions. For this purpose, we *assume* a simplified cache replacement policy: a *fully associative Least Recently Used* (LRU) algorithm. We consider the two extreme scenarios in which the operands immediately required by the kernels are either entirely in the L2 cache or not cached at all. These in- and out-of-cache scenarios serve as, respectively, lower and upper bounds on the in-algorithm timings.

For kernels with operands smaller than 6 MiB, repeated execution suffices to guarantee an in-cache setup. By contrast, when the aggregate size of the operands exceeds 6 MiB (as for `dgemmNT` (○)), different kernel implementations may initially access different memory portions. An ideal in-cache setup would place exactly these immediately accessed portions in cache. However, since we do not assume knowledge of kernel implementation, we restrict our in-

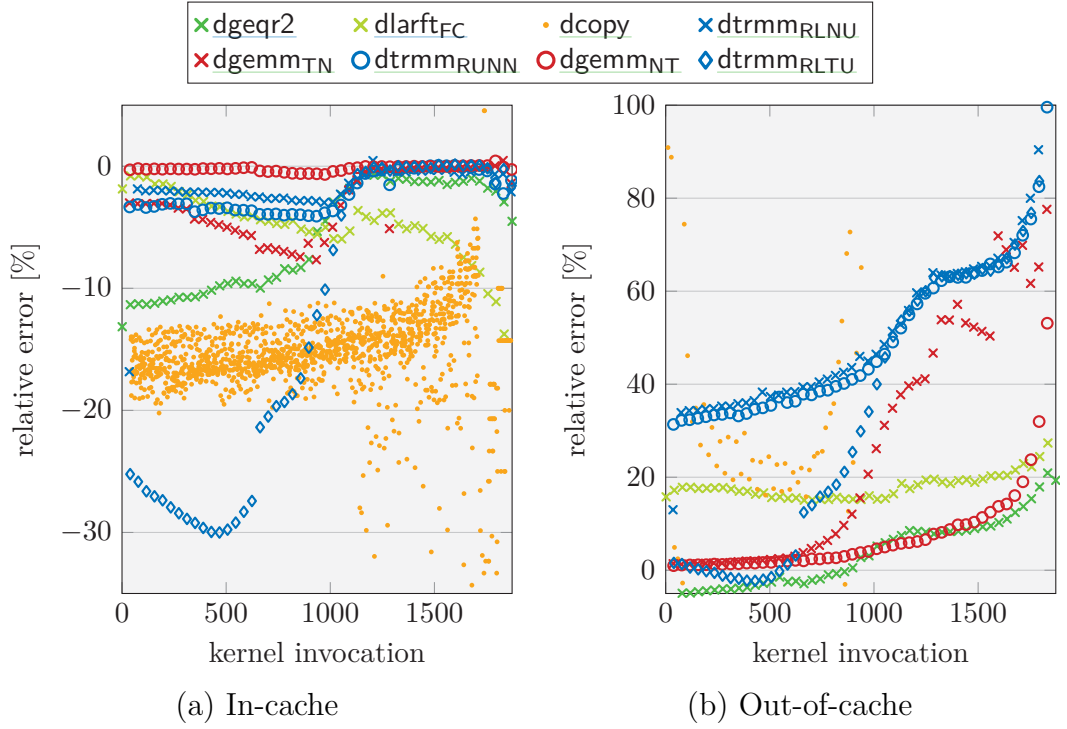


Figure 5.3: Error of in- and out-of-cache timings with respect to in-algorithm timings for `dgeqrf`. The out-of-cache errors for `dcopy` (•) are around 1000 %.

($n = 1568$, $b = 32$, HARPERTOWN E5450, 1 thread, OPENBLAS, median of 100 repetitions)

cache setup to fulfill the reasonable assumption that input-only operands are accessed before input/output and output-only operands. In order to prepare the cache accordingly, we load⁴ all input-only operands into the cache just before the kernel invocation. Figure 5.3a compares the such obtained *in-cache timings* to the in-algorithm timings: The estimates are in all cases equal to or underestimating the in-algorithm timings;⁵ the error is 4.44 %.

To ensure that the operands are not in the cache, it suffices to access a main memory section larger than the cache size. Figure 5.3b compares this setup's *out-of-cache timings* to the in-algorithm timings: Almost all estimates are equal to or overestimating the in-algorithm timings; the error is 29.14 %.

⁴ Through a simple update to each data element, e.g., $x := x + \varepsilon$.

⁵ To be precise, the largest overestimation is 0.06 %.

Not only do the established in- and out-of-cache timings indeed serve as lower and upper bounds on the in-algorithm timings; for most kernel invocations one of these two bounds is actually attained (see [Figures 5.3a](#) and [5.3b](#)). Based on this observation, the next section introduces a cache model to combine these in- and out-of-core timings to estimate the in-algorithm timings.

5.1.3 Modeling the Cache

To predict the state of the cache throughout the execution of `dgeqrf`, we consider which parts of A and W are accessed by each kernel invocation. We examine the sequence of kernel invocations within `dgeqrf` (see [Figure 4.9](#)), but, due to the lack of information on the implementations of these kernels, make no assumptions on the patterns in which the kernels access their operands.

For the assumed fully associative LRU cache replacement policy, identifying if a kernel operand is in cache boils down to counting how many other data elements were accessed since its last use. To determine this count—henceforth referred to as *access distance*—we scan the sequence of kernel invocations and keep a history of the memory regions they access.⁶ (Note that for our purposes cache lines are the smallest accessible units of memory: An access to a single data element means an access to the entire surrounding cache line.) For each operand, we go backward through the access history until (and including) we find its last occurrence; thereby summing the sizes of the encountered memory regions yields the operand’s access distance. If a previous access is not found, the access distance is set to the total size of A and W .⁷

By comparing the obtained access distances to the cache size, we determine whether the corresponding operand is expected to be in the cache or not. Given these expectations, we separately sum the sizes of the in- and out-of-cache operands to, respectively, s_{ic} and s_{oc} . These sums are then used to weight the runtime of the corresponding timings t_{ic} and t_{oc} to yield *initial estimates* of the

⁶ The length of this history is restricted to the number of kernel calls per iteration of the blocked algorithm.

⁷ This corresponds to the scenario where the entire QR decomposition is repeatedly executed on the same data.

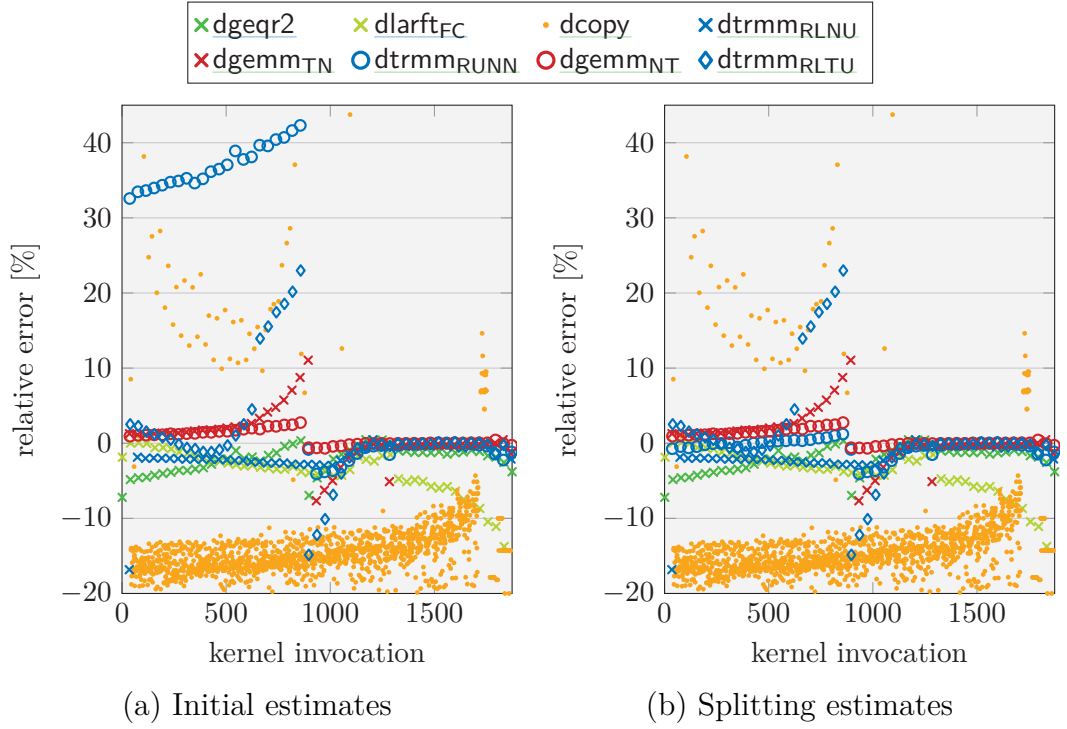


Figure 5.4: Error of our initial and splitting estimates with respect to in-algorithm timings for `dgeqrf`.

($n = 1568$, $b = 32$, HARPERTOWN E5450, 1 thread, OPENBLAS, median of 100 repetitions)

in-algorithm timings:

$$t_{\text{est}} := \frac{s_{\text{ic}}t_{\text{ic}} + s_{\text{oc}}t_{\text{oc}}}{s_{\text{ic}} + s_{\text{oc}}} . \quad (5.1)$$

Comparing these estimates in Figure 5.4a to Figures 5.3a and 5.3b, we find that our mechanism chooses (or weights) the in-cache and out-of-cache timings correctly for most kernels. However, the error is 4.61 %, because for `dtrmm_RUNN` (○) out-of-cache is erroneously favored over in-cache.

The reason for this flaw is that (see Figure 4.9) `dtrmm_RUNN` (○) is preceded by the large `dgemm_TN` (×): This `dgemm`'s operands, which are together larger than the cache, are accumulated into the access distance of `dtrmm_RUNN`'s operand W_2 . However, since W_2 happens to be the output of the matrix-times-vector-shaped `dgemm_TN`, it appears to be left in cache. We use this insight to extend our cache model with a crucial assumption: After a kernel whose

5.1 Case Study: QR Decomposition on a *HARPERTOWN E5450*

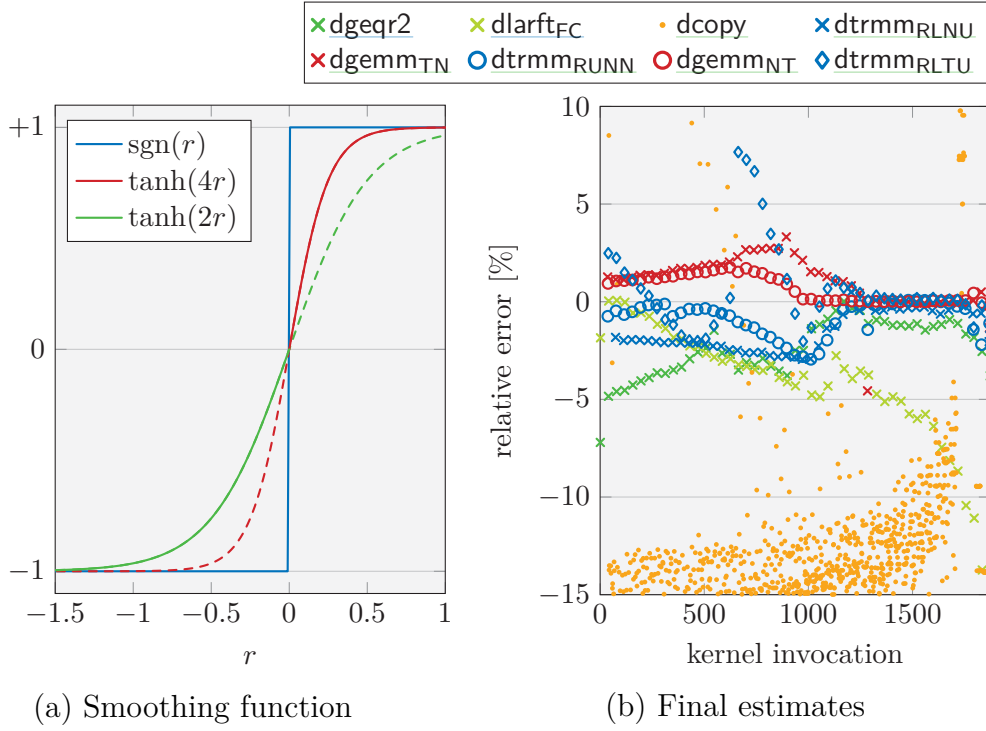


Figure 5.5: Smoothing function and error of final estimates with respect to in-algorithm timings for `dgeqrf`.

($n = 1568$, $b = 32$, *HARPERTOWN E5450*, 1 thread, OPENBLAS, median of 100 repetitions)

(input/output) operand is significantly smaller than its input-only operands we expect the (input/output) operand to remain in cache. This assumption is implemented by splitting the memory accesses of such a kernel into two parts: The first part contains the large input-only operand(s), while the second only involves the small (input/output) operand. Therefore, the back-traversal of the access history encounters the latter separately, and, in case it is the sought operand, terminates before processing the cache-exceeding accesses. The runtime estimates from this modifications—called *splitting estimates*—are evaluated in Figure 5.4b: All kernels are chosen correctly from the in-cache and out-of-cache timings; as a result, the error is reduced to 2.24 %.

The only remaining deficiency of our estimates is the cluster of spikes around the transition from out-of-cache to in-cache around the 900th kernel invocation.

5 Cache Modeling and Prediction

To avoid such spikes, we “smooth” the association of operands to in- and out-of-cache. To determine whether an operator $\mathcal{O}p$ is in-cache (+1) or out-of-cache (−1), we previously used a step function. In terms of the relative access distance

$$r_{\mathcal{O}p} = \frac{(\text{cache size}) - (\text{access distance})_{\mathcal{O}p}}{\text{cache size}} ,$$

this was the sign function: Based on the operand sizes $s_{\mathcal{O}p}$ the weights for our estimates (Equation (5.1)) were computed as

$$s_{\text{ic}} := \sum_{\mathcal{O}} p \frac{1 + \text{sgn}(r_{\mathcal{O}p})}{2} s_{\mathcal{O}p} \quad \text{and} \quad s_{\text{oc}} := \sum_{\mathcal{O}} p \frac{1 - \text{sgn}(r_{\mathcal{O}p})}{2} s_{\mathcal{O}p} .$$

We now replace the association function with

$$f(r) = \begin{cases} \tanh(\alpha r) & \text{for } r \geq 0 \\ \tanh(\beta r) & \text{for } r < 0 \end{cases} ,$$

where α and β are smoothing coefficients. As shown in Figure 5.5a, $f(r)$ converges toward $\text{sgn}(r)$ for both large and small values of r , and exhibits a smooth transition from −1 to +1 through the origin. When applied to our estimates with empirical values of $\alpha = 4$ and $\beta = 2$, we obtain the *final estimates* evaluated in Figure 5.5b. With all estimates close to the instrumentation timings, the error further decreases to 1.80 %.

5.1.4 Varying the Setup

In the previous sections we focused on one specific setup for the QR decomposition `dgeqrf` on a HARPertown E5450: We factorized a square matrix of size $n = 1568$ with block size $b = 32$ using single-threaded OPENBLAS. To demonstrate that our observations and models are more broadly applicable, we now vary this setup: For a range of scenarios Table 5.1 presents the improvements of our final estimates (e.g., Figure 5.5b) over the repeated execution timings (e.g., Figure 5.2b).

Although the error of our estimates remains above 1.5 %, they are in many

5.1 Case Study: QR Decomposition on a *HARPERTOWN E5450*

#cores	BLAS	n	b	repeated execution	final estimates	improvement
1	OPENBLAS	1568	32	4.42 %	1.80 %	2.46×
1	OPENBLAS	1568	64	3.15 %	1.64 %	1.92×
1	OPENBLAS	1568	128	2.68 %	2.13 %	1.26×
1	OPENBLAS	2080	32	5.11 %	1.84 %	2.78×
1	OPENBLAS	2400	32	5.23 %	1.75 %	2.99×
1	ATLAS	1568	32	3.55 %	1.98 %	1.79×
1	ATLAS	2400	32	4.22 %	2.51 %	1.68×
1	MKL	1568	32	8.58 %	4.40 %	1.95×
1	MKL	2400	32	9.58 %	6.22 %	1.54×
1	reference	1568	32	2.31 %	1.54 %	1.50×
2	OPENBLAS	1568	32	9.58 %	4.63 %	2.07×
4	OPENBLAS	1568	32	22.71 %	19.75 %	1.15×

Table 5.1: Estimation errors and improvements through cache-modeling for `dgeqrf`.

(*HARPERTOWN E5450*)

cases an improvement of about $2\times$ over the repeated execution timings. For both increasing block size b and problem size n the accuracy of the repeated executions timings varies, but our estimates reliably yield an error of around 2%.⁸ Changing the BLAS implementation, we can appreciate that with ATLAS the results are much the same as with OPENBLAS. While with MKL the error in both the repeated execution timings and our estimates instead increases significantly, the estimates are still a good improvement of the repeated execution timings. Even for the reference BLAS implementation our estimates improve the already low error further by a factor of 1.5. When doubling the number of cores to 2, the errors increase, but our estimates still provide a $2\times$ improvement over the repeated execution timings. When we use all 4 of our processor’s cores however, the error increases drastically—mainly because, while our model is designed for a single last-level cache, every two cores of the *HARPERTOWN* share a separate L2 cache. To account for multiple last-level caches, would

⁸ Since for larger block sizes the arithmetic intensity of the kernels increases, caching plays a smaller role and the repeated execution estimates become more accurate on their own.

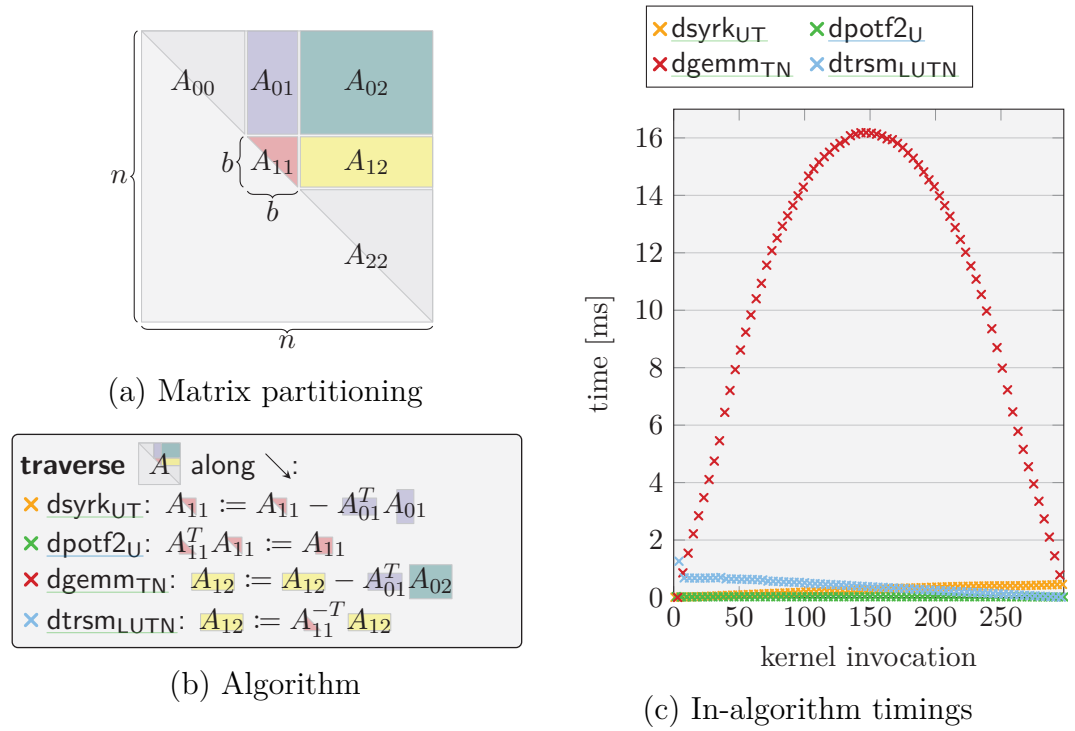


Figure 5.6: LAPACK’s blocked algorithm for the upper-triangular Cholesky decomposition dpotrf_{U} and in-algorithm timings.

($n = 2400$, $b = 32$, HARPERTOWN E5450, 1 thread, OPENBLAS, 100 repetitions)

require detailed knowledge of the BLAS implementation and thus substantial changes in our models.

5.2 Application to Other Algorithms

After studying LAPACK’s QR decomposition in great depth, we now consider two other blocked LAPACK algorithms: the upper-triangular Cholesky decomposition dpotrf_{U} (Section 5.2.1) and the inversion of a lower-triangular matrix $\text{dtrtri}_{\text{LN}}$ (Section 5.2.2).

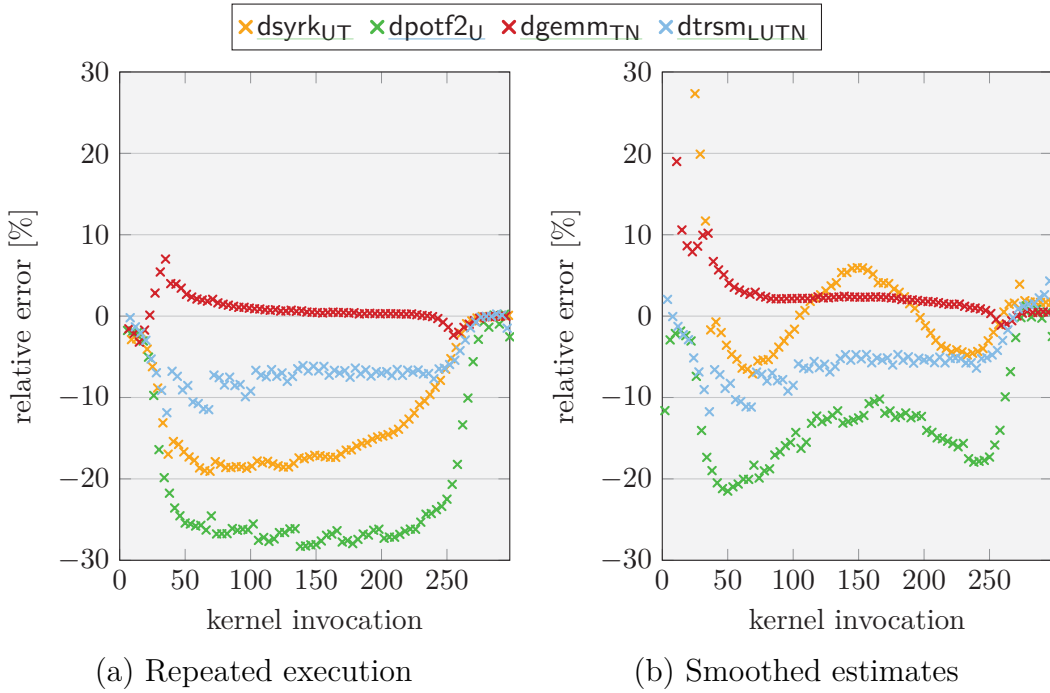


Figure 5.7: Error of our final estimates with respect to in-algorithm timings for the Cholesky decomposition `dpotrfU`.

($n = 2400$, $b = 32$, HARPertown E5450, 1 thread, OPENBLAS, median of 100 repetitions)

5.2.1 Cholesky Decomposition: `dpotrfU`

First, we consider LAPACK’s upper triangular Cholesky decomposition `dpotrfU`

$$\begin{matrix} \diagup & & \diagdown \end{matrix} U^T \begin{matrix} \diagdown & & \diagup \end{matrix} U := \begin{matrix} \boxed{A} \end{matrix}$$

of a symmetric positive definite $\boxed{A} \in \mathbb{R}^{n \times n}$ in upper triangular storage. [Figure 5.6](#) presents the blocked algorithm employed in this routine, which is the transpose of `dpotrf`’s algorithm for the lower-triangular case ([Figure 1.1c](#) on [Page 5](#)). As the algorithm traverses \boxed{A} , both the size and shape of $\boxed{A_{02}}$ (the largest operand) change noticeably: It starts as row panel, then grows to a square matrix and finally shrinks to a column panel. $\boxed{A_{02}}$ ’s size determines the workload performed by the algorithm’s large `dgemmTN` (\times), which is reflected in the in-algorithm timings in [Figure 5.6c](#).

In our experiments, we execute `dpotrfU` on a HARPertown E5450 with single-threaded OPENBLAS, $A \in \mathbb{R}^{2400 \times 2400}$,⁹ and block size $b = 32$. Figure 5.7 presents the relative performance difference with respect to in-algorithm timings for both repeated execution timings and our final estimates. Our estimates yield improvements for the `dsyrkUT` (✕) and `dpotf2U` (✕) involving large matrices in the middle of A 's traversal. In the beginning of the traversal, the estimates are generally too pessimistic because some matrices are (partially) brought into cache by prefetching, which is not accounted for in our estimates. On average the relative error is reduced from 11.11 % to 7.87 %, i.e., by a factor of 1.41.

However, note that the improvement is only visible in the averaged per-kernel relative error: Since the runtime of large `dgemmTN` (✕) is overestimated, the accumulated runtime estimate for the entire algorithm actually becomes less accurate.

5.2.2 Inversion of a Triangular Matrix: `dtrtriLN`

We now take a closer look at LAPACK's inversion of a lower-triangular matrix `dtrtriLN`

$$\triangle A := \triangle A^{-1}$$

with $A \in \mathbb{R}^{n \times n}$, whose blocked algorithm is presented in Figure 5.8. In contrast to the previous operations, this algorithm traverses $A \swarrow$ from the bottom-right to the top-left, thereby operating on sub-matrices of increasing size. Figure 5.8c shows the in-algorithm timings for the algorithm, which are dominated by `dtrmmLLNN` (✕).

We execute `dtrtriLN` on a HARPertown E5450 with single-threaded OPENBLAS, $A \in \mathbb{R}^{2400 \times 2400}$, and block size $b = 32$. Figure 5.9 compares the performance measurements from repeated execution and our final estimates to in-algorithm timings: The improvements of our estimates are most significant in `dtrmmLLNN` (✕) (which performs the most computation) and `dtrti2LN` (✕); the

⁹ For $n = 2400$, the upper-triangular portion of A takes up about 12 MiB—twice the size of the L2 cache.

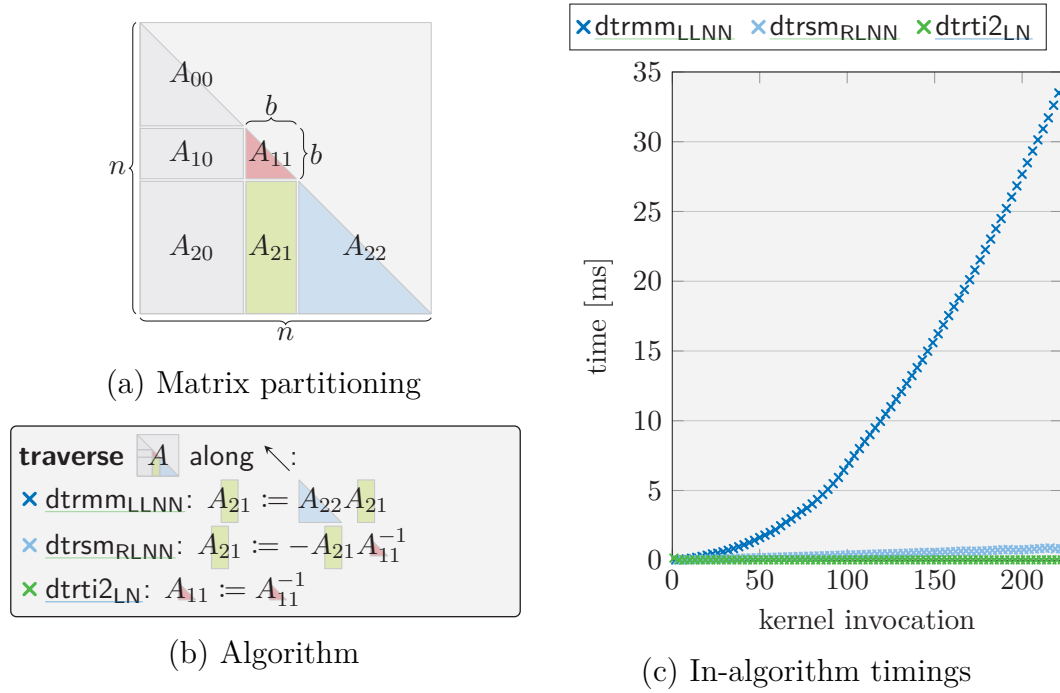


Figure 5.8: LAPACK’s blocked algorithm for the inversion of a lower-triangular matrix `dtrtriLN` and in-algorithms timings.

($n = 2400$, $b = 32$, HARPertown E5450, 1 thread, OPENBLAS, 100 repetitions)

error is reduced from an average of 6.70 % to 3.37 %—a total improvement of $1.99\times$.

5.2.3 Summary

We have seen that, on a HARPertown E5450 the accuracy of our runtime estimates for kernels within blocked algorithms is increased by taking the state of the L2 cache throughout the algorithm execution into consideration. For different algorithms, problem sizes, block sizes, BLAS implementations, and thread counts, we have seen improvements between $1.15\times$ (with all 4 cores) and $2.99\times$.

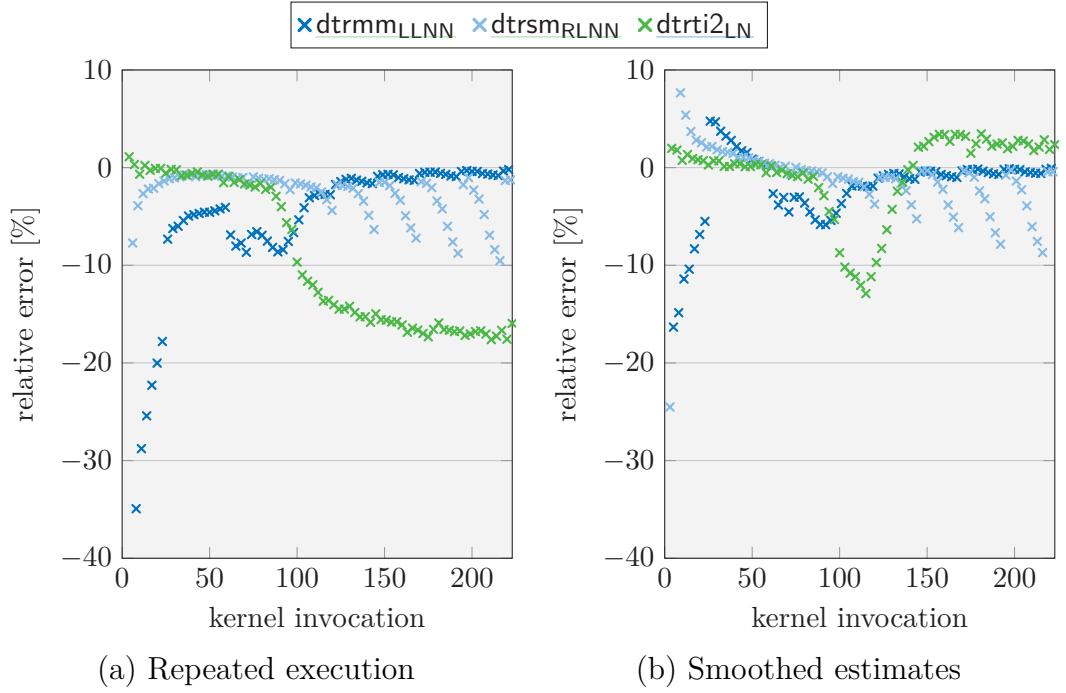


Figure 5.9: Error of our final estimates with respect to in-algorithm timings for the inversion of a lower-triangular matrix `dtrtriLN`.

($n = 2400$, $b = 32$, HARPERTOWN [E5450](#), 1 thread, OPENBLAS, median of 100 repetitions)

5.3 Feasibility on Modern Hardware

The analysis and cache model in the previous two sections focused on a HARPERTOWN [E5450](#)—a fairly old processor released in 2007. In this section, we study how well the same approach is applicable to more recent processors, namely a SANDY BRIDGE-EP [E5-2670](#) and a HASWELL-EP [E5-2680 v3](#).

The study reveals that on these processors it is especially challenging to establishing in- and out-of-cache timings as lower and upper bounds for the in-algorithm timings ([Section 5.3.1](#)). We present evidence that, while we can indeed estimate the in-algorithm timings, this is only possible by replicating the execution context within the algorithms, which is infeasible in the context of algorithm-independent performance models ([Section 5.3.2](#)).

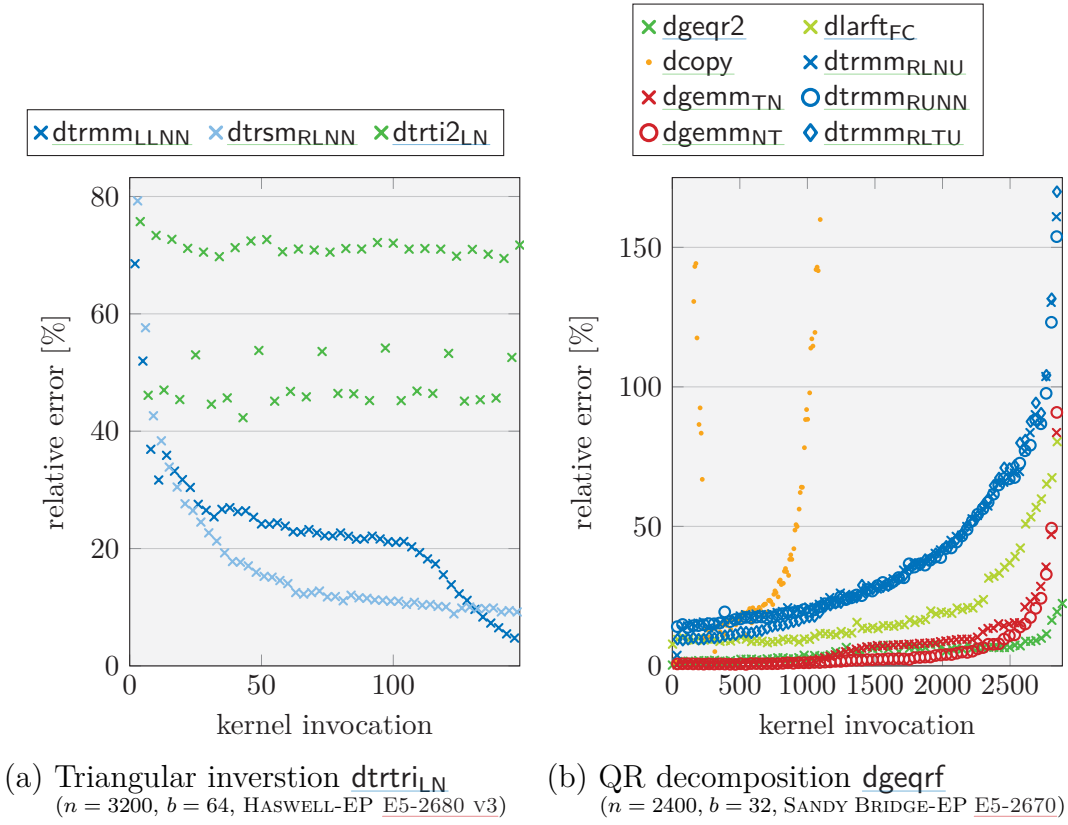


Figure 5.10: Error of out-of-cache timings with respect to in-algorithm timings for $\text{dtrtri}_{\text{LN}}$ and dgetrf .

(1 thread, OPENBLAS, median of 100 repetitions)

5.3.1 In- and Out-of-Cache Timings

Out-of-core timings are hardware independent, and just as on the HARPER-TOWN serve as an upper bound on the SANDY BRIDGE and HASWELL. This is illustrated in Figure 5.10 for the inversion of a lower-triangular matrix $A \in \mathbb{R}^{3200 \times 3200}$ with $\text{dtrtri}_{\text{LN}}$ (Figure 5.8) and block size $b = 64$ on the HASWELL, and the QR decomposition of $A \in \mathbb{R}^{2400 \times 2400}$ with dgeqrf (Figure 4.9) and $b = 32$ on the SANDY BRIDGE—the chosen matrices comprise around 40 MiB and thus exceed the SANDY BRIDGE’s and HASWELL’s last-level cache (L3) of, respectively, 20.30 MiB. The out-of-cache timings indeed consistently overestimate the in-algorithm timings—by up to 347 % for the last

call to `dtrmmRUNN` (○) in the QR decomposition `dgeqrf` on the SANDY BRIDGE (Figure 5.10b is clipped at 175 %). As such, these measurements serve well as an upper bound on the in-algorithm timings.

For the same scenarios Figure 5.11 presents the error of our previous in-cache setup with respect to the in-algorithm timings: While we expect the our setup to yield faster kernel executions than the in-algorithm timings, on the SANDY BRIDGE-EP E5-2670 (with TURBO BOOST disabled) the in-cache timings are still up to 0.51 % slower than the in-algorithm timings (not accounting for the small unblocked `dgeqr2`); on the HASWELL-EP E5-2680 v3 (with TURBO BOOST enabled), the relative errors for `dtrtriLN` and `dgeqrf` reach, respectively, 1.67 % and 3.44 %.

Further investigation reveals that the processor’s INTEL TURBO BOOST is a source of complication for our measurements: As Figure 5.12 shows, enabling TURBO BOOST on the SANDY BRIDGE-EP E5-2670 leads to overestimations of the `dtrtriLN`’s and `dgeqrf`’s most compute-intensive operations (i.e., the `dtrmmLLNN` (×) and the two `dgemms` (×, ○)), by up to, respectively, 3.20 % and 2.79 %.

While TURBO BOOST increases the overestimation of individual kernels, this phenomenon’s origin lies in the processor’s cache hierarchy: Within an algorithm, each kernel is invoked with a distinct cache precondition, i.e., with only portions of its operands in the processor’s caches. Since our algorithm-independent measurements do clearly not match such preconditions, we attempted to construct conditions in which the kernel executes at its absolute peak performance with different cache setups:

- First, we used simple repeated execution of the kernel without any modification of the cache in between as before.
- Next, we accessed the kernel operands in various orders prior to the invocation. E.g., for $\bar{C} := \boxed{A} \boxed{B} + \bar{C}$ (`dgemmNN`), we attempted all permutations of access orders, such as $\boxed{A} - \boxed{B} - \bar{C}$ and $\bar{C} - \boxed{A} - \boxed{B}$.
- Finally, we refined the access granularity and attempted to bring operands

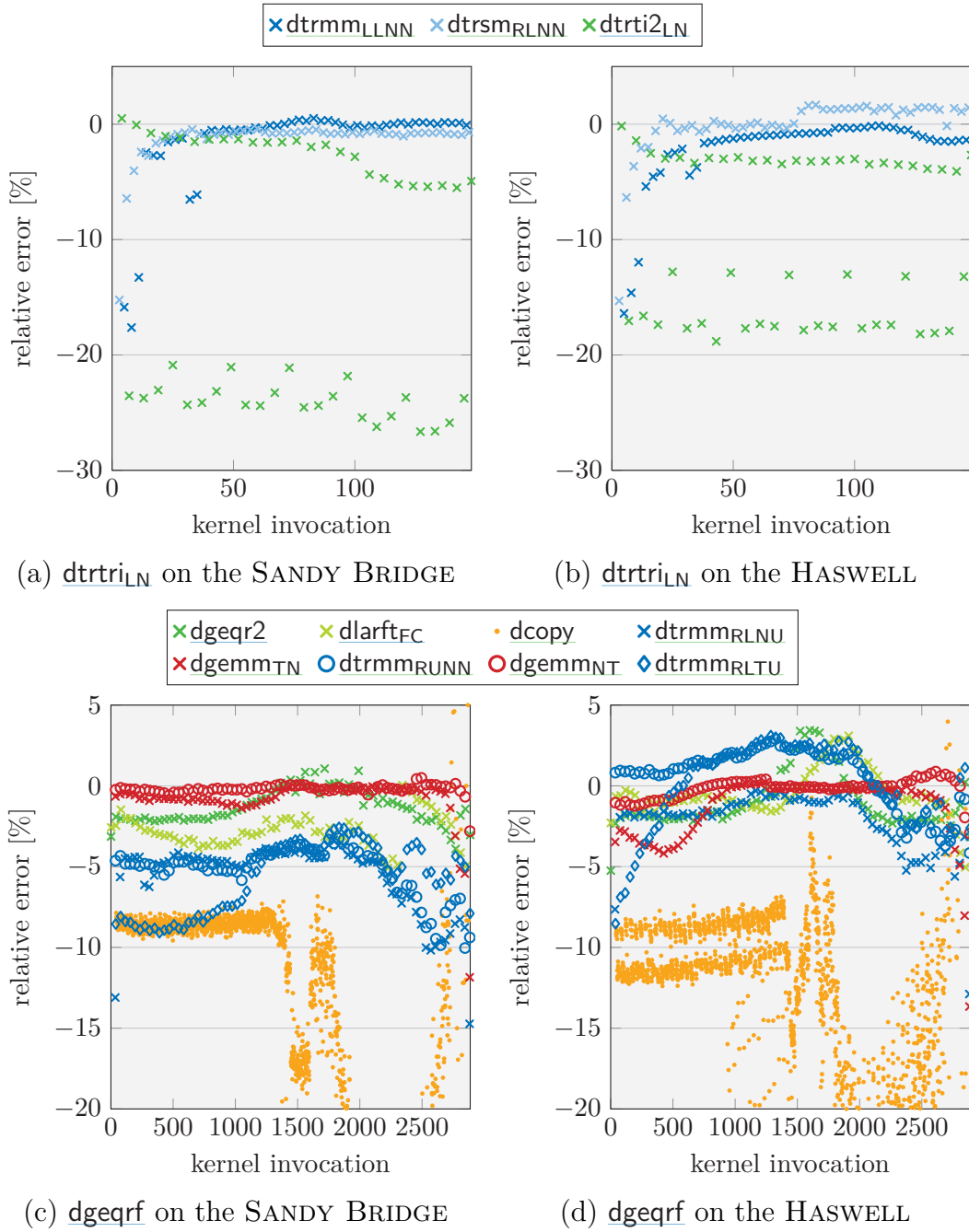


Figure 5.11: Error for attempted in-cache timings with respect to in-algorithm timings for $\text{dtrtri}_{\text{LN}}$ and dgetrf .

($\text{dtrtri}_{\text{LN}}$: $n = 3200$, $b = 64$; dgeqrf : $n = 2400$, $b = 32$; SANDY BRIDGE-EP [E5-2670](#) and HASWELL-EP [E5-2680 v3](#), 1 thread, OPENBLAS, 100 repetitions)

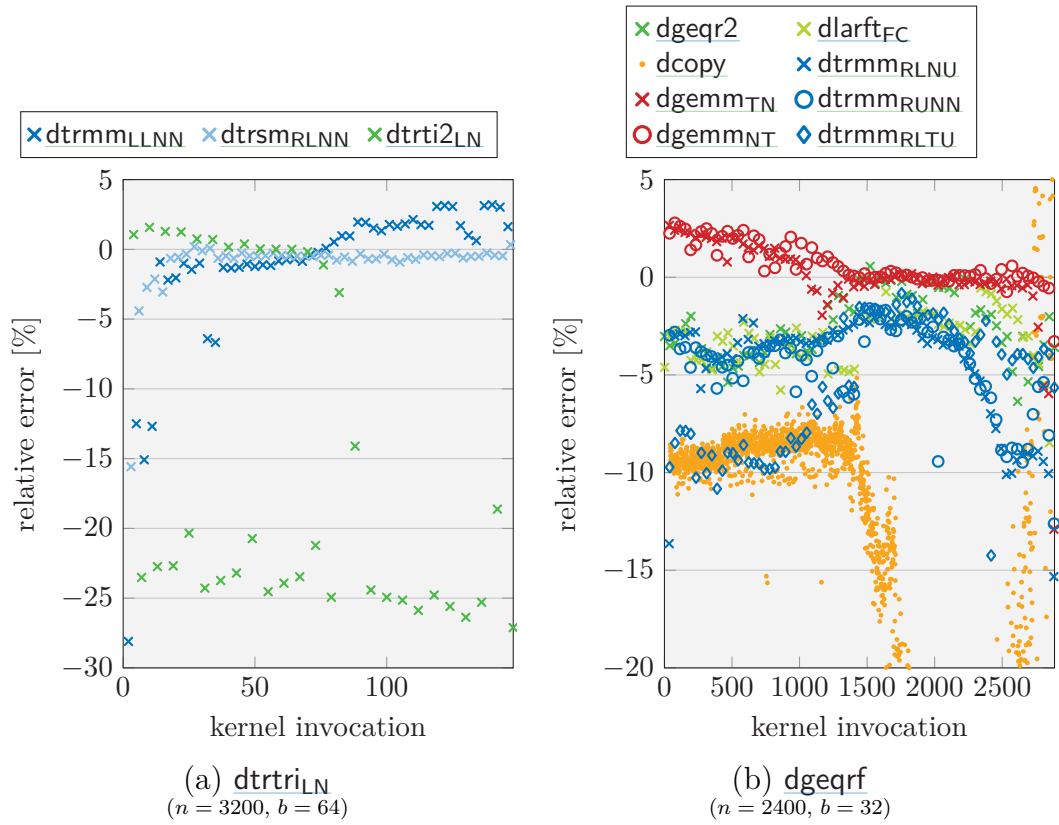


Figure 5.12: Error for attempted in-cache timings with respect to in-algorithm timings on a SANDY BRIDGE-EP [E5-2670](#) with TURBO BOOST enabled.

(1 thread, OPENBLAS, median of 100 repetitions)

into cache not as a whole but only partially: For a kernel with one operand larger than the cache and the other operand(s) only a fraction of that size (e.g., the `dgemmTN` (x) in `dgeqrf`: $C := A B + C$ where B and C are of width b and close to the problem size n in height), we bring the entire small operand(s) into cache but only portions of the large one.

Figure 5.13 presents which operand portions we chose to load into the cache. These choices are based on the assumption that any kernel implementation likely traverses the input matrix somehow from the

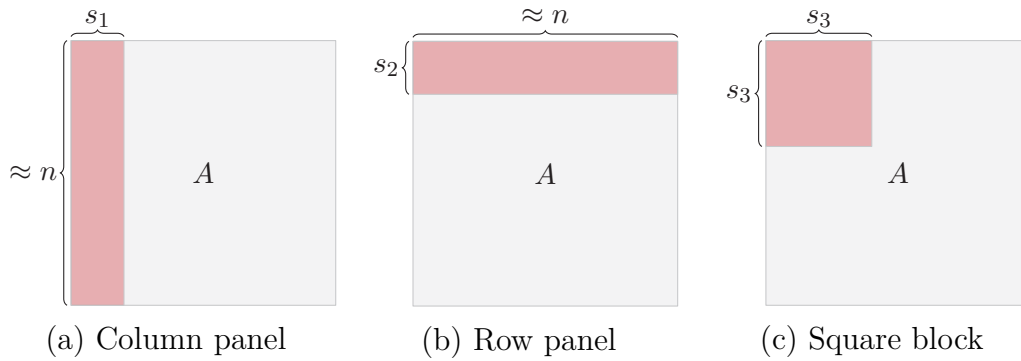


Figure 5.13: Basic operand regions accessed for attempted in-cache setups.

top-left \searrow to the bottom-right.¹⁰ Therefore, we bring a column panel of the operand, a row panel, a square block, or any combination of these into the processor's caches. While doing so, we varied the sizes s_1 , s_2 , and s_3 of the accessed operand portions.

While in some scenarios changing the in-cache setup for kernel invocations reduced the runtime overestimation, the effects were not consistent across different algorithms, kernels, processors, and BLAS implementations. Altogether, it was not possible to determine general, algorithm-independent in-cache setups that yield a clear lower bound on the in-algorithm timings.

5.3.2 Algorithm-Aware Timings

Since our above attempts at algorithm-independent in-cache timings did not yield the required lower bound on in-algorithm timings, the only alternative is to tailor the timing setups to individual algorithms. We might for instance setup each kernel timing with several preceding kernel invocations from within the algorithms. Such obtained *algorithm-aware timings* yield accurate estimates for the in-algorithm timings, and rid us of the need for combining in- and out-of-cache estimates.

¹⁰ Exceptions are, e.g., `dtrsmRLNN` ($B := BA^{-1}$) and `dtrsmLUNN` ($B := A^{-1}B$), which must traverse the triangular A from the bottom-right to the top-left—in these cases the accessed matrix portions are mirrored accordingly.

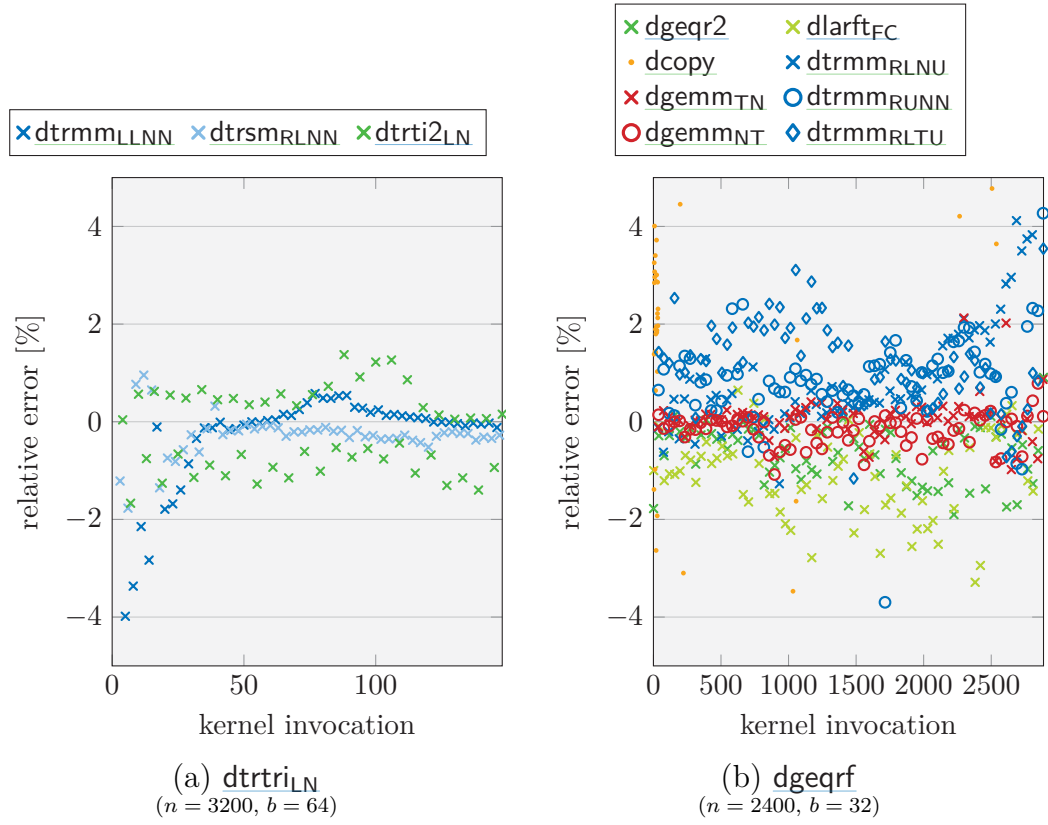


Figure 5.14: Error for algorithm-aware timings with respect to in-algorithm timings.

(SANDY BRIDGE-EP E5-2670, 1 thread, OPENBLAS, median of 10 repetitions)

Example 5.1: Algorithm-aware timings

Figure 5.14 presents the accuracy of algorithm-aware timings as estimates for in-algorithm timings for the inversion of a lower-triangular matrix (`dtrtriLN`) and the QR decomposition (`dgeqrf`) on a SANDY BRIDGE-EP E5-2670 (with TURBO BOOST enabled) using single-threaded OPENBLAS. The algorithm-aware timings were created by preceding each measured kernel invocation with the calls from the corresponding blocked algorithm that were executed since that kernel's last invocation.

Figure 5.14a shows that for `dtrtriLN` the algorithm-aware timings are with few exceptions within 1% of the in-algorithm timings with an average

absolute relative error (ARE) of 0.54 %. As seen in [Figure 5.14a](#), for the `dgetrf` the relative error is overall larger yet similarly spread around 0 % with an average ARE of 0.84 %.

While this approach yields accurate estimates, when the kernel invocations for each algorithm execution are timed separately and each measurement is preceded with a setup of one or more kernels, the timing procedure takes effectively longer than executing and measuring the target algorithm repeatedly. As a result, this method is at the same time highly accurate and impractical, which is why we do not further pursue it.

5.4 Summary

This chapter investigated the possibility of improving the accuracy of performance predictions for blocked algorithms by accounting for caching effects. On a HARPertown [E5450](#), we were able to establish algorithm-independent in- and out-of-cache kernel timings as, respectively, lower and upper bounds on in-algorithm timings. By tracking which (portions of) operands are in-cache throughout an algorithm’s execution, we were able to combine these timings into more accurate runtime estimates than repeated execution timings.

This approach did not work equally well on more recent processors: On a SANDY BRIDGE-EP [E5-2670](#) and a HASWELL-EP [E5-2680 v3](#), we concluded that constructing a cache precondition to yield lower bounds on the in-algorithm timings was only attainable with algorithm-aware measurements. Since such measurements are not only incompatible with our modeling approach but are also less efficient than straightforward measurements of the target algorithm, we conclude that no efficient strategy to improve the accuracy for our model-based predictions on modern hardware was found.

6 Micro-Benchmarks for Tensor Contractions

This chapter addresses the problem of accurately predicting the performance of BLAS-based algorithms for tensor contractions. Since in practice, such contractions are commonly used with skewed dimensions, the previously developed performance models are unfortunately unsuitable: For small matrices, the performance of BLAS kernels is quite irregular, and our models are less accurate. Furthermore, for small and skewed operations, caching effects can play an immense role. Hence, for tensor contractions, we follow a different approach, and exploit that contraction algorithms are based on repeated executions of a single kernel operation with fixed operand sizes: We use cache-aware micro-benchmarks that perform only a fraction of these executions in a replica of the algorithm’s executions environment, and extrapolate their runtime to obtain performance predictions.

In the following, [Section 6.1](#) discusses the systematic generation of BLAS-based algorithms for tensor contractions, [Section 6.2](#) introduces our micro-benchmarks and performance predictions, and [Section 6.3](#) presents experimental results for a range of contractions.

Publication

The work presented in this chapter is based on research previously published in:

- [6] Elmar Peise, Diego Fabregat-Traver, and Paolo Bientinesi. “On the Performance Prediction of BLAS-based Tensor Contractions”. In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*:

5th International Workshop, PMBS 2014. Volume 8966. Lecture Notes in Computer Science. Springer International Publishing, Apr. 2015, pages 193–212. DOI: [10.1007/978-3-319-17248-4_10](https://doi.org/10.1007/978-3-319-17248-4_10).

In this collaboration, Diego Fabregat-Traver implemented the algorithm generation presented in [Section 6.1](#), while this author developed the performance predictions detailed in [Sections 6.2](#) and [6.3](#).

6.1 Algorithm Generation

Following a brief overview of tensor notation and storage, this section explains the systematical generation of a family of BLAS-based algorithms for a tensor contraction. For a detailed discussion of the topic, see [\[37\]](#).

We express tensor contractions in Einstein notation:¹ E.g., a matrix-matrix product $C := A B$ is denoted by $C_{ab} := A_{ai}B_{ib}$, meaning the entries of C are computed as $C[a,b] := \sum_i A[a,i]B[i,b]$. The *indices* that appear in both tensors A and B —the summation indices i, j, \dots —are called *contracted*, while those that only appear in either A or B (and thus in C)— a, b, c, \dots —are called *uncontracted* or *free*. Without loss of generality, we assume that tensors are stored as FORTRAN-style contiguous multidimensional double-precision arrays: Vectors (1D tensors) are stored contiguously, matrices (2D tensors) are stored as sequences of column vectors, 3D tensors (visualized as cubes) are stored as sequences of matrices (planes of the cube), and so on.

Aware of the extreme level of efficiency inherent to optimized BLAS implementations, our approach for computing a contraction consists in reducing it to a sequence of calls to one BLAS kernel. Since BLAS operates on scalars, vectors, and matrices (zero-, one- and, two-dimensional objects), tensors must be expressed in terms of a collection of such objects. To this end, we introduce the concept of *slicing*: With the help of MATLAB’s “:” notation,² slicing a

¹ For the sake of simplicity and without any loss of generality, we ignore any distinction between covariant and contravariant vectors; this means we treat any index as a subscript.

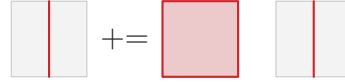
² In MATLAB the index “:” in a tensor refers to all elements along that dimension, e.g., $C[:,b]$ is the b -th column of C .

d -dimensional operand $\mathcal{O}_p \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$ along the i -th index (or dimension) means creating the n_i $(d-1)$ -dimensional slices $\mathcal{O}_p[\underbrace{:, \dots, :}_{i-1}, k, \underbrace{:, \dots, :}_{d-i}]$, where $k = 1, \dots, n_i$.

Example 6.1: Contraction algorithm for `dgemmNN`

Consider the matrix-matrix product $C_{ab} := A_{ai}B_{ib}$ (`dgemmNN`). Slicing the matrix B along dimension b reduces it to a collection of column vectors $B[:,b]$; accordingly, the matrix-matrix product is reduced to a sequence of matrix-vector operations:³

```
for b = 1:b
  dgemvN:  $C[:,b] += A[:,:]B[:,b]$ 
```

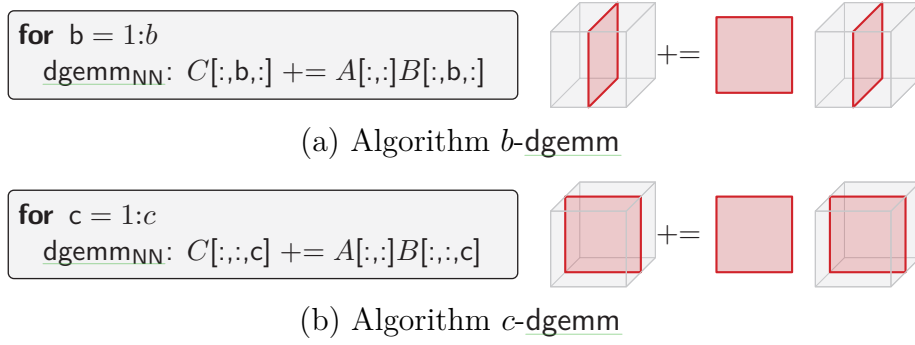


Depending on the slicing choices, a tensor contraction is reduced to a number of *nested loops* with one of the following five *kernels* at the innermost loop's body:

- BLAS Level 1:
 - `ddot`: vector-vector inner product $\alpha := x^T y$,
 - `daxpy`: vector scaling and addition $y += \alpha x$,
- BLAS Level 2:
 - `dgemv`: matrix-vector product $y += A x$,
 - `dger`: vector-vector outer product $A += x y^T$, and
- BLAS Level 3:
 - `dgemm`: matrix-matrix product $C += A B$.

Notice that to comply with the BLAS interface, the elements in one of the two dimensions of a matrix must be contiguous. Therefore, algorithms that rely on `dgemv`, `dger`, or `dgemm` as their computational kernel may require a *temporary*

³ The pictogram next to the algorithm visualizes the slicing of the tensors that originates the algorithm's sequence of `dgemvN`s. The red shapes represent the operands of the BLAS kernel.

Figure 6.1: Contraction algorithms for $C_{abc} := A_{ai}B_{ibc}$ based on `dgemm`.

Since `dgemm` involves one free index in each of its operands A and B , and one contracted index (common to both A and B), in order to reduce any contraction to a sequence of `dgemm` calls, one must slice all but one free index of both A and B , and all but one contracted index. For the above contraction, this is achieved by slicing either dimension b or c , resulting in the two algorithms *b-dgemm* and *c-dgemm*⁴ shown in Figure 6.1.

Since for a given contraction, there is no obvious a-priori choice of kernel and slicings to maximize performance, we consider all possible combinations. Moreover, we consider all possible *permutations of the loops*, because, due to caching effects, each permutation yields a different performance.

Example 6.3: Other algorithms for $C_{abc} := A_{ai}B_{ibc}$

For the contraction $C_{abc} := A_{ai}B_{ibc}$ from Example 6.2, the right part of Table 6.1 lists examples of algorithm generations for all five suitable BLAS kernels: A selection of the contraction's free and contracted indices are mapped to each kernel's indices (column "kernel indices"), the remaining indices can be sliced in any (loop-)order (column "sliced indices") with each order resulting in a different algorithm. The resulting algorithms are presented in full in Figures 6.1 to 6.3.

We developed a small *algorithm and code generator* that produces all algo-

⁴ The name of each algorithm stems from the dimensions its **for**-loops index and its BLAS kernel. If the algorithm uses `copy`-kernels, they are indicated by apostrophes '.

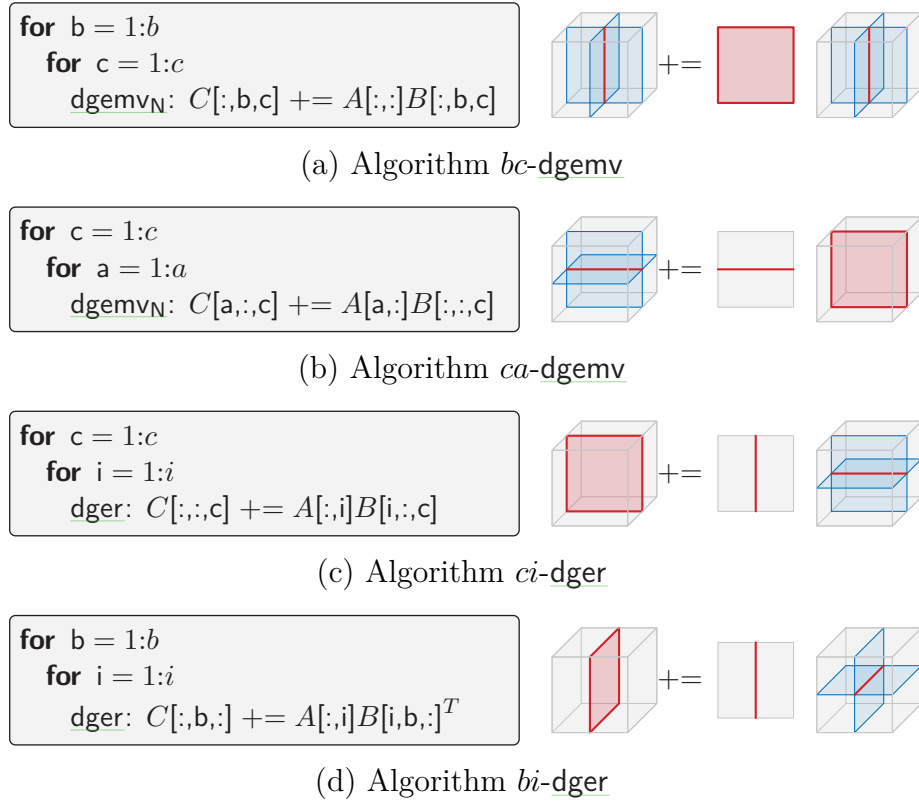


Figure 6.2: Sample of contraction algorithms for $C_{abc} := A_{ai}B_{ibc}$ based on BLAS Level 2. All slicings are visualized in **blue**; only the kernel operands (the intersections) are in **red**.

gorithms derived in this manner, and constructs corresponding *C-implementation*, as well as *abstract syntax trees* (ASTs) representing their loop-based structure. These ASTs form the starting point for the micro-benchmarks introduced in the following section.

6.2 Runtime Prediction

This section describes development accurate runtime and performance performance for the previously introduced type of BLAS-based algorithms for tensor contractions. Taking advantage of these algorithms loop-based structure, we

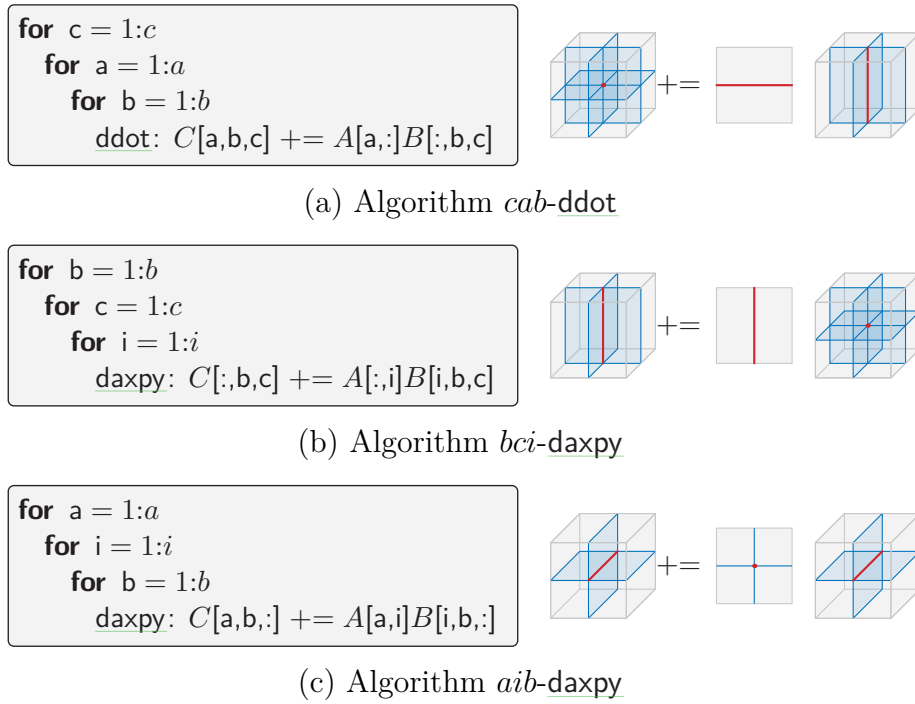


Figure 6.3: Sample of contractions algorithms for $C_{abc} := A_{ai}B_{ibc}$ based on BLAS Level 1.

aim at estimating each algorithm's runtime through *micro-benchmarks* of its BLAS kernel, i.e., with no direct execution of the algorithm itself. In order to obtain reliable estimates, these micro-benchmarks need to be executed in a setup that mirrors the computing environment (most importantly the cache) within the contraction algorithm as closely as possible. In the following, we incrementally go through the steps required to build meaningful “replicas” of the computing environment.

6.2.1 Example Contraction: $C_{abc} := A_{ai}B_{ibc}$

Throughout this section, we track the improvement of various changes to our predictions by considering the *contraction* $C_{abc} := A_{ai}B_{ibc}$ with $A \in \mathbb{R}^{a \times i}$ and

6 Micro-Benchmarks for Tensor Contractions

$B \in \mathbb{R}^{i \times b \times c}$ and sizes $i = 8$ and $a = b = c = 8, \dots, 1000$:

$$C := A_i B$$

This scenario is deliberately challenging due to the small tensor dimension i , for which BLAS kernels are generally not optimized.

For the selected contraction, our generator produces 36 algorithms, some of which are shown in [Figures 6.1 to 6.3](#):

- 6 ddot-based,
- 18 daxpy-based,
- 6 dgemv-based: bc-dgemv (---), cb-dgemv (---), ac-dgemv (---), ca-dgemv (---), ab-dgemv (---), ba-dgemv (---),
- 4 dger-based: ci-dger (---), ic-dger (---), bi-dger (---), ib-dger (---), and
- 2 dgemm-based: c-dgemm (—), b-dgemm (—).

However to avoid overloaded performance plots, this section only considers the algorithms based on BLAS Level 2 and 3, i.e., with the kernels dgemv, dger, and dgemm.

[Figure 6.4](#) displays the measured performance of these algorithms on a HARPertown [E5450](#) using single-threaded OPENBLAS. Our goal in the following sections is to accurately predict this performance without executing the algorithms. Although it is evident that only two of the algorithms—the dgemm-based c-dgemm (—) and b-dgemm (—)—are competitive⁵ we aim to accurately predict all algorithms to develop and demonstrate the broad applicability of our methodology.

⁵ Due to the extremely small dimension $i = 8$, they achieve less than half of the HARPertown's theoretical peak performance of 12 GFLOPs/s.

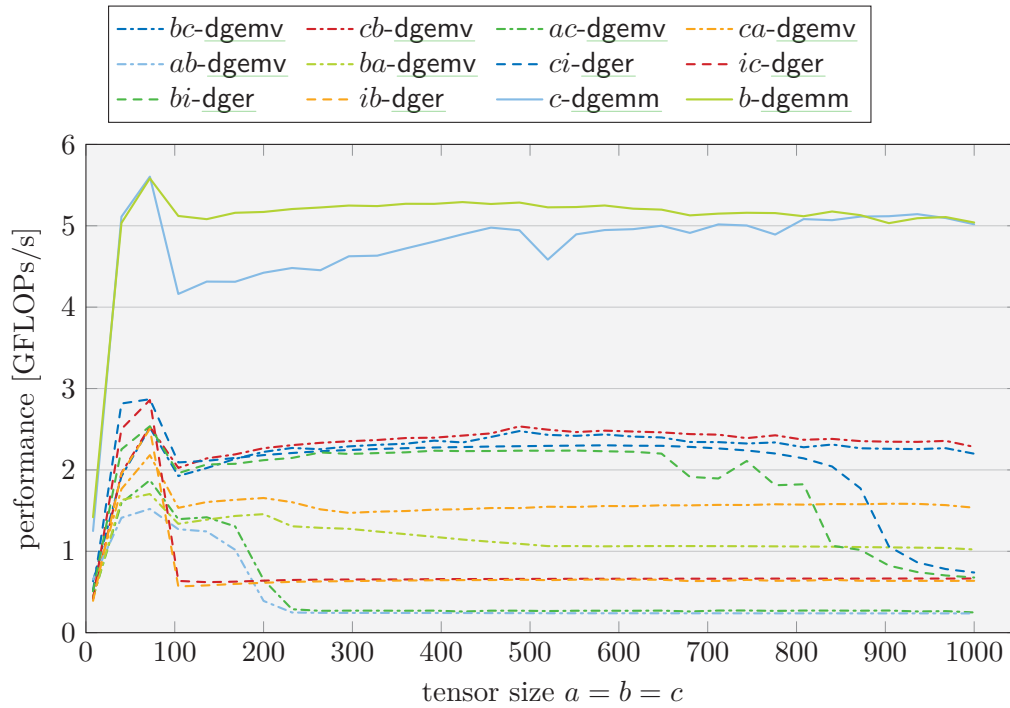


Figure 6.4: Performance measurements of algorithms for $C_{abc} := A_{ai}B_{ibc}$ based on BLAS Level 2 and 3.

($i = 8$, HARPERTOWN E5450, 1 thread, OPENBLAS, median of 10 repetitions)

6.2.2 Repeated Execution

The first, most intuitive, attempt to predict the performance of an algorithm through a micro-benchmark relies on the *repeated measurement* of its BLAS kernel's performance *in isolation*. We implemented this approach by executing each kernel ten times in the SAMPLER, and extracting the median runtime; the corresponding estimate is then obtained by multiplying this median by the number of kernel invocations within the algorithm. In our example, this boils down to multiplying the kernel runtime with the product of all loop lengths.

The performance predicted by this first, rough approach is shown in Figure 6.5a. By comparing this figure with the reference repeated in Figure 6.5b, it becomes apparent that while the two fastest algorithms are already correctly identified, the performance of almost all algorithms is consistently

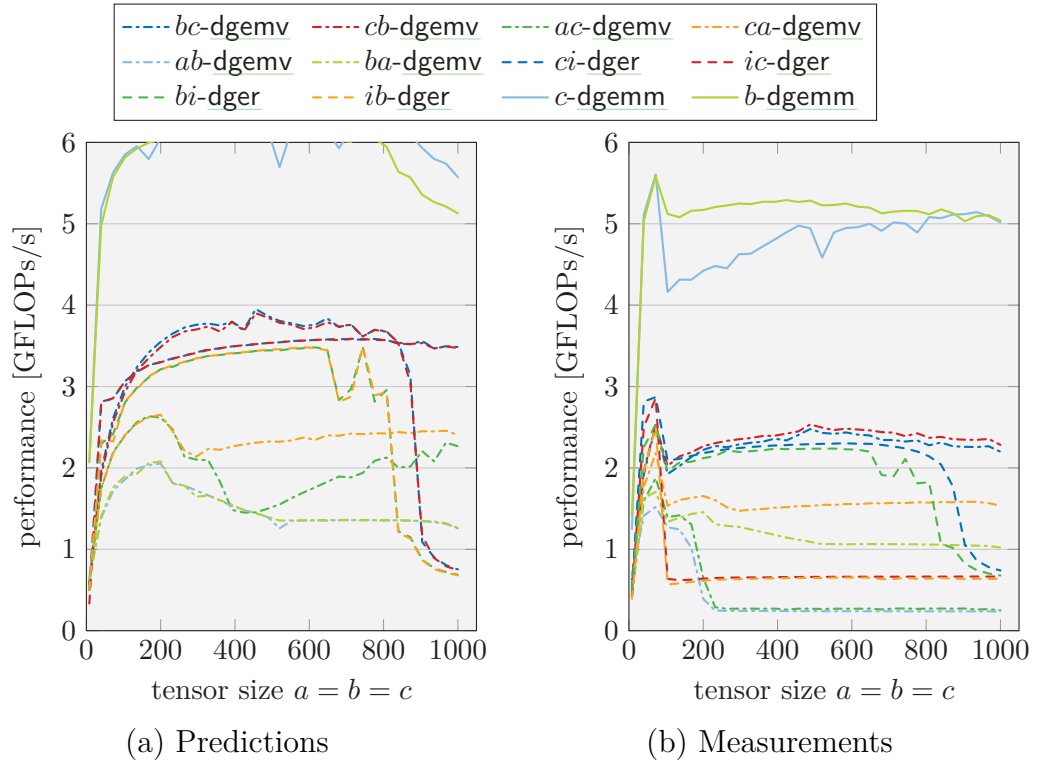


Figure 6.5: Performance predictions for $C_{abc} := A_{ai}B_{ibc}$ based on repeated execution.

($i = 8$, HARPETOWN E5450, 1 thread, OPENBLAS, median of 10 repetitions)

overestimated—the average absolute error with respect to the measured performance is 154%. In other words, when executed as part of the algorithms, the BLAS kernels take longer to complete than in the isolated micro-benchmarks. The reason for this discrepancy is that the micro-benchmarks invoke the kernels repeatedly with the same operands, i.e., they operate on cached (“warm”) data. Within an algorithm, by contrast, at least one operand varies from one invocation to the next, i.e., the kernel operates at least partially on “cold” data.

6.2.3 Operand Access Distance

In order to improve our predictions’ accuracy, we attempt to replicate the state of the cache within an algorithm prior to the kernel invocation (the “cache precondition”) within our micro-benchmarks. For this purpose, we assume a fully associative Least Recently Used (LRU) cache replacement policy,⁶ and, in first instance, consider the case where all loops surrounding the kernel are somewhere in the middle of their traversal (i.e., not in their first iteration); this second assumption will be lifted later.

To determine if an operand is cached and to place it in the correct cache level for the micro-benchmark, we determine how much data was used in any operations since its last access, referred to as its *access distance*. Once this access distance is known for all kernel operands, we create an artificial sequence of memory accesses to reconstruct the cache precondition. Using this cache setup, our micro-benchmark’s measurement of the kernel closely resembles the actual execution of the algorithm. As before, the median runtime of ten micro-benchmark repetitions multiplied with the number of kernel invocations yields the algorithm’s runtime prediction.

We now describe how to obtain the access distance for each operand. While the presented method allows for any combinations of loops and multiple kernels (e.g., a BLAS kernel and a `copy` kernel), for the sake of clarity, we limit the discussion to abstract syntax trees (ASTs) that consist of only a one or more nested loops with a single BLAS kernel at their core.

To determine the access distance for an operand \mathcal{Op} , we examine an algorithm’s AST (see [Section 6.1](#)) starting at the kernel, and traverse it backwards until the previous access to \mathcal{Op} (or the AST’s root) is found. While doing so, we collect the operands of all encountered kernels in an initially empty set M , whose total data volume—the sum of the collected operands’ sizes—ultimately determines the access distance. Going up the AST, three different cases can be encountered.

⁶ Due to the regular storage format and memory access strides of dense linear algebra operations such as the considered tensor contractions, this simplifying assumption does not affect the reliability of the results.

1. **\mathcal{Op} does not vary across the surrounding loop.**

In this case \mathcal{Op} referred to the same operand in the previous iteration of the surrounding loop. The back-traversal therefore terminates, and the operands collected in M so far determine the access distance.

Example 6.4: Loop-independent operand

In algorithm *ca-dgemv* (---) the operand $B[:, :, c]$ does not depend on the surrounding loop's iterator a :

```

for c = 1:c
  for a = 1:a
    dgemvN: C[a, :, c] += A[a, :]B[:, :, c]
  
```



Hence, $M = \emptyset$ and $B[:, :, c]$'s access distance is 0.

2. **\mathcal{Op} varies across the surrounding loop.**

In this case \mathcal{Op} referred to a different operand in the previous iteration of the loop. As a result, it is safe to assume that at least all kernel operands throughout this loop's iterations were accessed since the last access to \mathcal{Op} . Hence, all operands are added to M and they are symbolically joined along the dimensions the loop iterates over.

Since a previous access to \mathcal{Op} was not yet detected, the traversal proceeds by going up one level in the AST and applying the method recursively: The surrounding loop now takes the role of the starting node and we look for a previous access to \mathcal{Op} joined across this loop.

Example 6.5: Loop-dependent operand

In algorithm *ca-dgemv* (---) the operand $A[a, :]$ depends on the surrounding loop's iterator a . The algorithm's kernel operates on $A[a, :]$, $B[:, :, c]$, and $C[a, :, c]$, which joint across the index a yields the collection

$$M = \{A[:, :], B[:, :, c], C[:, :, c]\} .$$

The backward-traversal of the AST continues and now looks for a previous access to $A[:, :]$ — $A[a, :]$ joint across a —in the second-innermost loop. Since this operand is independent of this loop's iterator c , case 1

above applies and $A[\mathbf{a},:]$'s access distance is computed from the set M above.

3. The parent node is the AST's root.

In this case, \mathcal{Op} is accessed only once (and for the first time). Since we do not know how the contraction is used (within a surrounding application), we can generally not make any assertions on the access distance. For the purpose of this study, in which we execute the contraction repeatedly to measure its performance, however, we assume that no other data was accessed since the last invocation of the contraction; hence, we compute the access distance from the collection M .

Example 6.6: No loops remaining

In algorithm *ca-dgemv* (---), the operand $C[\mathbf{a},:\mathbf{c}]$ depends on both of the surrounding loops' iterators \mathbf{a} and \mathbf{c} . Therefore, the back-traversal encounters case 2 above in both its first and second step, and joining the kernel's operands $A[\mathbf{a},:]$, $B[:,:\mathbf{c}]$, and $C[\mathbf{a},:\mathbf{c}]$ across first \mathbf{a} and then \mathbf{c} , yields

$$M = \{A[:,:], B[:,:\mathbf{c}], C[\mathbf{a},:\mathbf{c}]\} .$$

In the third step of the back-traversal, the outermost loop is already the starting point—the AST's root is reached. Assuming repeated executions of the entire contraction, $C[\mathbf{a},:\mathbf{c}]$'s access distance is computed from the set M above.

Based on access distance for each operand of an algorithm's kernel, we construct a micro-benchmark that *emulates the accesses* within the algorithm prior to the kernel's execution. This micro-benchmark consists of accesses to the kernel's operands interleaved with accesses to remote memory regions that *flush* portions of the *cache* corresponding to the access distances: First, we access the operand with the largest access distance, and then a remote region that accounts for the difference to the next smaller access distance; this is repeated until the operand with the smallest access distance is loaded followed by a remote access of this size. If the access distances to the first operand in

operand	size [doubles]	collection of operands M	access distance [doubles]
$B[:, :, c]$	3200	\emptyset	0
$A[a, :]$	8	$\{A[:, :], B[:, :, c], C[:, :, c]\}$	166 400
$C[a, :, c]$	400	$\{A[:, :], B[:, :, c], C[:, :, c]\}$	65 283 200

Table 6.2: Operand sizes and access distances in *ca-dgemv* for $C_{abc} := A_{ai}B_{ibc}$.
($a = b = c = 400$, $i = 8$, sizes in doubles)

this list is larger than $\frac{5}{4}$ times the cache size, the list is truncated to this limit at the front.

Example 6.7: Cache access emulation

For algorithm *ca-dgemv* (----), Table 6.2 summarizes the operands, their sizes, the corresponding collections M , and the implicated access distances for tensor sizes $a = b = c = 400$ and $i = 8$. From these distances, we get the following list of memory accesses as a setup for the *dgemv*_N-kernel, where the $[s]$ correspond to remote memory accesses of s doubles ($= 8s$ bytes):

$$C[a, :, c], [65\,116\,792], A[a, :], [163\,200], B[:, :, c] \quad .$$

Note that the remote accesses do not directly correspond to the access distances; instead, this distance is reached for each operand as the sum of the sizes of all accesses to its right in this list. (e.g., the access distances of $A[a, :]$ is reached as $163\,200$ doubles + $\text{sizeof}(B[:, :, c]) = 166\,400$ doubles).

The largest access distance of $65\,283\,200$ doubles is considerably larger than $983\,040$ doubles ($= \frac{5}{4} \times 6 \text{ MiB} = \frac{5}{4} \times \text{L2 cache size}$). Hence, the list is cut at this size, yielding the final setup for this algorithm's micro-benchmark:

$$[816\,632], A[a, :], [163\,200], B[:, :, c] \quad .$$

The thus obtained benchmark, consisting of the setup followed by the kernel invocation, is as before executed ten times, and the resulting median runtime is used to compute our second runtime and performance predictions.

Figure 6.6a presents our new performance predictions: Compared to our

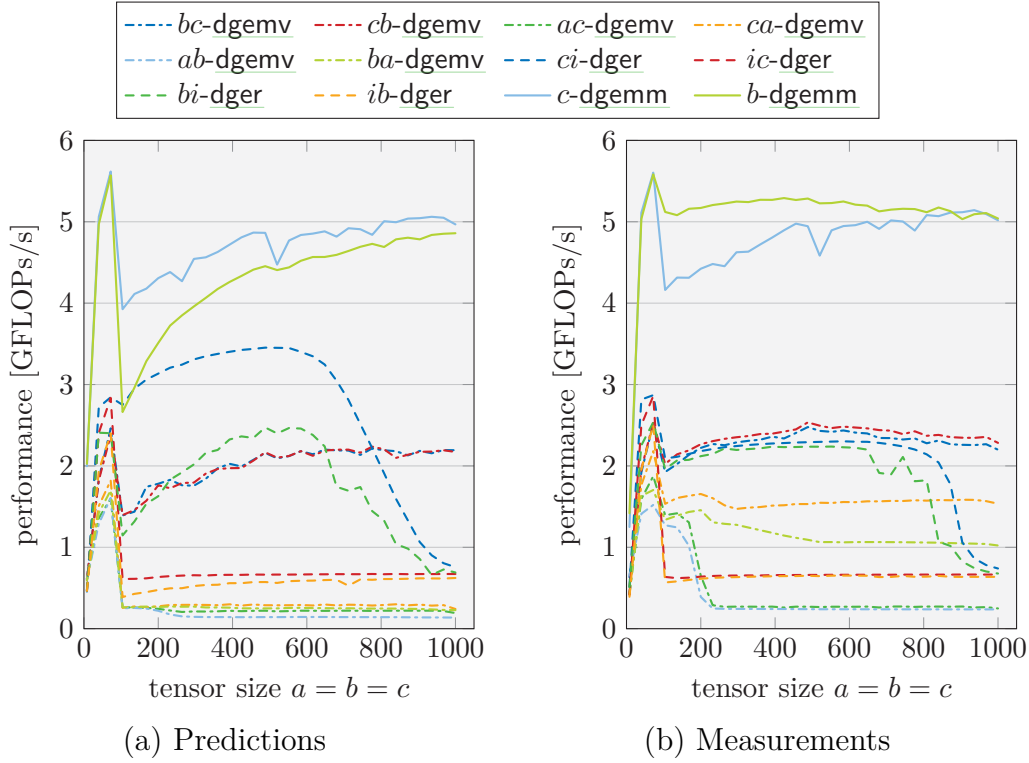


Figure 6.6: Performance predictions for $C_{abc} := A_{ai}B_{ibc}$ with cache emulation based on access distances.

($i = 8$, HARPertown E5450, 1 thread, OPENBLAS, median of 10 repetitions)

initial estimates (Figure 6.5a), these predictions are already much closer to the measured performance (Figure 6.6b); the average error is reduced to 26.3%. For several algorithms (such as *ic-dger* (---)), the error is already within a few percent; for many others instead, the predictions are still off. In particular, the performance of some algorithms—for instance, *bi-dger* (---)—is now underestimated; this is due to the fact that based on the access distance, certain operands are placed out of cache, while in practice they are (partially) brought into cache through either prefetching or because they share cache-lines across the innermost loop’s iterations. We address this disparity by further refining our micro-benchmarks.

6.2.4 Cache Prefetching

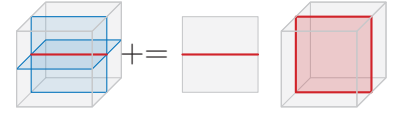
In the considered type of tensor contraction algorithms, prefetching of operands and sharing of cache-lines across loop iterations occur frequently.

Example 6.8: Prefetching and shared cache-lines

In algorithm *bi-dger* (---), the vector operand $A[:,i]$ points to a different memory location in each iteration i of the innermost loop:

```

for b = 1:b
  for i = 1:i
    dger:  $C[:,b,:] += A[:,i]B[i,b,:]^T$ 
  
```



However, since these vectors are consecutive in memory, when the end of $A[:,i]$ is reached, the prefetcher likely already loads the next elements, which constitute $A[:,i]$ in the next iteration. At the same time, the innermost loop over i indexes $B[i,b,:]$'s first dimension, and hence 8 consecutive operands $B[i,b,:]$ occupy the same cache-line⁷ (e.g., $B[0,b,:], \dots, B[7,b,:]$).

Such prefetching situations occur when the following conditions are met:

1. the operand varies across the directly surrounding loop, and
2. this loop's iterator indexes either
 - the first dimension of the operand,
 - or its second dimension, while the first is accessed entirely or fits in a single cache-line.

We test these conditions as part of our AST-based algorithm analysis, and when both are fulfilled, we use a slight modification of the previously introduced back-traversal of the AST to compute the *prefetch distance*, i.e., how long ago the prefetching occurred. These prefetch distances are then integrated into the micro-benchmark's setup just like the access distances, only that the prefetch accesses are limited to one cache-line along an operand's first dimension.

Example 6.9: Cache emulation with prefetch distances

In algorithm *ca-dgemv* (-----), for which [Example 6.7](#) constructed a cache-

⁷ Each cache-line fits 64 bytes = 8 doubles.

aware setup, operands $A[a,:]$ and $C[a,:,b]$ meet both prefetching conditions: 1) they vary with the surrounding loop's iterator a , and 2) a indexes their first dimensions (sharing of cache-lines). As a result, their prefetch distances are 0 bytes, and since their extent along the first, contiguously stored dimension is 1, the prefetching access loads them entirely. Since the remaining operand $B[:,c]$ has an access distance of 0 bytes, all operands are now accessed immediately before the kernel invocation; the setup is reduced to the accesses

$$C[a,:,c], A[a,:], B[:,c] .$$

Since this setup consists only of accesses to the operands, it becomes redundant in our micro-benchmarks, because each of the ten repetitions already touches all operands for the next repetition; hence, in such a case, we omit the setup altogether.

Accounting for prefetching, we obtain the performance predictions presented in Figure 6.7a. Here, several algorithms, such as b -dgemm (—) and ba -dgemv (---), are estimated closer to their measured performance, leading to a reduced average error of 19.1%. Note that this improvement also has a major influence on the fastest algorithm b -dgemm (—): Since its matrix operands $B[b:]$ of size $8 \times n$ are prefetched entirely by each preceding loop iteration, both of the dgemm_{NN} 's input operands are in-cache.

However, the new micro-benchmarks now overestimates the performance of several other algorithms, including ca -dgemv (---); i.e., the runtime is underestimated. There are two separate causes for this discrepancy:

- In several algorithms, such as ca -dgemv (---), where prefetching is implicit due to operands sharing cache-lines, the prefetcher fails once a new cache-line is reached.
- In other algorithms, such as bi -dger (---), the innermost loop is so short (here: 8 iterations) that each first iteration of the loop significantly impacts performance.

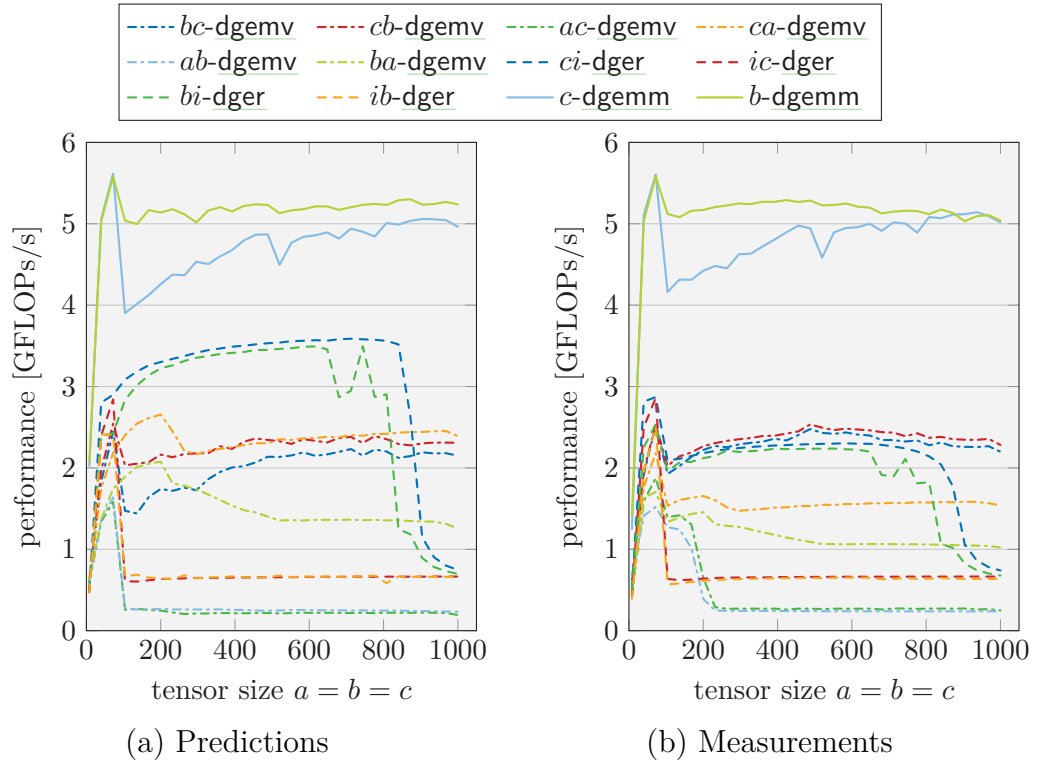


Figure 6.7: Performance predictions for $C_{abc} := A_{ai}B_{ibc}$ with cache emulation including prefetch distances.

($i = 8$, HARPertown E5450, 1 thread, OPENBLAS, median of 10 repetitions)

These two causes are treated separately in the following sections.

6.2.5 Prefetching Failures

When operands are identified as prefetched because they share cache-lines across iterations (i.e., the surrounding loop indexes their first dimension), the processor should prefetch the next cache-line every 8 iterations (1 cache-line = 8 doubles). However, as a detailed analysis of instrumented algorithms has shown, it *fails to do so*. As a result, in every 8th iteration of the innermost loop, the operand is not available and the kernel may take significantly longer.

We account for this prefetching-artefact by performing *two separate micro-benchmarks*: one simulating the 7 iterations in which the operand is available in

cache as before, and one for the 8th iteration. In this second micro-benchmark we account for the “prefetching failures”, and do not emulate a corresponding prefetching access. The prediction for the total runtime is now obtained by *weighting* these two benchmark timings according to their number of occurrences in the algorithm and summing their contributions.

Example 6.10: Benchmarks for prefetch failures

In algorithm *ca-dgemv* (---), the memory regions of both $A[a,:]$ and $C[a,:,c]$ each share cache-lines across iterations of the innermost loops over a . Hence, in every 8th iteration the kernel accesses a new cache line and its runtime increases drastically by a about $4.5\times$. To account for these “prefetching failures”, we introduce a second set of micro-benchmarks without the emulated prefetching accesses. For $a = b = c = 400$ and $i = 8$ this results in the same setup as without prefetching:

$$[816\ 632], A[a,:], [163\ 200], B[:, :, c] .$$

Figure 6.8a shows the predictions obtained after this improvement: The error is reduced to 14.7%. Most apparent in *ca-dgemv* (---), the overestimation of algorithms whose iterations share cache-lines are now corrected.

6.2.6 First Loop Iterations

The predictions for several algorithms, such as *ci-dger* (---), are still severely off, because the innermost loop of these algorithms is extremely short (in our example 8 iterations long). In such a case, the predictions are only accurate for all but the *first iteration*. Due to vastly different cache preconditions for this first iteration, however, its performance can differ significantly; e.g., in *ci-dger* (---) it is up to $10\times$ lower, which combined with the low total iteration count results in predictions that are off by up to $2\times$.

To treat such situations, we introduce separate micro-benchmarks to predict the performance of the first iterations of the innermost loop (and further loops if their first iterations account for more than 1 % of the total kernel invocations). For this purpose, the access distance evaluation is slightly modified: Instead

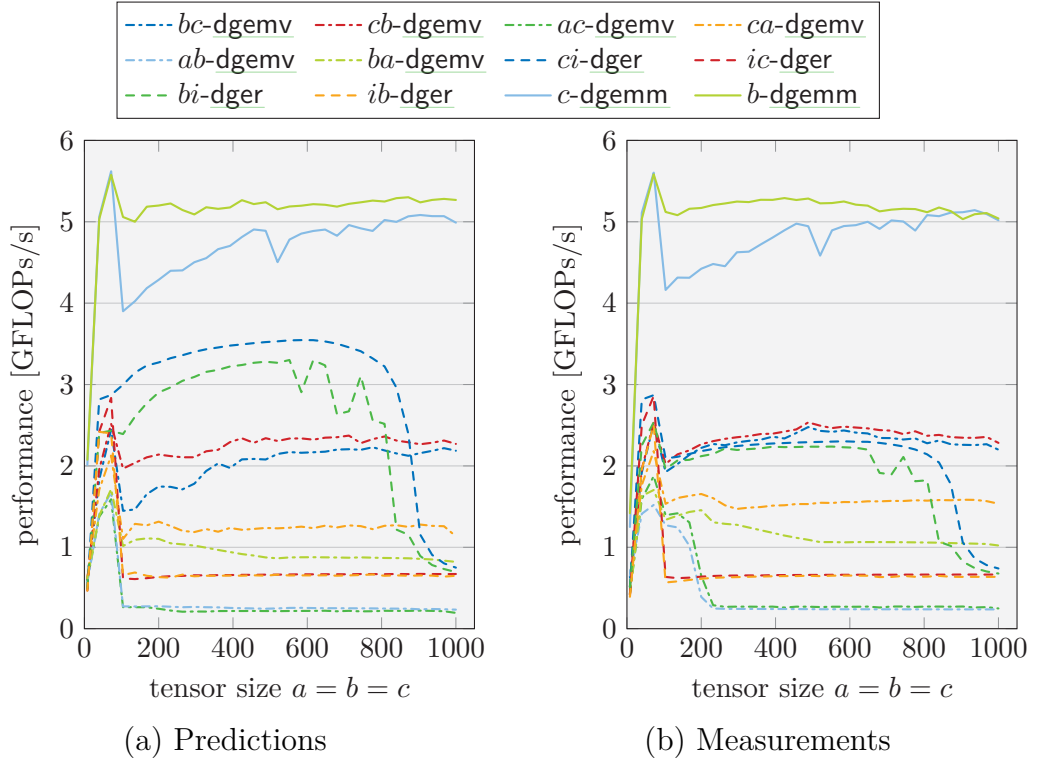


Figure 6.8: Performance predictions for $C_{abc} := A_{ai}B_{ibc}$ accounting for pre-fetching failures.

($i = 8$, HARPertown E5450, 1 thread, OPENBLAS, median of 10 repetitions)

of the kernel itself, the starting point is now the loop whose first iteration is considered, and the set M already contains all of the kernel's memory regions joined across this loop.

Example 6.11: First loop iterations

In algorithm *ci-dger* (---), the innermost loop over i is in our example only 8 iterations long. All but the first iteration use the same operand $C[:, :, c]$, and $A[:, i]$ and $B[i, :, c]$ are prefetched, leading to optimal conditions for performance. In the first iteration (i.e., the next c iteration) however, $C[:, :, c]$ refers to a different memory location and prefetching fails for both $A[:, i]$ and $B[i, :, c]$, leading to severely lower performance.

Based on these improved access distances, the cache setup and micro-

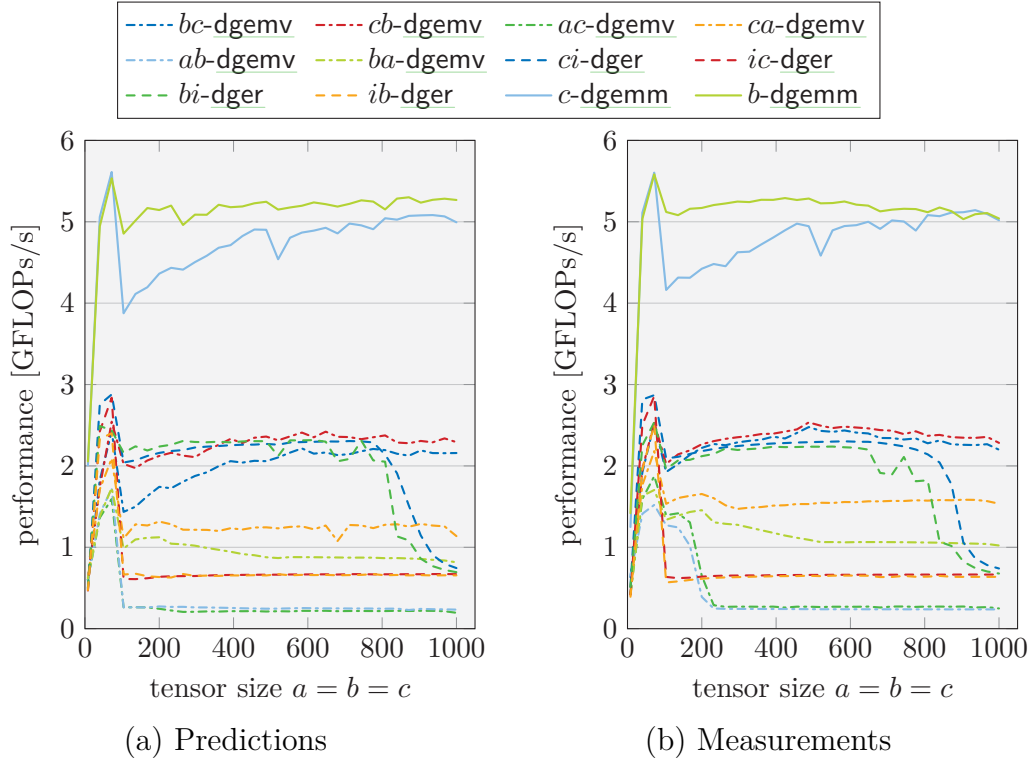


Figure 6.9: Final performance predictions for $C_{abc} := A_{ai}B_{ibc}$.

($i = 8$, HARPertown E5450, 1 thread, OPENBLAS, median of 10 repetitions)

benchmark are performed just as before. As before, the prediction for the total runtime is obtained from weighting all relevant benchmark timings with the corresponding number of occurrences within the algorithm.

In Figure 6.9a, we present the improved performance predictions obtained from this modification. The performance of all algorithms is now predicted with satisfying accuracy—the average absolute error is 9.47%.

6.3 Results

In order to showcase the applicability and effectiveness of our predictions, this section applies them to other contractions: Section 6.3.1 revisits $C_{abs} := A_{ai}B_{ibc}$ with entirely different problem sizes and a changed hardware and software

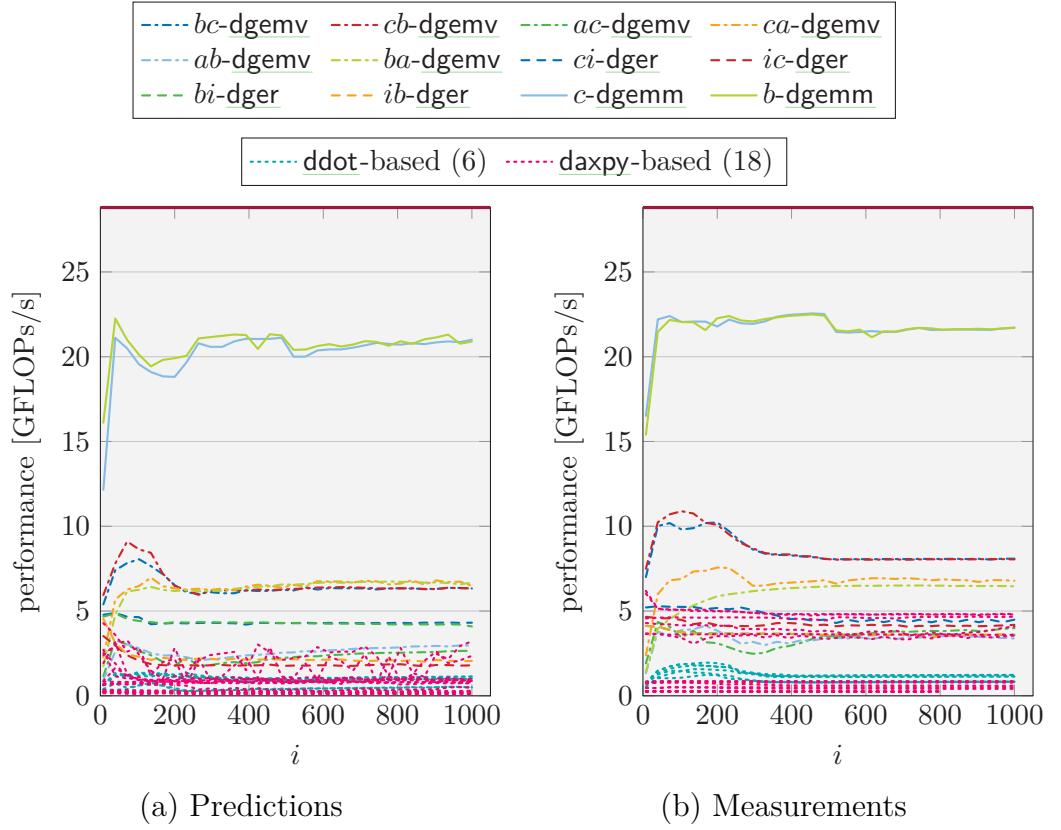


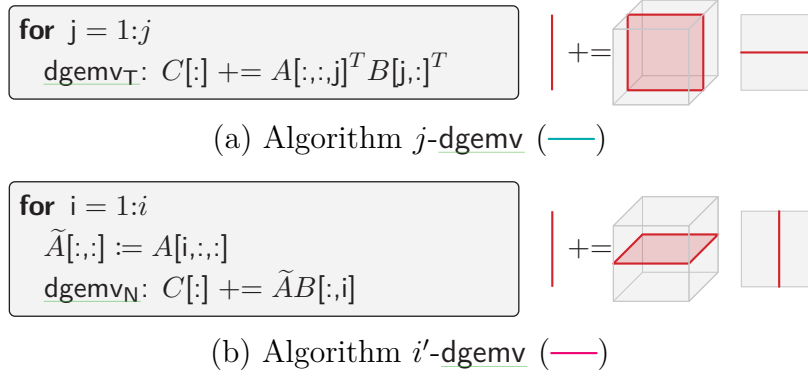
Figure 6.10: Performance predictions and measurements for $C_{abc} := A_{ai}B_{ibc}$ with $a = b = c = 128$ fixed.

(IVY BRIDGE-EP [E5-2680 v2](#), 1 thread, MKL, median of 10 repetitions)

setup, [Section 6.3.2](#) considers a contraction that only allows the use of BLAS Level 1 and 2 kernels, and [Section 6.3.3](#) studies a more complex contraction with numerous alternative algorithms and multi-threading.

6.3.1 Changing the Setup for $C_{abc} := A_{ai}B_{ibc}$

We consider the previously studied contraction with an entirely different setup: We use $a = b = c = 128$ and $i = 8, \dots, 1000$ in steps of 8 on an IVY BRIDGE-EP [E5-2680 v2](#) with single-threaded MKL. For this scenario, [Figure 6.10](#) presents the performance predictions and measurements for all 36 algorithms (see [Section 6.2.1](#)). Although everything, ranging from the problem sizes to

Figure 6.11: dgemv-based algorithms for $C_a := A_{iaj}B_{ji}$.

the machine and BLAS library was changed in this setup, the predictions are of equivalent quality and our tool correctly determines that the dgemm-based algorithms (—), (—) not only perform best and equally well but also reach over 75 % of the IVY BRIDGE’s theoretical peak performance of 28.8 GFLOPs/s.

6.3.2 Vector Contraction: $C_a := A_{iaj}B_{ji}$

For certain contractions (e.g., those involving vectors), dgemm cannot be used as a compute kernel, and algorithms can only be based on BLAS Level 1 or 2 kernels. One such scenario is encountered in the contraction $C_a := A_{iaj}B_{ji}$, for which our generator yields 8 algorithms:

- 4 ddot-based: aj-ddot (—), ja-ddot (—),
ai-ddot (—), ia-ddot (—);
- 2 daxpy-based: ij-daxpy (—), ji-daxpy (—), and
- 2 dgemv-based (see Figure 6.11): j-dgemv (—), i'-dgemv (—).

Note that since last algorithm operates on slices $A[i, :, :]$, which do not have contiguously-stored dimension, a copy kernel (indicated by the apostrophe in the algorithm name) is required before each dgemv_N (Figure 6.11b).

Figure 6.12 presents the predicted and measured performance for these algorithms. Our predictions clearly identify the fastest algorithm j-dgemv (—)

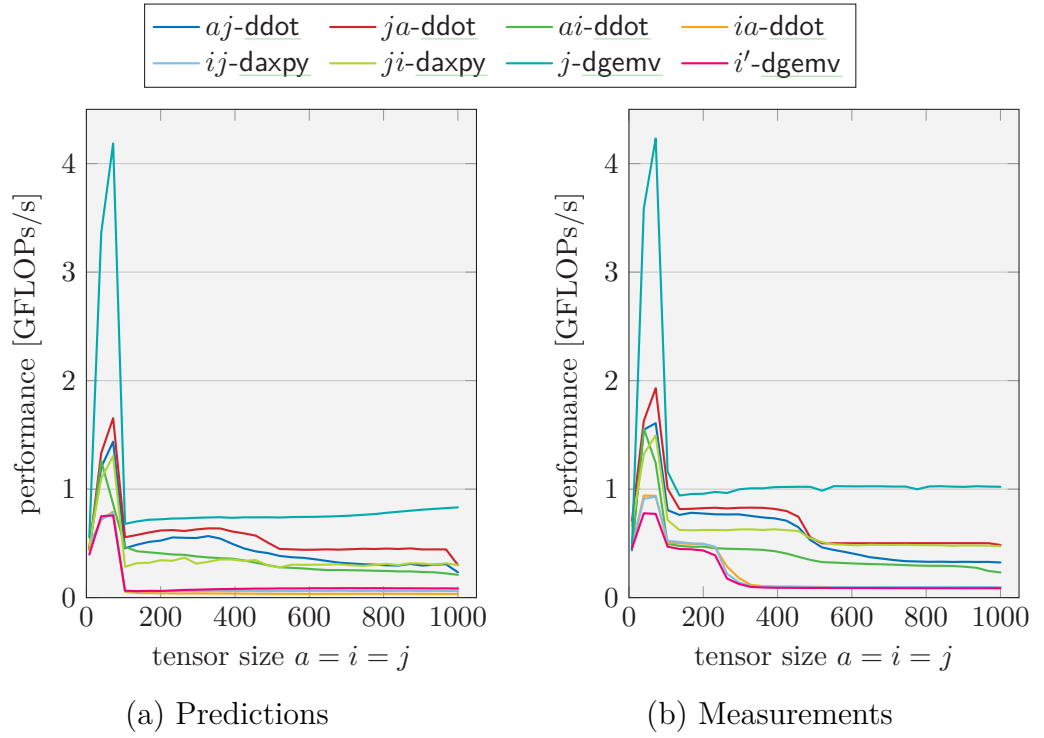


Figure 6.12: Performance predictions and measurements for $C_a := A_{iaj}B_{ji}$.
(HARPERTOWN E5450, 1 thread, OPENBLAS, median of 10 repetitions)

across the board. Furthermore, the next group of four algorithms is also correctly recognized, and the low performance of the second `dgemvN`-based algorithm `i'-dgemv` (—) (due to the overhead of the involved copy operation) is correctly predicted as well.

6.3.3 Challenging Contraction: $C_{abc} := A_{ija}B_{jbic}$

We now turn to a more complex example inspired by space-time continuum computations in the field general relativity [62]: $C_{abc} := A_{ija}B_{jbic}$. For this contraction, we generated a total of 176 different algorithms:

- 48 `ddot`-based (—),
- 72 `daxpy`-based (—),

```

for c = 1:c
  for j = 1:j
     $\tilde{B}[:,:] := B[j,:,c]$ 
     $\text{dgemm}_{\text{TT}}: C[:,c] += A[:,j,:]^T \tilde{B}^T$ 

```

(a) Algorithm cj' -dgemm (—)

```

for j = 1:j
  for c = 1:c
     $\tilde{B}[:,:] := B[j,:,c]$ 
     $\text{dgemm}_{\text{TT}}: C[:,c] += A[:,j,:]^T \tilde{B}^T$ 

```

(b) Algorithm jc' -dgemm (—)

```

for c = 1:c
  for i = 1:i
     $\tilde{A}[:,i] := A[i,:,c]$ 
     $\text{dgemm}_{\text{TN}}: C[:,c] += \tilde{A}^T B[:,i,c]$ 

```

(c) Algorithm ci' -dgemm (—)

```

for i = 1:i
   $\tilde{A}[:,i] := A[i,:,c]$ 
  for c = 1:c
     $\text{dgemm}_{\text{TN}}: C[:,c] += \tilde{A}^T B[:,i,c]$ 

```

(d) Algorithm $i'c$ -dgemm (—)

```

for b = 1:b
  for j = 1:j
     $\tilde{B}[:,j] := B[j,b,:]$ 
     $\text{dgemm}_{\text{TN}}: C[:,b] += A[:,j,:]^T \tilde{B}$ 

```

(e) Algorithm bj' -dgemm (—)

```

for j = 1:j
  for b = 1:b
     $\tilde{B}[:,j] := B[j,b,:]$ 
     $\text{dgemm}_{\text{TN}}: C[:,b] += A[:,j,:]^T \tilde{B}$ 

```

(f) Algorithm jb' -dgemm (—)

```

for b = 1:b
  for i = 1:i
     $\tilde{A}[:,i] := A[i,:,b]$ 
     $\text{dgemm}_{\text{TN}}: C[:,b] += \tilde{A}^T B[:,i,b]$ 

```

(g) Algorithm bi' -dgemm (—)

```

for i = 1:i
   $\tilde{A}[:,i] := A[i,:,b]$ 
  for b = 1:b
     $\text{dgemm}_{\text{TN}}: C[:,b] += \tilde{A}^T B[:,i,b]$ 

```

(h) Algorithm $i'b$ -dgemm (—)Figure 6.13: dgemm-based algorithms for $C_{abc} := A_{ija}B_{jbc}$.

- 36 dgemv-based (---),
- 12 dger-based (---), and
- 8 dgemm-based:
 - cj' -dgemm (—), jc' -dgemm (—), ci' -dgemm (—), $i'c$ -dgemm (—),
 - bj' -dgemm (—), jb' -dgemm (—), bi' -dgemm (—), $i'b$ -dgemm (—).

All dgemm-based (see Figure 6.13) and several of the dgemv-based algorithms involve copy operations to ensure that each matrix has a contiguously-stored

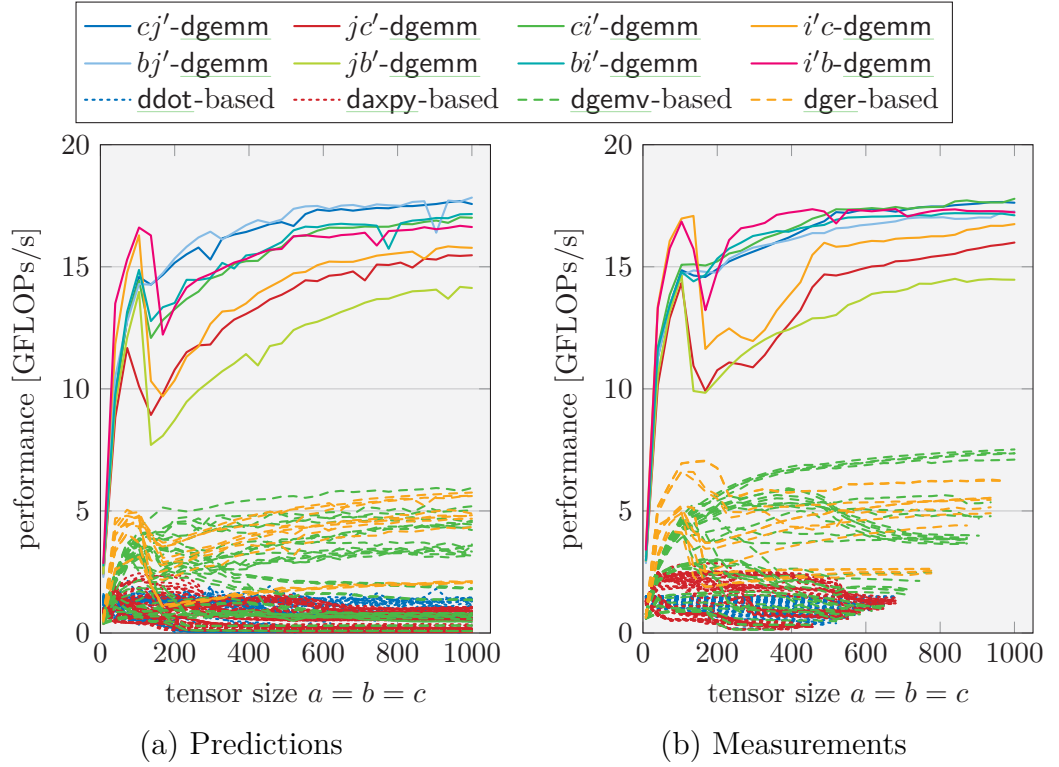


Figure 6.14: Performance predictions and measurements for $C_{abc} := A_{ija}B_{jbic}$.
 ($i = j = 8$, IVY BRIDGE-EP E5-2680 v2, 1 thread, OPENBLAS, median of 10 repetitions)

dimension as required by the BLAS interface. Once again, we consider a challenging scenario where both contracted indices are of size $i = j = 8$ and the free indices $a = b = c$ vary between 8 and 1000.

Figure 6.14a presents the predicted performance of the 176 algorithms, where algorithms based on BLAS Level 1 and 2 are grouped by kernel. Even with the copy operations, the dgemm-based algorithms are the fastest. However, within these 8 algorithms, the performance differs by more than 20%. Figure 6.14b compares our predictions with corresponding performance measurements⁸: Among the dgemm-based algorithms, our predictions clearly separate the bulk of fast algorithms from the slightly less efficient ones.

⁸ Slow tensor contraction algorithms were stopped before reaching the largest problem size by limiting the total measurement time per algorithm to 15 min.

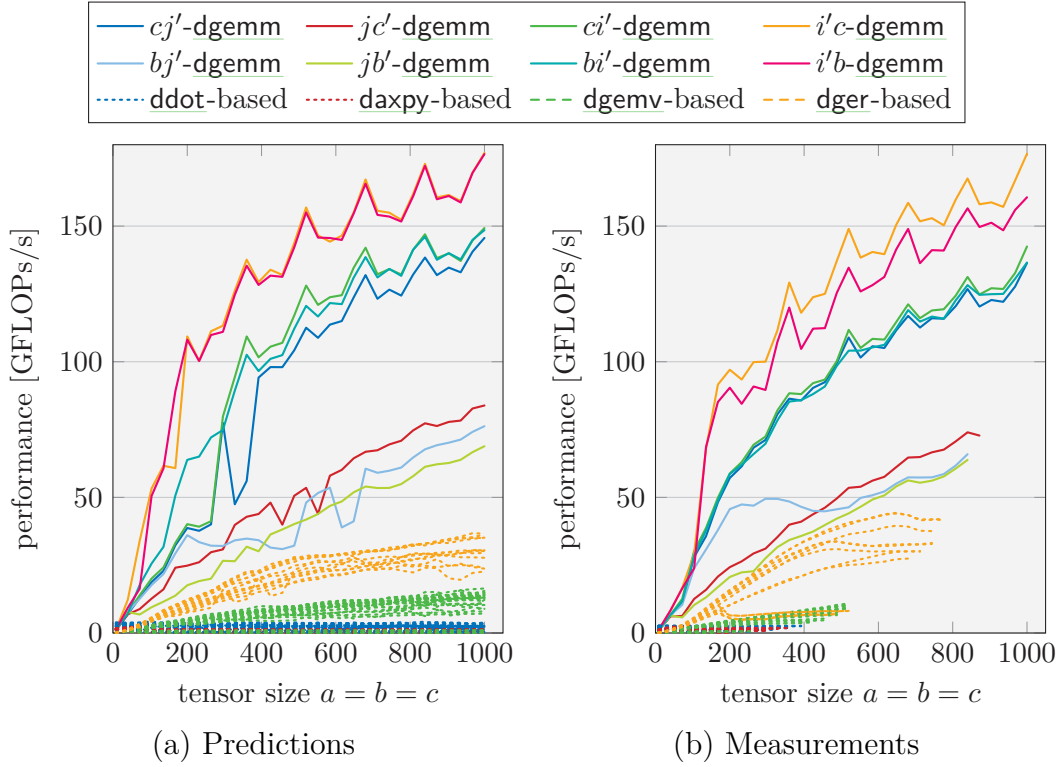


Figure 6.15: Performance predictions and measurements for $C_{abc} := A_{ija}B_{jbc}$ on 10 cores.

($i = j = 32$, IVY BRIDGE-EP E5-2680 v2, OPENBLAS, median of 10 repetitions)

Multi-Threading

Our contraction algorithms can profit from shared-memory parallelism through multi-threaded BLAS kernels. To focus on the impact of parallelism, we increase the contracted tensor dimension sizes to $i = j = 32$ and use all 10 cores of the IVY BRIDGE-EP E5-2680 v2 with multi-threaded OPENBLAS. Figure 6.15 presents performance predictions and measurements for this setup: Our predictions accurately distinguish the three groups of dgemm-based implementations, and algorithms ic'-dgemm (—) and ib'-dgemm (—) (see Figure 6.13), which reach 170 GFLOPs/s, are correctly identified as the fastest. jb'-dgemm (—) on the other hand merely reaches 60 GFLOPs/s. This

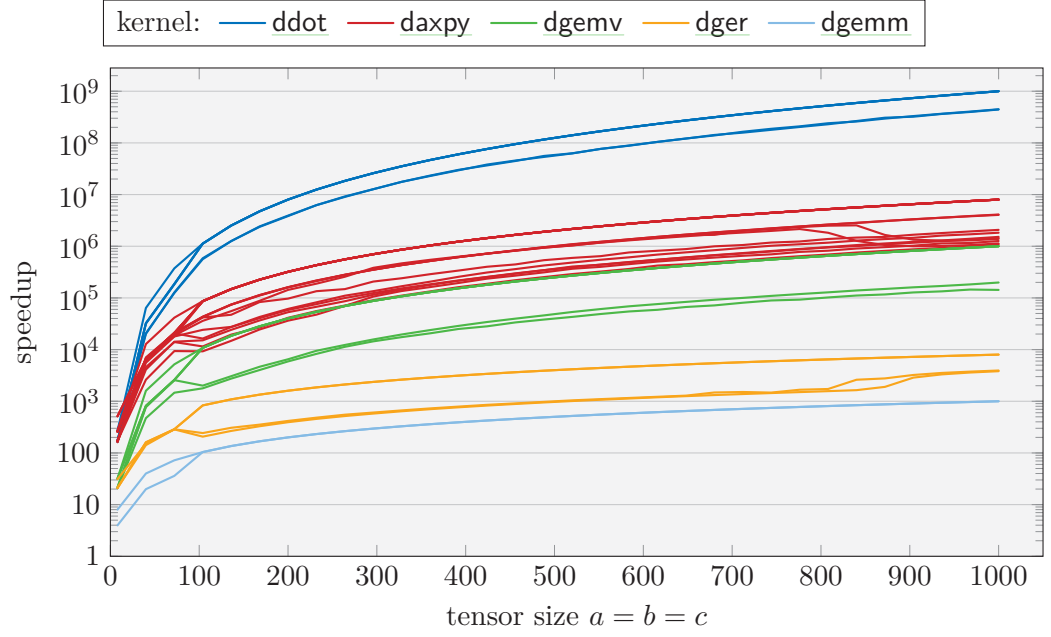


Figure 6.16: Speedup of predictions over algorithm executions for $C_{abc} := A_{ai}B_{ibc}$.
 ($i = 8$, IVY BRIDGE-EP [E5-2680 v2](#), 1 thread, OPENBLAS, median of 10 repetitions)

$3\times$ difference in performance among `dgemm`-based algorithms emphasizes the importance of selecting the right algorithm.

6.3.4 Efficiency Study

The above study provided evidence that our automated approach successfully identifies the most efficient algorithm(s). In the following we show how much faster this approach is compared to empirical measurements. For this purpose, we once more consider the contraction $C_{abc} := A_{ai}B_{ibc}$ with $i = 8$ and varying $a = b = c$ on a HARPETOWN [E5450](#) with OPENBLAS. [Figure 6.16](#) presents the speedup of our micro-benchmark over corresponding algorithm measurements: Generally our predictions are several orders of magnitude faster than such algorithm executions. For $a = b = c = 1000$, this relative improvement is smallest for the `dgemm`-based algorithms (—) at $1000\times$, because each `dgemm` performs a significant portion of the computation; for the `dger`-based

algorithms (—), it lies between 6000 and 10 000 and for the `dgemv`-based algorithms (—) the gain is $5 \cdot 10^5 \times$ to $10^6 \times$; finally, for the BLAS Level 1-based algorithms (—, —), where each kernel invocation only performs a tiny fraction of the contraction, our predictions are $1 \cdot 10^6$ to $1 \cdot 10^9$ times faster than the algorithm executions.

6.4 Summary

This chapter focused on the performance prediction of automatically-generated BLAS-based algorithms for tensors contractions. We tackled the problem of selecting the fastest algorithm without ever executing it. Instead, our approach is based on timing the BLAS kernels in a small set of micro-benchmarks that emulate the execution context of the algorithms. Thanks to careful treatment of cache-locality and a model of the cache prefetcher’s behavior, our performance predictions are capable of identifying the best-performing algorithm in a tiny fraction of the time required to actually run any of the alternatives.

The quality of the predictions was showcased for a number of challenging scenarios, including contractions among tensors with small dimensions, contractions that can only be cast in terms of BLAS Level 1 and 2 kernels, and multi-threaded computations.

7 Conclusion

This dissertation set out to predict the performance of dense linear algebra algorithms. It targeted two types of algorithms that require different prediction approaches: blocked algorithms and tensor contractions.

For blocked algorithms, we accomplished accurate performance predictions through automatically generated performance models for compute kernels. Our predictions both reliably identify the fastest blocked algorithm from potentially large numbers of available alternatives, and select a block size for near-optimal algorithm performance. Our approach’s main advantage is its separation of the model generation and the performance prediction: While the generation may take several hours, thousands of algorithm executions are afterwards predicted within seconds. A discussed downside to the approach, however, is that it does not account for algorithm-dependent caching effects.

For tensor contractions, we established performance predictions that identify the fastest among potentially hundreds of alternative BLAS-based contraction algorithms. By using cache-aware micro-benchmarks instead of our performance models, our solution is highly accurate even for contractions with severely skewed dimensions. Furthermore, since these micro-benchmarks only execute a tiny fraction of each tensor contraction, they provide performance predictions orders of magnitude faster than empirical measurements.

Together, our model generation framework and micro-benchmarks form a solid foundation for accurate and fast performance prediction for dense linear algebra algorithms.

7.1 Outlook

The techniques presented in this dissertations offer numerous opportunities for applications and extensions:

- Our methods can be applied to predict the performance various types of algorithms and operations, such as recursive algorithms and algorithms-by-blocks.
- For dense eigenvalue solvers, our models can predict the two most computationally intensive stages: The reduction to tridiagonal form and the back-transformation. By additionally estimating the data-dependent performance of tridiagonal eigensolvers, one can predict the solution of complete eigenproblems.
- Beyond individual operations, our predictions can be applied to composite operations and algorithms, such as matrix chain multiplications or least squares solvers.
- Our models were designed to provide estimates for configurable yet limited ranges of problem sizes. For extrapolations to larger problems they should be revised to ensure that local performance phenomena do not distort faraway estimates.
- Computations on distributed-memory machines, accelerators, and graphics cards can be predicted by combining our techniques with models for data movement and communication.

A Terminology:

Performance and Efficiency

In a nutshell, performance is the rate at which a software—such as a code segment, a routine, or an entire application—performs useful work, and efficiency is the ratio of the attained performance to the used processor’s theoretical peak performance.

This appendix introduces these concepts in detail and thereby provides the terminology used throughout this work. It is intended for readers new to the high-performance computing and as a small reference. It covers the following material:

- [Section A.1](#) describes an operation’s implementation-independent *workload* in terms of *floating-point operations*, *data volume* and *movement*, and *arithmetic intensity*.
- [Section A.2](#) details *cycle accurate timing*, which allows to measure the *runtime* of a computation with high precision.
- [Section A.3](#) defines a computation’s *attained performance* and *bandwidth* based on its workload and runtime.
- [Section A.4](#) briefly introduces the *hardware capabilities* relevant to dense linear algebra computations, such as *peak performance* and *peak bandwidth*.
- [Section A.5](#) differentiates between *bandwidth-* or *compute-bound* computations by relating the attained performance to the hardware capabilities,

and evaluates a computation’s *efficiency*—the most meaningful metric to quantify how well a piece of software performs its work.

- [Section A.6](#) gives an overview of other performance-related measures, such as *hardware counters* and *energy metrics*.

A.1 Workload

In scientific computing, the ultimately most desirable measure of a computation’s work is the “amount of new science performed”, which, however, is impractical to quantify—not least because it may well be opinion-based. Instead, we resort to simpler, computation-oriented metrics, namely the *number of arithmetic operations* required to perform a operation ([Section A.1.1](#)), and the involved *data volume* and *movement* ([Section A.1.2](#)). Furthermore, useful characterization of an operation’s workload is its ratio of arithmetic operations to memory accesses—called *arithmetic intensity*—is a useful characterization of its workload ([Section A.1.3](#)).

A.1.1 Floating-Point Operations

Most scientific computations, as complex as they may be, perform their work through a small set of elementary arithmetic operations on floating-point representations of real numbers, such as scalar additions or multiplications¹—These the so-called *floating-point operations* (FLOPs).²

Contemporary hardware offers two floating-point precisions standardized in IEEE 754 [[58](#)]: *single-precision*, and *double-precision*. They differ in the range of representable numbers, their representation accuracy, and their implementation in hardware. While we distinguish between single-precision FLOPs and double-precision FLOPs, throughout this work we are mostly concerned with double-precision computations. Hence we use “FLOPs” without a specification refers

¹ Exceptions that work on integer data or other structures include graph algorithms and discrete optimization.

² Not to be confused with floating-point operations *per second* (FLOPs/s).

to double-precision floating-point operations, and \mathbb{R} is used to denote double-precision numbers.

As commonly practiced in dense linear algebra, we assume that the multiplication of two $n \times n$ matrices requires $2n^3$ FLOPs—it has an asymptotic complexity of $O(n^3)$. While algorithms with lower asymptotic complexities (such as the *Strassen algorithm* with a complexity of $O(n^{2.807})$ [76] or the *Coppersmith-Winograd algorithm* with a complexity of $O(n^{2.376})$ [35]) were already known in the 1970s, due to considerably higher constant factors they found little to no application in high-performance computing until recently [55].

The FLOP-count of most dense linear algebra operations such as the matrix-matrix multiplication is *data-independent*, i.e., the operand entries do not affect what arithmetic operations are performed.³ In particular, this means that all multiplications with 0's are explicitly performed no matter how sparse an operand is (i.e., how few non-zero entries it has). A notable exception to the data-independence are numerical eigensolvers, whose FLOP-counts depend on the eigenspectrum of the input matrix; however, we do not study eigensolvers in further detail in this work.

Assuming the cubic complexity of the matrix-matrix multiplication, the data-independence allows us to compute the *minimal FLOP-count*—also referred to as *cost*—for most operations solely based on their operands' sizes.

Example A.1: Minimal FLOP-counts

The vector inner product $\alpha := x^T y$ (`ddot`) with $x, y \in \mathbb{R}^n$ costs $2n$ FLOPs: one multiplication and one addition per vector entry.

The solution of a triangular linear system with multiple right-hand-sides $\bar{B} := \bar{A}^{-1} \bar{B}$ (`dtrsmLLNN`) with $\bar{A} \in \mathbb{R}^{n \times n}$ and $\bar{B} \in \mathbb{R}^{n \times m}$ requires $n^2 m$ FLOPs.

The Cholesky decomposition of a symmetric positive definite (SPD)

³ Exceptions may be caused by corrupted input, such as NaNs, or floating-point exceptions, such as division by 0 or under-/overflows.

matrix $\begin{bmatrix} L & L^T \end{bmatrix} := \begin{bmatrix} A \end{bmatrix}$ (`dpotrfL`) with $\begin{bmatrix} A \end{bmatrix} \in \mathbb{R}^{n \times n}$ costs

$$\frac{1}{6}n(n+1)(2n+1) \text{ FLOPs} \approx \frac{1}{3}n^3 \text{ FLOPs} .$$

Note that an operation’s minimal FLOP-count only provides a lower bound for routines implementing it; reasons for exceeding this bound range from technical limitations to cache-aware data movement patterns and algorithmic schemes that perform extra FLOPs to use faster compute kernels.

A.1.2 Data Volume and Movement

The largest portion of a scientific computation’s memory footprint is typically occupied by its numerical data consisting of floating-point numbers. A real number in single- and double-precision requires, respectively, 4 and 8 bytes, whereas complex numbers are represented as two consecutive real numbers and thus require twice the space. Since throughout this work we mostly use double-precision numbers—conventionally called “*doubles*”—we can proceed with the assumption that each number takes up 8 bytes.

In dense linear algebra, the *data volume* (in bytes) involved in a computation is determined almost exclusively by the involved matrix operands. For instance, a square matrix of size 1000×1000 consists of 10^6 doubles $= 8 \cdot 10^6$ bytes ≈ 7.63 MiB;⁴ vector and scalar operands in comparison take up little space: A vector of size 1000 requires 8000 bytes $= 7.81$ KiB, and a scalar fits in just 8 bytes.

While a computation’s data volume describes how much data is involved in an operation, it says nothing about how often it is accessed. For this purpose we introduce the concept of *data movement* that quantifies how much data is read from or written to memory. A computation’s data movement is commonly higher than its data volume, because (parts of) the data are accessed multiple times.

⁴ We use the 1024-based binary prefixes for data volumes: 1024 bytes = 1 KiB (“kibibyte”), 1024 KiB = 1 MiB (“mebibyte”), and 1024 MiB = 1 GiB (“gibibyte”).

While the actual data movement of any dense linear algebra operation is highly implementation dependent, we can easily derive the *minimal data movement* from the operation's mathematical formulation by summing the size of all input and output operands, counting the operands that are both input and output twice.

Example A.2: Data volume and movement

The vector inner product $\alpha := -x^T y$ (`ddot`) with $x, y \in \mathbb{R}^n$ involves a data volume of $2n$ doubles = $16n$ bytes (ignoring the scalar α); since both x and y need only be read once the data movement is also $16n$ bytes.

The matrix-matrix product $C := A B + C$ (`dgemmNN`) with $A, B, C \in \mathbb{R}^{n \times n}$ involves a data volume of $3n^2$ doubles = $24n^2$ bytes, however, since C is updated, the minimal data movement is $4n^2$ doubles = $32n^2$ bytes.

The Cholesky decomposition $L L^T := A$ (`dpotrf`) with $A \in \mathbb{R}^{n \times n}$ uses only the lower-triangular part of the symmetric matrix A ,⁵ and A is decomposed in place, i.e., it is overwritten by L upon completion. Hence the data volume is $\frac{1}{2}n(n+1)$ doubles $\approx 4n^2$ bytes, while the minimal data movement is at least $2 \cdot \frac{1}{2}n(n+1)$ doubles $\approx 8n^2$ bytes.

Note that the minimal data movement is a strict lower bound when none of the involved data is in any of the processor's caches. Furthermore, depending on the operation and the cache sizes, it may not be attainable in implementations.

A.1.3 Arithmetic Intensity

Dividing an operation's minimal flop count by its minimal data movement yields its *arithmetic intensity*:

$$\text{arithmetic intensity} \stackrel{\text{def}}{:=} \frac{\text{minimal FLOP-count}}{\text{minimal data movement}} . \quad (\text{A.1})$$

⁵ Space for the whole matrix is allocated, but the strictly upper-triangular part is not accessed.

A low arithmetic intensity means that few operations are performed per memory access, thus making the data movement a likely bottleneck; a high arithmetic intensity on the other hand indicates that a lot of work is performed per data element, thus making the floating-point computations the potential bottleneck. Arithmetic intensity divides dense linear algebra operations into two groups: While for BLAS Level 1 (vector-vector) and 2 (matrix-vector) operations the intensity is quite small and independent of the problem size, it is considerably larger for BLAS Level 3 (matrix-matrix) and dense LAPACK-level operations, for which increases linearly with the problem size.

Example A.3: Arithmetic intensity

The vector inner product $\alpha := x^T y$ (`ddot`) with $x, y \in \mathbb{R}^n$ is a BLAS Level 1 operation that performs $2n$ FLOPs over $2n$ doubles of data movement. Hence its arithmetic intensity is

$$\frac{\text{minimal FLOP-count}}{\text{minimal data movement}} = \frac{2n \text{ FLOPs}}{2n \text{ doubles}} = \frac{1}{8} \text{ FLOPs/byte} .$$

The matrix-vector multiplication $y := A x + y$ (`dgemvN`) with $A \in \mathbb{R}^{n \times n}$ and $x, y \in \mathbb{R}^n$ is a BLAS Level 2 operation that performs $2n^2$ FLOPs over $n^2 + 3n$ doubles of data movement (y is both read and written). Therefore, its arithmetic intensity is

$$\frac{\text{minimal FLOP-count}}{\text{minimal data movement}} = \frac{2n^2 \text{ FLOPs}}{n^2 + 3n \text{ doubles}} \approx \frac{1}{4} \text{ FLOPs/byte} .$$

The matrix-matrix multiplication $C := A B + C$ (`dgemmNN`) with $A, B, C \in \mathbb{R}^{n \times n}$ is a BLAS Level 3 that performs $2n^3$ FLOPs over $4n^2$ doubles of data movement (C is both read and written). Hence, its arithmetic intensity

$$\frac{\text{minimal FLOP-count}}{\text{minimal data movement}} = \frac{2n^3 \text{ FLOPs}}{4n^2 \text{ doubles}} = \frac{n}{16} \text{ FLOPs/byte}$$

grows linearly with the problem size n and already exceeds the intensity of `dgemv` for matrices as small as 5×5 .

We revisit the arithmetic intensity in [Section A.5](#), where it determines whether a computation’s performance is limited by the processor’s memory subsystem or its floating-point units.

A.2 Runtime

Since performance describes the amount of work performed per time unit, it is a critical requirement to accurately measure a calculation’s *runtime*, i.e., the duration of its execution. This can be achieved in any number of ways, such as the UNIX command `time`, the UNIX function `gettimeofday()`, or the OPENMP routine `omp_get_wtime()`. While all of these measure time in seconds or fractions thereof, we are interested in a *cycle-accurate timer* that counts exactly how many processor cycles a computation took.

On x86 and x86_64 machines, the assembly instruction `rdtsc` (read time stamp counter) returns the value of the Time Stamp Counter, a 64-bit register that is incremented once per cycle at the processor’s *base frequency*.⁶ It provides a cycle-accurate timer with minimal overhead, and its cycle count can be converted to seconds through multiplication with the base frequency.

While we use `rdtsc` as a cycle accurate timer throughout most of this work, we need to be aware that it does not necessarily count actual core cycles. Although the increment rate of the Time Stamp Counter is fixed at the processor’s base frequency, the individual cores may run at *varying frequencies*—both lower and higher to adapt to their current workload: While during idle times, the frequency is reduced to save energy, during peak loads, exceeding the base frequency provides a performance boost. On INTEL processors, the *SPEEDSTEP* technology (or ENHANCED INTEL SPEEDSTEP—EIST) to dynamically scale the frequency was introduced in 2005 (AMD’s counterpart is called AMD

⁶ Technically, it is only guaranteed to be incremented at a constant rate, which we observed to be to the processor’s base frequency on all machines used in this work. However, one could easily adapt to any other frequency through multiplication with a constant factor.

POWERTUNE), and in 2008 *INTELTURBO BOOST* added the ability to scale beyond the base frequency—often called “dynamic overclocking”—up to a model-dependent *maximum turbo frequency*. However, this peak frequency can typically not be maintained indefinitely, since it increases the processor’s power consumption and temperature, which cannot exceed certain model-specific limits (see [Example 2.3](#)).

If we are specifically interested in counting core cycles at the dynamic frequency, the PERFORMANCE APPLICATION PROGRAMMING INTERFACE (PAPI) [28, 119] offers a solution in the form of the hardware performance counter *PAPI_TOT_CYC*. However, PAPI, which is integrated into our performance measurement tool and framework presented in [Section 2.2](#), not only introduces a significantly larger overhead than *rdtsc*, but is also not available on all platforms (e.g., MACOS).

A.3 Performance and Attained Bandwidth

In scientific computing the central metric that describes at what rate a computation performs its work is *floating-point performance*—or simply *performance*—measured in GFLOPs/s (giga-FLOPs per second, sometimes abbreviated as GFLOPS). For a dense linear algebra computation, it is the result of dividing the operation’s minimal FLOP-count (cost) by the measured runtime:

$$\text{performance} \stackrel{\text{def}}{:=} \frac{\text{minimal FLOP-count}}{\text{runtime}} . \quad (\text{A.2})$$

Example A.4: Performance

The matrix-matrix multiplication $C := A B + C$ (*dgemm_{NN}*) with $A, B, C \in \mathbb{R}^{1000 \times 1000}$ requires 2×1000^3 FLOPs $= 2 \cdot 10^9$ FLOPs. If it is computed in 102 ms, it attained a floating-point performance of

$$\frac{\text{minimal FLOP-count}}{\text{runtime}} = \frac{2 \cdot 10^9 \text{ FLOPs}}{102 \text{ ms}} \approx 19.61 \text{ GFLOPs/s} .$$

Similarly, dividing an operation’s minimal data movement by its measured

runtime yields its *attained bandwidth* measured in GiB/s:

$$\text{attained bandwidth} \stackrel{\text{def}}{:=} \frac{\text{minimal data movement}}{\text{runtime}} . \quad (\text{A.3})$$

Note that we define the attained bandwidth independent of whether the hardware's available bandwidth is a computation's limiting factor. Performance and attained bandwidth are related to the hardware capabilities in [Section A.5](#).

Example A.5: Attained bandwidth

The vector inner product $\alpha := x^T y$ (`ddot`) with $x, y \in \mathbb{R}^{100\,000}$ has a minimal data movement of $2 \times 100\,000$ doubles ≈ 1.53 MiB. If it is performed in 0.13 ms while loading both x and y from main memory (i.e., they were not in any of the processor's caches; see also [Section 2.1.5](#)), it attained a bandwidth of

$$\frac{\text{minimal data movement}}{\text{runtime}} = \frac{1.53 \text{ MiB}}{0.13 \text{ ms}} \approx 11.49 \text{ GiB/s} .$$

Both performance and the attained bandwidth were so far not put into the context of the used hardware and its capabilities. As such, they provide an idea how fast a computation was, yet not how well it used the available resources. To evaluate how efficiently the hardware was used, we first need to understand the hardware capabilities and limitations.

A.4 Hardware Constraints

Dense linear algebra operations on shared-memory machines are generally constrained by the processor's capabilities in terms of floating-point performance and bandwidth, which are covered in this section.

A quick overview of what hardware resources perform floating-point operations allows us to easily determine the physical limitations to floating-point performance: Within a processor's core, floating-point operations are performed in the form of *floating-point instructions*. In particular, contemporary processors offer so-called *vectorized* instructions that operate on vectors of 2 to

16 floating-point numbers simultaneously. Both the length of these vectors and how many vectorized instructions can be issued each cycle are determined by a processor's floating-point hardware and instruction set. Multiplying the total number of scalar operations per cycle with the frequency and number cores yields the processor's *peak floating-point performance* in GFLOPs/s:

$$\text{peak performance} \stackrel{\text{def}}{=} \frac{\text{FLOPs}}{\text{cycle and core}} \times \text{frequency} \times \#\text{cores} . \quad (\text{A.4})$$

Example A.6: Peak floating-point performance

A SANDY BRIDGE-EP E5-2670 can operate on vectors of 4 doubles with its ADVANCED VECTOR EXTENSIONS (AVX). It is capable of one vectorized addition and one vectorized multiplication instruction per cycle and core, i.e., a total of 8 FLOPs/cycle/core. At the processor's base frequency of 2.6 GHz each of its cores has a peak double-precision floating-point performance of

$$8 \text{ FLOPs/cycle/core} \times 2.6 \cdot 10^9 \text{ cycles/s} = 20.8 \text{ GFLOPs/s/core} .$$

Hence, the total peak performance of the processor is

$$20.8 \text{ GFLOPs/s/core} \times 8 \text{ cores} = 166.4 \text{ GFLOPs/s} .$$

At the processor's maximum turbo frequency of 3.5 GHz, the peak performance is about 35 % higher: 28 GFLOPs/s/core and 224 GFLOPs/s in total.

The AVX registers and instructions also allow to operate on vectors of 8 single-precision numbers while still offering one vector addition and one vector multiplication each cycle. Hence the peak single-precision floating-point performance is twice the peak double-precision performance, i.e., 448 GFLOPs/s using all 8 cores and TURBO BOOST.

A computation's data movement is limited by a processor's *peak main-memory bandwidth*, i.e., how much data it can load from and store to main memory per second. This theoretical peak can be computed from the I/O bus

frequency, the bus width, and the number of memory channels, but is usually easily found in the manufacturer’s specifications. Note that this nominal peak bandwidth always assumes that the processor is equipped with the fastest compatible main-memory.

A machine’s peak bandwidth can only be attained using multiple cores; using a single core, the bandwidth is determined by the memory latency and the maximum number of pending (“in-flight”) cache-misses plus the rate at which the prefetcher loads cache-lines [69]. Unfortunately, since especially the prefetcher is typically not well documented, it is difficult to determine a theoretical single-core peak bandwidth.

In practice, the peak bandwidth is commonly measured with benchmarks such as STREAM [68, 124], LIKWID [79, 114], or a highly tuned BLAS Level 1 kernel (e.g., `daxpy`). While such benchmarks do not report the theoretical peak bandwidth, they give an excellent estimate of the practically attainable bandwidth.

Example A.7: Peak bandwidth

A SANDY BRIDGE-EP E5-2670 has a documented peak bandwidth of 51.2 GB/s. This bandwidth is the result of a 4 memory channels each loading 8 bytes simultaneously from a DDR3-1600 main-memory module over a bus running at 800 MHz:

$$4 \text{ channels} \times 8 \text{ bytes/channel} \times 800 \text{ MHz} = 51.2 \text{ GB/s} = 47.68 \text{ GiB/s} .$$

However, the `load`-benchmark from the LIKWID suite only reports a practical peak bandwidth for the entire processor of 37.65 MiB/s. To determine the single-threaded peak bandwidth, we used the highly tuned OPENBLAS kernel `daxpy` ($y := \alpha x + y$) with vectors of size 10 000 000 (76.29 MiB per vector). Since `daxpy`’s minimal memory movement is 3 vectors (load x , update y) and it took 14.77 ms in our measurements it attained a bandwidth of

$$\frac{3 \times 76.29 \text{ MiB}}{14.77 \text{ ms}} = 16.25 \text{ GiB/s} .$$

A.5 Efficiency

To determine a computation’s theoretical attainable peak performance on a specific processor, we compare the computation’s arithmetic intensity to the hardware’s ratio of peak floating-point performance to peak bandwidth (both in FLOPs/byte). If the arithmetic intensity is higher than this ratio, the computation is limited by the peak floating-point performance and said to be *compute-bound*; if it is lower, it is limited by the bandwidth and said to be *bandwidth-bound*. While in the compute-bound case, a computation’s efficiency is the ratio of its attained performance to the processor’s peak floating-point performance (Section A.5.1), in the bandwidth-bound case it is the ratio of the attained bandwidth to the processor’s peak bandwidth (Section A.5.2). Finally, the Roofline Model (Section A.5.3) provides a visualization combining the arithmetic intensity and both types of efficiency.

A.5.1 Compute-Bound Efficiency

A computation is compute-bound on a hardware platform if the memory operations to load and store the involved data can be amortized by floating-point operations, i.e., the available memory bandwidth is sufficient for all transfers and the speed at which the processor performs FLOPs is the bottleneck. An operation is theoretically bandwidth-bound when

$$\text{arithmetic intensity} \geq \frac{\text{peak performance}}{\text{peak bandwidth}} .$$

Furthermore, a computation’s *compute-bound efficiency* (or simply *efficiency*) is given by

$$\text{compute-bound efficiency} \stackrel{\text{def}}{=} \frac{\text{attained performance}}{\text{peak performance}} . \quad (\text{A.5})$$

This unit-less metric between 0 and 1 indicates how well the available hardware resources are utilized: While a value close to 1 corresponds to near-optimal utilization, lower values indicate untapped resource potential.

Example A.8: Compute-bound efficiency

The matrix-matrix multiplication $\boxed{C} := \boxed{A} \boxed{B} + \boxed{C}$ (`dgemmNN`) with $\boxed{A}, \boxed{B}, \boxed{C} \in \mathbb{R}^{1000 \times 1000}$ has an arithmetic intensity of (see [Example A.3](#))

$$1000 \times \frac{1}{16} \text{ FLOPs/byte} = 62.5 \text{ FLOPs/byte} .$$

On a single core of a SANDY BRIDGE-EP [E5-2670](#) with a peak floating-point performance of 20.8 GFLOPs/s (TURBO BOOST disabled) and peak bandwidth of 51.2 GiB/s this operation is clearly compute-bound:

$$\frac{20.8 \text{ GFLOPs/s}}{16.25 \text{ GiB/s}} \approx 1.28 \text{ FLOPs/byte} < 62.5 \text{ FLOPs/byte} .$$

If the `dgemmNN` runs at 19.61 GFLOPs/s ([Example A.4](#)), it reached an efficiency of

$$\frac{\text{attained performance}}{\text{peak performance}} = \frac{19.61 \text{ GFLOPs/s}}{20.8 \text{ GFLOPs/s}} \approx 94.27 \% .$$

There are many different ways to look at efficiency other than the ratio of attained performance to peak performance. Rewriting the definition of efficiency as

$$\begin{aligned} \text{efficiency} &= \frac{\text{attained performance}}{\text{peak performance}} \\ &= \frac{\text{cost/runtime}}{\text{cost/optimal runtime}} \\ &= \frac{\text{optimal runtime}}{\text{runtime}} , \end{aligned}$$

it is expressed as the ratio of the minimum time required to perform the operation's minimal FLOPs on the given hardware to the computation's runtime. If we reorganize it as

$$\text{efficiency} = \frac{\text{attained performance}}{\text{peak performance}}$$

$$\begin{aligned}
 &= \frac{\text{cost/runtime}}{\text{peak performance}} \\
 &= \frac{\text{cost}}{\text{runtime} \times \text{peak performance}} \\
 &= \frac{\text{cost}}{\text{available FLOPs}} ,
 \end{aligned}$$

it can be seen as the ratio of the operation's minimal FLOP-count to how many FLOPs the processor could theoretically perform during the computation's runtime.

Example A.9: Expressing compute-bound efficiency

In [Example A.8](#) the `dgemmNN` took 102 ms, while the SANDY BRIDGE-EP E5-2670 with a peak performance of 20.8 GFLOPs/s (TURBO BOOST disabled) could have performed the required $2 \times 1000^3 \text{ FLOPs} = 2 \cdot 10^9 \text{ FLOPs}$ in

$$\frac{2 \cdot 10^9 \text{ FLOPs}}{20.8 \text{ GFLOPs/s}} \approx 96.15 \text{ ms} .$$

Hence, the computation's efficiency can be computed as

$$\frac{\text{optimal runtime}}{\text{runtime}} = \frac{96.15 \text{ ms}}{102 \text{ ms}} \approx 94.26 \% .$$

We can also consider that in the 102 ms that the `dgemmNN` took, the SANDY BRIDGE core could have performed

$$102 \text{ ms} \times 20.8 \text{ GFLOPs/s} \approx 2.12 \cdot 10^9 \text{ FLOPs} .$$

Once again we obtain the same efficiency, as a FLOP-count ratio:

$$\frac{\text{cost}}{\text{available FLOPs}} = \frac{2 \cdot 10^9 \text{ FLOPs}}{2.12 \cdot 10^9 \text{ FLOPs}} \approx 94.26 \% .$$

A.5.2 Bandwidth-Bound Efficiency

A computation is bandwidth-bound on a hardware platform if the memory operations cannot load and store the involved data as fast as the processor's

floating-point units can process it, i.e., the memory bandwidth is the bottleneck and the compute units are partially idle. An operation is theoretically bandwidth-bound when

$$\text{arithmetic intensity} \leq \frac{\text{peak performance}}{\text{peak bandwidth}} .$$

Furthermore, a computation's *bandwidth-bound efficiency* is defined as

$$\text{bandwidth-bound efficiency} \stackrel{\text{def}}{=} \frac{\text{attained bandwidth}}{\text{peak bandwidth}} . \quad (\text{A.6})$$

A bandwidth-bound efficiency close to 1 indicates a good utilization of the processor's main-memory bandwidth, while smaller values signal underutilization.

Example A.10: Bandwidth-bound efficiency

The vector inner product $\alpha := -x^T y$ (`ddot`) with $x, y \in \mathbb{R}^{100\,000}$ has an arithmetic intensity of $\frac{1}{8}$ FLOPs/byte (Example A.3) and is thus clearly bandwidth-bound. If on one core of a SANDY BRIDGE-EP E5-2670, it attains a bandwidth of 11.49 GiB/s (Example A.5), relative to the processor's empirical peak bandwidth of 16.25 GiB/s (Example A.7), it performed at a bandwidth-bound efficiency of

$$\frac{\text{attained bandwidth}}{\text{peak bandwidth}} = \frac{11.49 \text{ GiB/s}}{16.25 \text{ GiB/s}} \approx 70.71 \% .$$

A.5.3 The Roofline Model

The *Roofline model* [88] plots the performance of computations (in GFLOPs/s) against their arithmetic intensity (in FLOPs/byte). In addition to data-points from measurements, two lines are added to such a plot to indicate the theoretically attainable performance depending on the arithmetic intensity: The product of peak bandwidth and arithmetic intensity (in units: $\text{GiB/s} \times \text{FLOPs/byte} = \text{GiFLOPs/s} \approx 0.93 \text{ GFLOPs/s}$) constitutes a straight line through the origin with the bandwidth as a gradient (visually: \nearrow) that

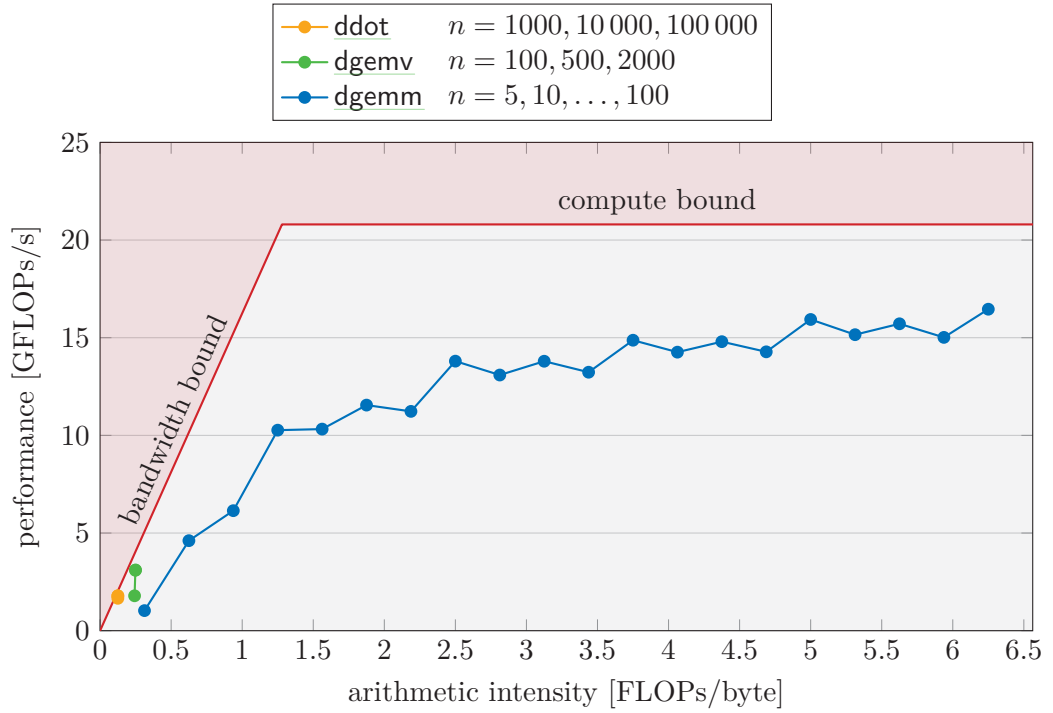


Figure A.1: `ddot`, `dgemv`, and `dgemm` in the Roofline Model.

(SANDY BRIDGE-EP E5-2670, 1 thread, OPENBLAS)

represents the bandwidth-bound performance limit; and the peak floating-point performance is a constant line (—). Together these two lines form the roofline-shaped performance limit (↗) that gives the visualization its name:

$$\text{performance limit} = \min \left(\begin{array}{c} \text{peak bandwidth} \times \text{arithmetic intensity,} \\ \text{peak performance} \end{array} \right). \quad (\text{A.7})$$

Comparing the attained performance of a computation to this limit yields the computation’s efficiency—bandwidth-bound below the left part of the “roof” and compute-bound below the right part.

Example A.11: The roofline model

Figure A.1 presents the Roofline model for one core of a SANDY BRIDGE-EP E5-2670. This processor has a single-core peak performance of

20.8 GFLOPs/cycle (TURBO BOOST disabled), and we use the measured single-core peak bandwidth of 16.25 GiB/s (Example A.7). Together these two factors impose the performance limit (—)

$$\min(16.25 \text{ GiB/s} \times \text{arithmetic intensity}, 20.8 \text{ GFLOPs/s})$$

Figure A.1 also contains the measured performance of representative BLAS Level 1, 2, and 3 operations, whose arithmetic intensity was determined in Example A.3.

- The vector inner product $\alpha := x^T y$ (`ddot`) with $x, y \in \mathbb{R}^n$ (—●—) has an arithmetic intensity of $\frac{1}{8}$ FLOPs/byte, making it clearly bandwidth-bound below the left part of the “roofline”. The attained (bandwidth-bound) efficiency, which is given by the ratio of the measured performance (—●—) to the attainable peak performance (—), is quite high at 87.93 %.
- The matrix-vector multiplication $y := A x + y$ (`dgemvN`) with $A \in \mathbb{R}^{n \times n}$ and $x, y \in \mathbb{R}^n$ (—●—) has an arithmetic intensity of $\approx \frac{1}{4}$ FLOPs/byte, making it also bandwidth-bound. The (bandwidth-bound) efficiency (—●— divided by —) is between 45.32 % (for $n = 100$) and 76.66 % (for $n = 2000$).
- The matrix-matrix multiplication $C := A B + C$ (`dgemmNN`) with $A, B, C \in \mathbb{R}^{n \times n}$ (—●—) has a higher arithmetic intensity of $\frac{n}{16}$ FLOPs/byte, which makes it theoretically compute-bound on our machine for $n \geq 21$. In the bandwidth-bound domain it reaches its peak (bandwidth-bound) efficiency (—●— divided by —) of 50.15 % at $n = 20$. Within the compute-bound domain, its (compute-bound) efficiency grows towards 74.32 % for our largest problem size $n = 100$. Beyond this size the efficiency keeps growing and converge to its peak of 93.70 % for matrices of size $n = 2000$.

A.6 Other Metrics

In addition to the fundamental metrics of workload, time, performance, and efficiency, a range of other metrics provides further insights into the execution and behavior of computations. For instance, many hardware events—such as various types of cache misses, interrupts, and branch prediction failures—can be counted via a processor’s performance counters, which are easily accessed through tools such as PAPI [28, 119] and INTELVTUNE [110]. While our performance measurement framework introduced in Section 2.2 also provides access to these counters, they only play a minor role throughout this work; the central metric for our modeling and prediction efforts of BLAS-based algorithms are runtime and its derivatives.

Another noteworthy class of performance metrics that play an increasingly important role quantify a computation’s energy consumption and efficiency. The commonly used metric of FLOPs/s/W for instance is used to rank the TOP500 [127] supercomputers according to their energy efficiency in the GREEN500 list [108].

B Dense Linear Algebra

Routines and Libraries

This appendix gives an overview of the core dense linear algebra libraries used throughout this work: BLAS and LAPACK.

- The BASIC LINEAR ALGEBRA SUBPROGRAMS (BLAS) [63, 40, 39] provide kernels for various vector and matrix multiplications, as well as triangular linear system solvers (back substitution).
- On top of BLAS, the LINEAR ALGEBRA PACKAGE (LAPACK) [16, 111] offers more advanced operations, such as matrix decompositions, inversions, linear-system and least-squares solvers, and eigensolvers.

While BLAS and LAPACK are sometimes referred to as “libraries”, they should be seen as standardized interface specifications with fully functional, yet unoptimized reference implementations.

In the following, [Section B.1](#) introduces the operand storage format expected by both BLAS and LAPACK, [Sections B.2](#) and [B.3](#) give an overview of these interfaces and their routines used in this work, and [Section B.4](#) discusses significant BLAS and LAPACK implementations.

B.1 Storage Format

This section describes how operands are stored and passed as arguments to BLAS and LAPACK routines. Note that due to the interfaces’ roots in FORTRAN, all arguments are passed by reference.

B.1.1 Scalars

Each scalar operand (e.g., $\alpha \in \mathbb{R}$) is passed as a single argument, (e.g., `double *alpha`). Complex scalars are stored as two consecutive elements of the basis data-type (`float` or `double`) that represent the real and imaginary parts.

B.1.2 Vectors

Each vector operand (e.g., $x \in \mathbb{R}^n$) is specified by three arguments:

- A size argument (e.g., `int *n`) determines the length of the vector. One size argument can describe multiple vectors (and/or matrices) with the same size.
- A data argument (e.g., `double *x`) points to the vector's first element in memory.
- An increment argument (e.g., `int *incx`) identifies the stride between consecutive elements of the vector. For instance, a contiguously stored vector has an increment of 1.

Note that most routines allow negative increments. In this case, the vector is stored in reverse, and the data argument points to the vector's last element—the first memory location.

To summarize, vector element x_i is stored at `x[i * incx]` if `incx` is positive and `x[(i - n + 1) * incx]` otherwise.

B.1.3 Matrices

Each matrix (e.g., $A \in \mathbb{R}^{m \times n}$) is specified by four arguments:

- Two size arguments (e.g., `int *m` and `int *n`) determine the matrix height (m) and width (n). One size argument can describe the dimensions of multiple matrices (and/or vectors), or both dimensions of a square matrix.

- A data argument (e.g., `double *A`) points to the first matrix element in memory (e.g., a_{00}). The following elements of the first column (e.g., a_{i0}) are stored consecutively in memory as vector with increment 1.
- A leading dimension argument (e.g., `int *ldA`) describes the distance in memory between matrix columns. It can hence be understood and used as the increment argument for the matrix rows as vectors. The term “leading dimension” comes from the concept that a referenced matrix is part of a larger, contiguously stored “leading” matrix. It allows to operate on sub-matrices or tensor panels as shown throughout this work. Leading dimensions must be at least equal to the height of the matrix (e.g., m).

To summarize, matrix element a_{ij} is stored at $A[i + j * ldA]$.

B.2 BASIC LINEAR ALGEBRA SUBPROGRAMS

The BASIC LINEAR ALGEBRA SUBPROGRAMS (BLAS) cover fundamental dense vector and matrix operations, such as various types of multiplications and triangular linear system solvers. BLAS is structured in three levels:

- BLAS Level 1 [63] provides vector operations, such as copying, scaling, additions, norms, and inner products.
- BLAS Level 2 [40] provides matrix-vector operations, such as outer products, matrix-vector multiplications and solvers for triangular linear systems.
- BLAS Level 3 [39] provides matrix-matrix operations, such as various multiplications, and solvers for triangular linear system with multiple right-hand-sides.

The following details the BLAS routines used throughout this work; a complete reference can be found online [96].

Note that for BLAS Level 2 and 3 kernels the minimal FLOP-counts assume that all scalars are $\alpha = \beta = 1$.

B.2.1 BLAS Level 1

dcopy(n, x, incx, y, incy)

double-precision vector copy

Operations

$y := \alpha x$

Arguments

n: dimension n

x: vector $x \in \mathbb{R}^n$

incx: increment for x

y: vector $y \in \mathbb{R}^n$

incy: increment for y

Minimal FLOP-count

0

Data volume

$2n$

Minimal data movement

$2n$

dswap(n, x, incx, y, incy)

double-precision vector swap

Operations

$x, y := y, x$

Arguments

n: dimension n

x: vector $x \in \mathbb{R}^n$

incx: increment for x

y: vector $y \in \mathbb{R}^n$

incy: increment for y

Minimal FLOP-count

0

Data volume

$2n$

Minimal data movement

$4n$

daxpy(n, alpha, x, incx, y, incy)

double-precision scaled vector addition

Operations

$y := \alpha x + y$

Arguments

n: dimension n

alpha: scalar α

x: vector $x \in \mathbb{R}^n$

incx: increment for x

y: vector $y \in \mathbb{R}^n$

incy: increment for y

Minimal FLOP-count

$2n$

Data volume

$2n$

Minimal data movement

$3n$

ddot(n , x , incx, y , incy)

double-precision inner vector product

Operations

$$\alpha := x^T x$$

Arguments

n : dimension n

x : vector $x \in \mathbb{R}^n$

incx: increment for x

y : vector $y \in \mathbb{R}^n$

incy: increment for y

Minimal FLOP-count

$2n$

Data volume

$2n$

Minimal data movement

$2n$

B.2.2 BLAS Level 2

dgemv(trans, m , n , alpha, A , ldA, x , incx, beta, y , incy)

double-precision matrix-vector product

Operations

$$y := \alpha A x + \beta y$$

$$y := \alpha A^T x + \beta y$$

Arguments

trans: A is transposed

m : dimension m

n : dimension n

alpha: scalar α

A : matrix $A \in \mathbb{R}^{m \times n}$

ldA: leading dimension for A

x : vector $x \in \begin{cases} \mathbb{R}^n & \text{if trans} = N \\ \mathbb{R}^m & \text{else} \end{cases}$

incx: increment for x

beta: scalar β

y : vector $y \in \begin{cases} \mathbb{R}^m & \text{if trans} = N \\ \mathbb{R}^n & \text{else} \end{cases}$

incy: increment for y

Minimal FLOP-count

$2mn$

Data volume

$mn + m$ if trans = N

$mn + n$ else

Minimal data movement

$mn + 2m$ if trans = N

$mn + 2n$ else

dger(m, n, alpha, x, incx, y, incy, A, ldA)

double-precision vector outer product

Operations

$$\boxed{A} := \alpha x y^T + \boxed{A}$$

Arguments

m: dimension m
n: dimension n
alpha: scalar α
x: vector $x \in \mathbb{R}^m$
incx: increment for x
y: vector $y \in \mathbb{R}^n$
incy: increment for y

A: matrix $\boxed{A} \in \mathbb{R}^{m \times n}$

ldA: leading dimension for \boxed{A}

Minimal FLOP-count

$$2mn$$

Data volume

$$mn + m + n$$

Minimal data movement

$$2mn + m + n$$

dtrsv(uplo, trans, diag, n, A, ldA, x, incX)

double-precision triangular linear system solve

Operations

$$x := \boxed{A}^{-1} x$$

$$x := \boxed{A}^{-T} x$$

Arguments

uplo: \boxed{A} is lower- or upper-triangular
trans: \boxed{A} is transposed
diag: \boxed{A} is unit triangular
n: dimension n
A: matrix $\boxed{A} \in \mathbb{R}^{n \times n}$
ldA: leading dimension for \boxed{A}

x: vector $x \in \mathbb{R}^n$

incX: increment for x

Minimal FLOP-count

$$n^2$$

Data volume

$$\frac{1}{2}n(n+1) + n$$

Minimal data movement

$$\frac{1}{2}n(n+1) + 2n$$

B.2.3 BLAS Level 3

dgemm(transA, transB, m, n, k, alpha, A, ldA, B, ldB, beta, C, ldC)

double-precision matrix-matrix product

Operations

$$\boxed{C} := \alpha \boxed{A} \boxed{B} + \beta \boxed{C}$$

$$\boxed{C} := \alpha \boxed{A} \boxed{B}^T + \beta \boxed{C}$$

$$\boxed{C} := \alpha \boxed{A}^T \boxed{B} + \beta \boxed{C}$$

$$\boxed{C} := \alpha \boxed{A}^T \boxed{B}^T + \beta \boxed{C}$$

Arguments

transA: $\begin{bmatrix} A \end{bmatrix}$ is transposed
 transB: $\begin{bmatrix} B \end{bmatrix}$ is transposed
 m: dimension m
 n: dimension n
 k: dimension k
 alpha: scalar α
 A: matrix

$$\begin{bmatrix} A \end{bmatrix} \in \begin{cases} \mathbb{R}^{m \times k} & \text{if transA} = \text{N} \\ \mathbb{R}^{k \times m} & \text{else} \end{cases}$$

 ldA: leading dimension for $\begin{bmatrix} A \end{bmatrix}$
 B: matrix

$$\begin{bmatrix} B \end{bmatrix} \in \begin{cases} \mathbb{R}^{k \times n} & \text{if transB} = \text{N} \\ \mathbb{R}^{n \times k} & \text{else} \end{cases}$$

ldB: leading dimension for $\begin{bmatrix} B \end{bmatrix}$

beta: scalar β

C: matrix $\begin{bmatrix} C \end{bmatrix} \in \mathbb{R}^{m \times n}$

ldC: leading dimension for $\begin{bmatrix} C \end{bmatrix}$

Minimal FLOP-count

$2mnk$

Data volume

$mk + kn + mn$

Minimal data movement

$mk + kn + 2mn$

dsymm(side, uplo, m, n, alpha, A, ldA, B, ldB, beta, C, ldC)

double-precision symmetric matrix-matrix product

Operations

$$\begin{aligned} \begin{bmatrix} C \end{bmatrix} &:= \alpha \begin{bmatrix} A \end{bmatrix} \begin{bmatrix} B \end{bmatrix} + \beta \begin{bmatrix} C \end{bmatrix} \\ \begin{bmatrix} C \end{bmatrix} &:= \alpha \begin{bmatrix} B \end{bmatrix} \begin{bmatrix} A \end{bmatrix} + \beta \begin{bmatrix} C \end{bmatrix} \end{aligned}$$

Arguments

side: $\begin{bmatrix} A \end{bmatrix}$ is on the left or right of $\begin{bmatrix} B \end{bmatrix}$
 uplo: $\begin{bmatrix} A \end{bmatrix}$ is in lower- or upper-triangular storage
 m: dimension m
 n: dimension n
 alpha: scalar α
 A: matrix

$$\begin{bmatrix} A \end{bmatrix} \in \begin{cases} \mathbb{R}^{m \times m} & \text{if side} = \text{L} \\ \mathbb{R}^{n \times n} & \text{else} \end{cases}$$

 ldA: leading dimension for $\begin{bmatrix} A \end{bmatrix}$
 B: matrix $\begin{bmatrix} B \end{bmatrix} \in \mathbb{R}^{m \times n}$

ldB: leading dimension for $\begin{bmatrix} B \end{bmatrix}$

beta: scalar β

C: matrix $\begin{bmatrix} C \end{bmatrix} \in \mathbb{R}^{m \times n}$

ldC: leading dimension for $\begin{bmatrix} C \end{bmatrix}$

Minimal FLOP-count

$2m^2n$ if side = L
 $2mn^2$ else

Data volume

$\frac{1}{2}m(m+1) + 2mn$ if side = L
 $\frac{1}{2}n(n+1) + 2mn$ else

Minimal data movement

$\frac{1}{2}m(m+1) + 3mn$ if side = L
 $\frac{1}{2}n(n+1) + 3mn$ else

dtrmm(side, uplo, transA, diag, m, n, alpha, A, ldA, B, ldB)

double-precision triangular matrix-matrix product

Operations

$$\begin{bmatrix} B \end{bmatrix} := \alpha \begin{bmatrix} A \end{bmatrix} \begin{bmatrix} B \end{bmatrix}$$

$$\begin{aligned} B &:= \alpha A^T B \\ B &:= \alpha A B \\ B &:= \alpha A^T B \\ B &:= \alpha B A \\ B &:= \alpha B A^T \\ B &:= \alpha B A \\ B &:= \alpha B A^T \end{aligned}$$

Arguments

side: A is on the left or right of B
 uplo: A is lower- or upper-triangular
 transA: A is transposed
 diag: A is unit triangular
 m: dimension m
 n: dimension n
 alpha: scalar α

A: matrix

$$A \in \begin{cases} \mathbb{R}^{m \times m} & \text{if side} = L \\ \mathbb{R}^{n \times n} & \text{else} \end{cases}$$

ldA: leading dimension for A

B: matrix $B \in \mathbb{R}^{m \times n}$

ldB: leading dimension for B

Minimal FLOP-count

$$\begin{aligned} m^2 n & \text{ if side} = L \\ mn^2 & \text{ else} \end{aligned}$$

Data volume

$$\begin{aligned} \frac{1}{2}m(m+1) + mn & \text{ if side} = L \\ \frac{1}{2}n(n+1) + mn & \text{ else} \end{aligned}$$

Minimal data movement

$$\begin{aligned} \frac{1}{2}m(m+1) + 2mn & \text{ if side} = L \\ \frac{1}{2}n(n+1) + 2mn & \text{ else} \end{aligned}$$

ssyrk(uplo, trans, n, k, alpha, A, ldA, beta, C, ldB)

single-precision symmetric rank-k update. See dsyrk.

dsyrk(uplo, trans, n, k, alpha, A, ldA, beta, C, ldB)

double-precision symmetric rank-k update

Operations

$$\begin{aligned} C &:= \alpha A A^T + C \\ C &:= \alpha A^T A + C \end{aligned}$$

Arguments

uplo: C has lower- or upper-triangular storage
 trans: A is transposed
 n: dimension n
 k: dimension k
 alpha: scalar α
 A: matrix

$$A \in \begin{cases} \mathbb{R}^{n \times k} & \text{if trans} = N \\ \mathbb{R}^{k \times n} & \text{else} \end{cases}$$

ldA: leading dimension for A

beta: scalar β

C: symmetric matrix $C \in \mathbb{R}^{n \times n}$

ldB: leading dimension for C

Minimal FLOP-count

$$n(n+1)k$$

Data volume

$$\frac{1}{2}n(n+1) + nk$$

Minimal data movement

$$n(n+1) + nk$$

cherk(uplo, trans, n, k, alpha, A, ldA, beta, C, ldB)

single-precision complex Hermitian rank- k update. See [dsyrk](#).

zherk(uplo, trans, n, k, alpha, A, ldA, beta, C, ldB)

double-precision complex Hermitian rank- k update. See [dsyrk](#).

dsyr2k(uplo, trans, n, k, alpha, A, ldA, B, ldB, beta, C, ldC)

double-precision symmetric rank- $2k$ update

Operations

$$\begin{aligned} C &:= \alpha A B^T + \alpha B A^T + C \\ C &:= \alpha A^T B + \alpha B^T A + C \end{aligned}$$

Arguments

uplo: C has lower- or upper-triangular storage

trans: A is transposed

n: dimension n

k: dimension k

alpha: scalar α

A: matrix

$$A \in \begin{cases} \mathbb{R}^{n \times k} & \text{if trans} = N \\ \mathbb{R}^{k \times n} & \text{else} \end{cases}$$

ldA: leading dimension for A

B: matrix

$$B \in \begin{cases} \mathbb{R}^{n \times k} & \text{if trans} = N \\ \mathbb{R}^{k \times n} & \text{else} \end{cases}$$

ldB: leading dimension for B

beta: scalar β

C: symmetric matrix $C \in \mathbb{R}^{n \times n}$

ldC: leading dimension for C

Minimal FLOP-count

$$2n(n+1)k$$

Data volume

$$\frac{1}{2}n(n+1) + 2nk$$

Minimal data movement

$$n(n+1) + 2nk$$

strsm(side, uplo, transA, diag, m, n, alpha, A, ldA, B, ldB)

single-precision triangular linear system solve with multiple right hand sides.

See [dtrsm](#).

dtrsm(side, uplo, transA, diag, m, n, alpha, A, ldA, B, ldB)

double-precision triangular linear system solve with multiple right hand sides

Operations

$$B := \alpha A^{-1} B$$

$$B := \alpha A^{-T} B$$

$$B := \alpha A^{-1} B$$

$$B := \alpha A^{-T} B$$

$$B := \alpha B A^{-1}$$

$$\begin{aligned} B &:= \alpha B A^{-T} \\ B &:= \alpha B A^{-1} \\ B &:= \alpha B A^{-T} \end{aligned}$$

Arguments

side: A is on the left or right of B
 uplo: A is lower- or upper-triangular
 transA: A is transposed
 diag: A is unit triangular
 m: dimension m
 n: dimension n
 alpha: scalar α
 A: matrix
 $A \in \begin{cases} \mathbb{R}^{m \times m} & \text{if side} = \text{L} \\ \mathbb{R}^{n \times n} & \text{else} \end{cases}$

ldA: leading dimension for A
 B: matrix $B \in \mathbb{R}^{m \times n}$
 ldB: leading dimension for B

Minimal FLOP-count

$m^2 n$ if side = L
 mn^2 else

Data volume

$\frac{1}{2}m(m+1) + mn$ if side = L
 $\frac{1}{2}n(n+1) + mn$ else

Minimal data movement

$\frac{1}{2}m(m+1) + 2mn$ if side = L
 $\frac{1}{2}n(n+1) + 2mn$ else

ctrsm(side, uplo, transA, diag, m, n, alpha, A, ldA, B, ldB)

single-precision complex triangular linear system solve with multiple right hand sides. See [dtrsm](#).

ztrsm(side, uplo, transA, diag, m, n, alpha, A, ldA, B, ldB)

double-precision complex triangular linear system solve with multiple right hand sides. See [dtrsm](#).

B.3 LINEAR ALGEBRA PACKAGE

The LINEAR ALGEBRA PACKAGE (LAPACK) is a collection of advanced dense matrix operations, such as various factorizations and equation solvers. A large portion of LAPACK casts the majority of its computations in terms of BLAS routines; it thereby extends the high performance of BLAS implementations to its operations.

The following gives an overview of the LAPACK routines employed and studied this work; a complete reference can be found online [111].

ilaenv(ispec, name, opts, n1, n2, n3, n4)*query algorithmic parameters (LAPACK internal)***Note**

Provides various algorithm parameters (e.g., block sizes). Should be modified for each architecture and BLAS implementation to optimize performance.

Arguments

ispec: queried parameter
(e.g., 1 for block size)

name: name of calling routine
opts: calling routine's concatenated flag arguments
n1: problem size 1
n2: problem size 2
n3: problem size 3
n4: problem size 4
(according to calling routine)

dlauum(uplo, n, A, ldA, info)*double-precision triangular matrix multiplication with its transpose***Operations**

$$\begin{aligned} A &:= L^T L \\ A &:= U U^T \end{aligned}$$

Arguments

uplo: L is lower- or upper-triangular
n: dimension n
A: matrix $A \in \mathbb{R}^{n \times n}$
input: L
ldA: leading dimension of A

info: return error and info codes

Minimal FLOP-count

$$\frac{1}{6}n(n+1)(2n+1) \approx \frac{n^3}{3}$$

Data volume

$$\frac{1}{2}n(n+1) \approx \frac{n^2}{2}$$

Minimal data movement

$$n(n+1) \approx n^2$$

dlauu2(uplo, n, A, ldA, info)*unblocked double-precision triangular matrix multiplication with its transpose.*See [dlauum](#).**dsygst(itype, uplo, n, A, ldA, B, ldB, info)***double-precision symmetric linear system solve***Operations**

$$\begin{aligned} A &:= B^{-1} A B^{-T} \\ A &:= B^{-T} A B^{-1} \\ A &:= B^T A B \end{aligned}$$

$$A := B A B^T$$

Arguments

itype: whether to invert B
uplo: B is lower- or upper-triangular

n: dimension n
A: matrix $\underline{A} \in \mathbb{R}^{n \times n}$
ldA: leading dimension of \underline{A}
B: matrix $\underline{B} \in \mathbb{R}^{n \times n}$
ldB: leading dimension of \underline{B}
info: return error and info codes

Minimal FLOP-count

$$n * (n + 1)^2 \approx n^3$$

Data volume

$$n(n + 1) \approx n^2$$

Minimal data movement

$$\frac{3}{2}n(n + 1) \approx \frac{3}{2}n^2$$

dsygs2(itype, uplo, n, A, ldA, B, ldB, info)

unblocked double-precision symmetric linear system solve. See [dsygst](#).

dtrtri(uplo, diag, n, A, ldA, info)

double-precision triangular matrix inversion

Operations

$$\underline{A} := \underline{A}^{-1}$$

$$\underline{A} := \underline{A}^{-1}$$

Arguments

uplo: \underline{A} is lower- or upper-triangular
diag: \underline{A} is unit diagonal
n: dimension n
A: matrix $\underline{A} \in \mathbb{R}^{n \times n}$
output: \underline{A}^{-1}

ldA: leading dimension of \underline{A}
info: return error and info codes

Minimal FLOP-count

$$\frac{1}{6}n(n + 1)(2n + 1) \approx \frac{n^3}{3}$$

Data volume

$$\frac{1}{2}n(n + 1) \approx \frac{n^2}{2}$$

Minimal data movement

$$n(n + 1) \approx n^2$$

dtrti2(uplo, diag, n, A, ldA, info)

unblocked double-precision triangular matrix inversion. See [dtrtri](#).

dpotrf(uplo, n, A, ldA, info)

double-precision Cholesky decomposition

Operations

$$\underline{L} \underline{L}^T := \underline{A}$$

$$\underline{U}^T \underline{U} := \underline{A}$$

Arguments

uplo: \underline{A} is in lower- or upper-triangular storage
n: dimension n
A: SPD matrix $\underline{A} \in \mathbb{R}^{n \times n}$

output: \underline{L} or \underline{U}
 ldA: leading dimension of \underline{A}
 info: return error and info codes

Minimal FLOP-count

$$\frac{1}{6}n(n+1)(2n+1) \approx \frac{n^3}{3}$$

Data volume

$$\frac{1}{2}n(n+1) \approx \frac{n^2}{2}$$

Minimal data movement

$$n(n+1) \approx n^2$$

spotf2(uplo, n, A, ldA, info)

unblocked single-precision Cholesky decomposition. See [dpotrf](#).

dpotf2(uplo, n, A, ldA, info)

unblocked double-precision Cholesky decomposition. See [dpotrf](#).

cpotf2(uplo, n, A, ldA, info)

unblocked single-precision complex Cholesky decomposition. See [dpotrf](#).

zpotf2(uplo, n, A, ldA, info)

unblocked double-precision complex Cholesky decomposition. See [dpotrf](#).

dgetrf(m, n, A, ldA, ipiv, info)

double-precision LU decomposition with partial pivoting

Note

The matrix \underline{P} is represented as a list of single-row swaps; see [dlaswp](#).

Operations

$$\underline{P} \underline{L} \underline{U} := \underline{A}$$

Arguments

m: dimension m
 n: dimension n
 A: matrix $\underline{A} \in \mathbb{R}^{m \times n}$
 output: \underline{L} and \underline{U}

ldA: leading dimension of \underline{A}
 ipiv: permutation matrix \underline{P}
 info: return error and info codes

Minimal FLOP-count

$$\frac{2}{3}mn \min(m, n)$$

Data volume

$$mn$$

Minimal data movement

$$2mn$$

dgetf2(m, n, A, ldA, ipiv, info)

unblocked double-precision LU decomposition with partial pivoting. See [dgetrf](#).

dlaswp(n, A, ldA, k1, k2, ipiv, incx)

double-precision multiplication with permutation matrix from the left

Note

The matrix P is represented as a list of indices ipiv: For each $i = k_1, \dots, k_2$, row i is swapped with row $\text{ipiv}[i]$.

Operations

$$A := P A$$

Arguments

n: dimension n
A: matrix $A \in \mathbb{R}^{m \times n}$
ldA: leading dimension of A
k1: index k_1
k2: index k_2
ipiv: permutation matrix P
incx: vector increment for P

dgeqrf(m, n, A, ldA, tau, Work, lWork, info)

double-precision QR decomposition

Note

The matrix Q is represented as a series of elementary reflectors in A and scalar factors in τ .

Operations

$$Q R := A$$

Arguments

m: dimension m
n: dimension n
A: matrix $A \in \mathbb{R}^{m \times n}$
output: part of Q and R
ldA: leading dimension of A

tau: vector $\tau \in \mathbb{R}^{\min(m,n)}$
Work: auxiliary buffer $W \in \mathbb{R}^l$
lWork: buffer size $l \geq n$
info: return error and info codes

Minimal FLOP-count

$$\approx \begin{cases} 2m^2(n - \frac{1}{3}m) & \text{if } m < n \\ 2n^2(m - \frac{1}{3}n) & \text{else} \end{cases}$$

Data volume

$$mn + \min(m, n)$$

Minimal data movement

$$2mn + \min(m, n)$$

dgeqr2(m, n, A, ldA, tau, Work, info)

unblocked double-precision QR decomposition. See [dgeqrf](#).

dlarft(direct, storev, n, k, alpha, V, ldV, tau, T, ldT)*double-precision construction of the triangular factor for a block reflector***Arguments**

direct: order of elementary reflectors
 storev: elementary reflectors are stored
 as rows or columns
 n: dimension n
 k: dimension k
 alpha: scalar α

V: matrix

$$V \in \begin{cases} \mathbb{R}^{n \times k} & \text{if storev} = \text{C} \\ \mathbb{R}^{k \times n} & \text{else} \end{cases}$$

 ldV: leading dimension for V
 tau: vector $\tau \in \mathbb{R}^k$
 T: triangular factor $T \in \mathbb{R}^{k \times k}$
 ldT: leading dimension for T

dlarfb(side, trans, direct, storev, n, m, k, alpha, V, ldV, T, ldT, C, IdC, Work, IdWork)*double-precision block reflector application to a matrix***Arguments**

side: H is applied from the left or
 right
 trans: H is transposed
 direct: order of elementary reflectors
 storev: elementary reflectors stored as
 rows or columns
 m: dimension m
 n: dimension n
 k: dimension k
 alpha: scalar α

V: matrix

$$V \in \begin{cases} \mathbb{R}^{m \times k} & \text{if storev} = \text{C} \\ & \text{and side} = \text{L} \\ \mathbb{R}^{n \times k} & \text{if storev} = \text{C} \\ & \text{and side} = \text{R} \\ \mathbb{R}^{k \times m} & \text{if storev} = \text{R} \\ & \text{and side} = \text{L} \\ \mathbb{R}^{k \times n} & \text{if storev} = \text{R} \\ & \text{and side} = \text{R} \end{cases}$$

 ldV: leading dimension for V
 T: triangular factor $T \in \mathbb{R}^{k \times k}$
 ldT: leading dimension for T
 C: triangular factor $C \in \mathbb{R}^{m \times n}$
 IdC: leading dimension for C
 Work: auxiliary buffer

$$W \in \begin{cases} \mathbb{R}^{n \times k} & \text{if side} = \text{L} \\ \mathbb{R}^{m \times k} & \text{else} \end{cases}$$

 IdWork: leading dimension for W

dtrsyl(tranA, tranB, isgn, m, n, A, ldA, B, ldB, C, ldC, scale, info)

double-precision triangular Sylvester equation solver

Operations

solve for X :

$$\begin{aligned} \nabla A X + X B &= \gamma C \\ \nabla A X + X B^T &= \gamma C \\ \nabla A^T X + X B &= \gamma C \\ \nabla A^T X + X B^T &= \gamma C \\ \nabla A X - X B &= \gamma C \\ \nabla A X - X B^T &= \gamma C \\ \nabla A^T X - X B &= \gamma C \\ \nabla A^T X - X B^T &= \gamma C \end{aligned}$$

Arguments

tranA: ∇A is transposed
 tranB: B is transposed
 isgn: sign in the equation
 m: dimension m
 n: dimension n

A: matrix $\nabla A \in \mathbb{R}^{m \times m}$
 ldA: leading dimension of ∇A
 B: matrix $B \in \mathbb{R}^{n \times n}$
 ldB: leading dimension of B
 C: matrix $C \in \mathbb{R}^{m \times n}$
 output: X
 ldC: leading dimension of C
 scale: output scalar γ
 info: return error and info codes

Minimal FLOP-count

$$mn(m + n + 4)$$

Data volume

$$\frac{1}{2}m(m + 1) + \frac{1}{2}n(n + 1) + mn$$

Minimal data movement

$$\frac{1}{2}m(m + 1) + \frac{1}{2}n(n + 1) + 2mn$$

B.4 Implementations

All dense linear algebra algorithms and libraries—including LAPACK—only reach high-performance through optimized BLAS implementations. While some highly tuned BLAS implementations are open-source (e.g., `OPENBLAS` and `BLIS`), others are provided by hardware vendors (e.g., `MKL` and `ACCELERATE`). This section gives an overview of the implementations used throughout this work.

Reference Implementations

The BLAS and LAPACK reference implementations [96, 111] are fully functional and well-documented and thus of great value as references for routine interfaces and semantics. However, on their own they only attain poor performance, and should therefore not be used in production codes.

All routines in the BLAS reference implementation are single-threaded and unoptimized. The central kernel `dgemm`, for instance, is realized as a simple triple loop that reaches around 6 % of modern processors’ single-threaded theoretical peak performance—optimized implementations are commonly $15\times$ faster on a single core and provide excellent multi-threaded scalability.

Since LAPACK primarily relies on a tuned BLAS implementation for speed, the reference implementation can in principle reach good performance. However, as its documentation states, this requires careful tuning of its block sizes, whose default values are generally too low on contemporary processors. Optimized implementations may further improve LAPACK’s performance through faster algorithms, tuned unblocked kernels (e.g., `dtrti2`, `dpotf2`), and algorithm-level parallelism (e.g., task-based algorithms-by-blocks).

Throughout this work, we use reference BLAS and LAPACK version 3.5.0.

OPENBLAS

`OPENBLAS` [118] is a high-performance open-source BLAS and LAPACK implementation that is currently developed and maintained at the MAS-

SACHUSETTS INSTITUTE OF TECHNOLOGY. It provides optimized and multi-threaded BLAS kernels for a wide range of architectures, and offers tuned version of core LAPACK routines, such as the `dlaaum`, `dtrtri`, `dpotrf`, and `dgetrf`. OPENBLAS is based on the discontinued GOTOBLAS2, adopting its approach and much of its source-code; it includes assembly kernels for more recent architectures, such as SANDY BRIDGE and HASWELL, as well AMD processors.

Throughout this work, we use OPENBLAS version 0.2.15.

BLIS

The BLAS-LIKE LIBRARY INSTANTIATION SOFTWARE (BLIS) [81, 82, 73, 97] is a fairly recent framework for dense linear algebra libraries that is actively developed at the UNIVERSITY OF TEXAS AT AUSTIN. While it comes with its own API, which is a superset, generalization, and extension of the BLAS, it contains a compatibility layer offering the original de-factor standard BLAS interface. BLIS builds upon the GOTOBLAS approach, yet restructures and solidifies it to make all but a tiny “micro-kernel” architecture-independent. While its performance is so far generally lower than that of OPENBLAS (see examples in Section 3.1), its ambitious goal is to significantly speed up both the development of new application-specific kernels, and the adaptation to other architectures.

Although multi-threading was introduced into BLIS [73] soon after its inception, its flexible threading model lacked a simple end-user interface (such as following the environment variable `OMP_NUM_THREADS`) until November 2016 (commit 6b5a403). As a result, we only presents single-threaded results for BLIS.

Throughout this work we use BLIS version 0.2.0.

MKL

INTEL’S MATH KERNEL LIBRARY (MKL) [109] is a high-performance library

for INTEL processors that covers BLAS and LAPACK, as well as other high-performance computations, such as for Fast Fourier Transforms (FFT) and Deep Neural Networks (DNN). While MKL is a closed-source library, it recently began offering free developer licenses. In terms of performance, it is in most scenarios superior to open-source libraries such as OPENBLAS and BLIS (see examples in [Section 3.1](#)).

Throughout this work we use MKL version 11.3.

ACCELERATE

APPLE’s framework ACCELERATE [\[105\]](#) is a high-performance library that ships with MACOS and, among others, provides full BLAS and LAPACK functionality. Its performance is for many cases comparable to OPENBLAS or slightly better.

Other Implementations

The following notable BLAS and LAPACK implementations are not used throughout this work:

- The AUTOMATICALLY TUNED LINEAR ALGEBRA SOFTWARE (ATLAS) [\[85, 87, 86, 95\]](#) is a high-performance BLAS implementation that relies on auto-tuning. While ATLAS kernels typically don not reach the performance of hand-tuned implementations such as OPENBLAS, BLIS, and MKL, it provides good performance for new and exotic architectures with little effort.
- GOTOBLAS2 [\[50, 51, 107\]](#) is a high-performance BLAS implementation that was developed at the TEXAS ADVANCED COMPUTING CENTER. Since its discontinuation, much of its code-base was picked up by its successor OPENBLAS in 2011, and its approach was refined and generalized in BLIS.

- IBM's ENGINEERING AND SCIENTIFIC SUBROUTINE LIBRARY (ESSL) [104] provides a high-performance BLAS implementation and parts of LAPACK for POWER-based systems, such as BLUE GENE supercomputers.

C Hardware

This appendix gives an overview of the processors used throughout this work and their relevant properties.

Note that, while the single-threaded peak performance is, where appropriate, based on the processors' maximum turbo frequency, the multi-threaded peak performance is instead computed from the base frequency. Furthermore, we only list the vector instructions that allow to reach a processor's theoretical peak performance.

C.1 HARPERTOWN E5450

http://ark.intel.com/products/33083/Intel-Xeon-Processor-E5450-12M-Cache-3_00-GHz-1333-MHz-FSB

Our HARPERTOWN E5450s were part of our compute cluster. Because they were disposed of in mid 2016, they are only used in a part of this work's performance analyses.

Name	INTEL® XEON® PROCESSOR E5450
Codename	HARPERTOWN
Lithography	45 nm
Release	Q4 2007
Cores / Threads	4 / 4
Base Frequency	3.00 GHz
Peak Performance	12 GFLOPs/s (single-threaded) 48 GFLOPs/s (all cores)
Peak Bandwidth	10.6 GB/s
L2 cache	6 MiB <i>per 2 cores</i> , 24-way set associative
L1d cache	32 KiB per core, 8-way set-associative

Vector Instructions	1 SSE FMUL + 1 SSE FADD per cycle = 4 FLOPs/cycle
---------------------	--

C.2 SANDY BRIDGE-EP E5-2670

http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI

Our SANDY BRIDGE E5-2680 v2s are part of our compute cluster. INTEL TURBO BOOST is disabled on these machines unless otherwise stated.

Name	INTEL® XEON® PROCESSOR E5-2670
Codename	SANDY BRIDGE-EP
Lithography	32 nm
Release	Q1 2012
Cores / Threads	8 / 16
Base Frequency	2.60 GHz
Max Turbo Frequency	3.30 GHz (<i>disabled unless otherwise stated</i>)
Peak Performance	20.8 GFLOPs/s (single-threaded) 166.4 GFLOPs/s (all cores)
Peak Bandwidth	51.2 GB/s
L3 cache	20 MiB shared, 20-way set associative
L2 cache	256 KiB per core, 8-way set associative
L1d cache	32 KiB per core, 8-way set-associative
Vector Instructions	1 AVX FMUL + 1 AVX FADD per cycle = 8 FLOPs/cycle

C.3 IVY BRIDGE-EP E5-2680 v2

http://ark.intel.com/products/75277/Intel-Xeon-Processor-E5-2680-v2-25M-Cache-2_80-GHz

Our IVY BRIDGE E5-2680 v3s are part of our compute cluster.

Name	INTEL® XEON® PROCESSOR E5-2680 v2
Codename	IVY BRIDGE-EP
Lithography	22 nm
Release	Q3 2013

Cores / Threads	10 / 20
Base Frequency	2.80 GHz
Max Turbo Frequency	3.60 GHz
Peak Performance	28.8 GFLOPs/s (single-threaded) 224 GFLOPs/s (all cores)
Peak Bandwidth	59.7 GB/s
L3 cache	25 MiB shared, 20-way set associative
L2 cache	256 KiB per core, 8-way set associative
L1d cache	32 KiB per core, 8-way set-associative
Vector Instructions	1 AVX FMUL + 1 AVX FADD per cycle = 8 FLOPs/cycle

C.4 HASWELL-EP E5-2680 v3

http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz

Our HASWELL-EP E5-2680 v3s are part of our compute cluster.

Name	INTEL® XEON® PROCESSOR E5-2680 v3
Codename	HASWELL-EP
Lithography	22 nm
Release	Q3 2014
Cores / Threads	12 / 24
Base Frequency	2.50 GHz
Max Turbo Frequency	3.30 GHz
Peak Performance	52.8 GFLOPs/s (single-threaded) 480 GFLOPs/s (all cores)
Peak Bandwidth	68 GB/s
L3 cache	30 MiB shared, 20-way set associative
L2 cache	256 KiB per core, 8-way set associative
L1d cache	32 KiB per core, 8-way set-associative
Vector Instructions	2 AVX FMA per cycle = 16 FLOPs/cycle

C.5 BROADWELL I7-5557U

https://ark.intel.com/products/84993/Intel-Core-i7-5557U-Processor-4M-Cache-up-to-3_40-GHz

Our BROADWELL I7-5557U is part of a MACBOOK PRO.

Name	INTEL® CORE™ I7-5557U PROCESSOR
Codename	BROADWELL-U
Lithography	14 nm
Release	Q1 2015
Cores / Threads	2 / 4
Base Frequency	3.10 GHz
Max Turbo Frequency	3.40 GHz
Peak Performance	54.4 GFLOPs/s (single-threaded) 99.2 GFLOPs/s (all cores)
Peak Bandwidth	25.6 GB/s
L3 cache	4 MiB shared, 16-way set associative
L2 cache	256 KiB per core, 8-way set associative
L1d cache	32 KiB per core, 8-way set-associative
Vector Instructions	2 AVX FMA per cycle = 16 FLOPs/cycle

List of Examples

1.1	Blocked algorithms for the Cholesky decomposition	4
1.2	Performance of alternative algorithms	6
1.3	Influence of the block size on performance	7
1.4	Tensor contraction algorithms	10
1.5	Performance of contraction algorithms	13
2.1	Library initialization overhead	24
2.2	Influence of background noise	25
2.3	TURBO BOOST	27
2.4	Performance levels	29
2.5	Thread pinning	30
2.6	NUMA effects	32
2.7	Caching	33
2.8	The SAMPLER	36
2.9	ELAPS installation	39
2.10	ELAPS workflow	39
3.1	Argument types	45
3.2	Flag arguments	47
3.3	Scalar arguments	49
3.4	Aligning leading dimensions to cache-lines	51
3.5	Cache conflict misses caused by leading dimensions	54
3.6	Increment arguments in BLAS Level 1	56
3.7	Increment arguments in BLAS Level 2	57
3.8	Small variations of size arguments	58

List of Examples

3.9	Polynomial fitting for size arguments	59
3.10	Data arguments	62
3.11	Sampling point distributions	68
3.12	Polynomial basis functions	70
3.13	Adaptive refinement	73
3.14	Model accuracy	78
4.1	Runtime prediction	86
4.2	Prediction summary statistics	88
4.3	Performance and efficiency predictions	89
4.4	Algorithm performance measurements	90
4.5	Prediction error	91
4.6	Block size trade-off	124
5.1	Algorithm-aware timings	156
6.1	Contraction algorithm for <code>dgemm_{NN}</code>	161
6.2	<code>dgemm</code> -based algorithms for $C_{abc} := A_{ai}B_{ibc}$	162
6.3	Other algorithms for $C_{abc} := A_{ai}B_{ibc}$	163
6.4	Loop-independent operand	170
6.5	Loop-dependent operand	170
6.6	No loops remaining	171
6.7	Cache access emulation	172
6.8	Prefetching and shared cache-lines	174
6.9	Cache emulation with prefetch distances	174
6.10	Benchmarks for prefetch failures	177
6.11	First loop iterations	178
A.1	Minimal FLOP-counts	193
A.2	Data volume and movement	195
A.3	Arithmetic intensity	196
A.4	Performance	198
A.5	Attained bandwidth	199

A.6	Peak floating-point performance	200
A.7	Peak bandwidth	201
A.8	Compute-bound efficiency	203
A.9	Expressing compute-bound efficiency	204
A.10	Bandwidth-bound efficiency	205
A.11	The roofline model	206

List of Figures

1.1	Blocked algorithms for the lower-triangular Cholesky decomposition.	5
1.2	Performance of the three blocked Cholesky decomposition algorithms.	7
1.3	Performance of the blocked Cholesky decompositions algorithm 3 for varying block sizes.	8
1.4	Sample of algorithms for the tensor contraction $C_{abc} := A_{ai}B_{ibc}$	11
1.5	Performance of tensor contraction algorithms.	12
2.1	Runtime fluctuations <code>dgemm_{NN}</code> caused by background processes and system noise.	26
2.2	Effect of TURBO BOOST on the runtime of <code>dgemm_{NN}</code>	28
2.3	Varying runtime for a skewed <code>dgemm_{NN}</code> over a period of time.	29
2.4	Effects of thread pinning on the compute-bound efficiency of a multi-threaded <code>dgemm_{TN}</code>	31
2.5	Setting up an ELAPS experiment in the PLAYMAT via X11.	40
2.6	The ELAPS VIEWER showing a performance plot.	41
3.1	Runtime of <code>dtrsm</code> as a function of its flag arguments.	48
3.2	Runtime of <code>dtrsm_{LLNN}</code> with different values for α	50
3.3	Runtime of <code>dtrsm</code> as a function of its leading dimension arguments on a small scale.	52
3.4	Runtime of <code>dtrsm_{LLNN}</code> as a function of its leading dimension arguments on a large scale.	54

List of Figures

3.5	Runtime of <code>daxpy</code> and <code>dtrsv_{LLNN}</code> as a function of their increment arguments.	56
3.6	Runtime of <code>dtrsm_{LLNN}</code> as a function of its size arguments on a small scale.	58
3.7	Runtime and error of piecewise cubic polynomial fits <code>dtrsm_{LLNN}</code>	60
3.8	Runtime of <code>dtrsm_{LLNN}</code> with in-cache and out-of-cache operands.	62
3.9	Structure of the performance models.	65
3.10	Sampling point distributions and reuse.	68
3.11	Modeling through adaptive refinement for <code>dtrsm_{LLNN}</code>	74
3.12	Accuracy and structure of models for <code>dtrsm_{LLNN}</code>	79
3.13	Model configuration trade-off in accuracy versus cost and steps towards selecting a default configuration.	81
4.1	Blocked algorithm 3 for the lower-triangular Cholesky decomposition.	92
4.2	Measurements and predictions for the Cholesky decomposition.	93
4.3	Prediction accuracy for the Cholesky decomposition.	95
4.4	Predictions and prediction accuracy for the Cholesky decomposition with varying block size.	96
4.5	Prediction accuracy for the Cholesky decomposition.	98
4.6	Predictions and prediction accuracy for the Cholesky decomposition with different data-types.	100
4.7	Predictions and prediction accuracy for the Cholesky decomposition with multi-threaded OPENBLAS.	101
4.8	LAPACK's blocked algorithms for <code>dlauum_L</code> , <code>dsygst_{1L}</code> , <code>dtrtri_{LN}</code> , <code>dpotrf_L</code> , and <code>dgetrf</code>	103
4.9	LAPACK's blocked algorithm for <code>dgeqrf</code>	106
4.10	Single-threaded prediction accuracy for LAPACK algorithms.	107
4.11	Multi-threaded prediction accuracy for LAPACK algorithms.	110
4.12	Performance measurements and predictions for the blocked Cholesky decomposition algorithms in lower-triangular storage.	113
4.13	Blocked algorithms for the inversion of a lower-triangular matrix.	115

4.14	Performance measurements and predictions for the eight blocked lower-triangular inversion algorithms.	116
4.15	Blocked algorithms solving the triangular Sylvester equation with 1×3 and 3×1 matrix partitionings.	119
4.16	Sample of blocked algorithms solving the triangular Sylvester equation with 3×3 matrix partitionings.	120
4.17	Performance predictions and measurements for the blocked triangular Sylvester equation solvers.	122
4.18	Breakdown of the blocked Cholesky decomposition algorithm 3 in terms of kernel runtime and performance.	124
4.19	Model-based block size optimization and empirical optima for the Cholesky decomposition algorithm 3.	126
4.20	Predicted and empirical optimal block sizes and prediction yields for the Cholesky decomposition algorithm 3.	127
4.21	Predicted and empirical optimal block sizes and prediction yields for the inversion of a lower-triangular matrix algorithm 3. . . .	129
4.22	Predicted and empirical optimal block sizes and prediction yields for <code>dsygst_{1L}</code> , <code>dgetrf</code> , and <code>dgeqrf</code>	130
5.1	LAPACK's blocked algorithm for <code>dgeqrf</code>	137
5.2	In-algorithm timings and error of repeated execution timings with respect to these for the 1873 kernel invocations within <code>dgeqrf</code>	138
5.3	Error of in- and out-of-cache timings with respect to in-algorithm timings for <code>dgeqrf</code>	140
5.4	Error of our initial and splitting estimates with respect to in-algorithm timings for <code>dgeqrf</code>	142
5.5	Smoothing function and error of final estimates with respect to in-algorithm timings for <code>dgeqrf</code>	143
5.6	LAPACK's blocked algorithm for the upper-triangular Cholesky decomposition <code>dpotrf_U</code> and in-algorithm timings.	146
5.7	Error of our final estimates with respect to in-algorithm timings for the Cholesky decomposition <code>dpotrf_U</code>	147

List of Figures

5.8	LAPACK's blocked algorithm for the inversion of a lower-triangular matrix <code>dtrtri_{LN}</code> and in-algorithms timings.	149
5.9	Error of our final estimates with respect to in-algorithm timings for the inversion of a lower-triangular matrix <code>dtrtri_{LN}</code>	150
5.10	Error of out-of-cache timings with respect to in-algorithm timings for <code>dtrtri_{LN}</code> and <code>dgetrf</code>	151
5.11	Error for attempted in-cache timings with respect to in-algorithm timings for <code>dtrtri_{LN}</code> and <code>dgetrf</code>	153
5.12	Error for attempted in-cache timings with respect to in-algorithm timings on a SANDY BRIDGE-EP <u>E5-2670</u> with TURBO BOOST enabled.	154
5.13	Basic operand regions accessed for attempted in-cache setups.	155
5.14	Error for algorithm-aware timings with respect to in-algorithm timings.	156
6.1	Contraction algorithms for $C_{abc} := A_{ai}B_{ibc}$ based on <code>dgemm</code>	163
6.2	Sample of contraction algorithms for $C_{abc} := A_{ai}B_{ibc}$ based on BLAS Level 2.	164
6.3	Sample of contractions algorithms for $C_{abc} := A_{ai}B_{ibc}$ based on BLAS Level 1.	165
6.4	Performance measurements of algorithms for $C_{abc} := A_{ai}B_{ibc}$ based on BLAS Level 2 and 3.	167
6.5	Performance predictions for $C_{abc} := A_{ai}B_{ibc}$ based on repeated execution.	168
6.6	Performance predictions for $C_{abc} := A_{ai}B_{ibc}$ with cache emulation based on access distances.	173
6.7	Performance predictions for $C_{abc} := A_{ai}B_{ibc}$ with cache emulation including prefetch distances.	176
6.8	Performance predictions for $C_{abc} := A_{ai}B_{ibc}$ accounting for prefetching failures.	178
6.9	Final performance predictions for $C_{abc} := A_{ai}B_{ibc}$	179

6.10	Performance predictions and measurements for $C_{abc} := A_{ai}B_{ibc}$ with $a = b = c = 128$ fixed.	180
6.11	<u>dgemv</u> -based algorithms for $C_a := A_{iaj}B_{ji}$	181
6.12	Performance predictions and measurements for $C_a := A_{iaj}B_{ji}$	182
6.13	<u>dgemm</u> -based algorithms for $C_{abc} := A_{ija}B_{jbic}$	183
6.14	Performance predictions and measurements for $C_{abc} := A_{ija}B_{jbic}$	184
6.15	Performance predictions and measurements for $C_{abc} := A_{ija}B_{jbic}$ on 10 cores.	185
6.16	Speedup of predictions over algorithm executions for $C_{abc} :=$ $A_{ai}B_{ibc}$	186
A.1	<u>ddot</u> , <u>dgemv</u> , and <u>dgemm</u> in the Roofline Model.	206

List of Tables

2.1	BLAS library initialization overhead for two identical <code>dgemm_{NN}</code> s.	24
2.2	Effects of data distribution in a NUMA machine on the performance of <code>dgemm_{NN}</code> .	32
2.3	Influence of caching on the execution time of <code>dgemv</code> .	33
3.1	Configuration parameters for the model generation and their studied values.	78
3.2	Model configuration parameters for minimum and maximum error and cost.	78
3.3	Model generator configurations remaining after pruning.	82
4.1	Sequence of calls, runtime estimates, and accumulated prediction for the inversion of a lower-triangular matrix with blocked algorithm 1.	87
4.2	Kernels in the Cholesky decomposition for different data-types.	99
4.3	Single-threaded runtime prediction ARE $t_{\text{ARE}}^{\text{med}}$ for blocked LAPACK algorithms averaged across problem sizes.	106
4.4	Multi-threaded runtime prediction ARE $t_{\text{ARE}}^{\text{med}}$ for blocked LAPACK algorithms averaged across problem sizes.	111
4.5	Average performance yields and improvement over LAPACK for <code>dsygst_{1L}</code> , <code>dgetrf</code> , and <code>dgeqrf</code> .	131
5.1	Estimation errors and improvements through cache-modeling for <code>dgeqrf</code> .	145

List of Tables

6.1	Free and contracted indices in BLAS kernels, examples of mapping them to $C_{abc} := A_{ai}B_{ibc}$, and resulting contraction algorithms. A and B refer to, respectively, the first and second kernel operand.	162
6.2	Operand sizes and access distances in <i>ca-dgemv</i> for $C_{abc} := A_{ai}B_{ibc}$	172

Bibliography

The bibliography is split into three parts: Papers of which I am a (co-)author are listed under [Publications](#) below, other scientific publications are collected in [References](#) on [Page 246](#), and websites and repositories are found under [Online Resources](#) on [Page 255](#).

Publications

- [1] Elmar Peise and Paolo Bientinesi. “Algorithm 979: Recursive Algorithms for Dense Linear Algebra—The ReLAPACK Collection”. In: *ACM Trans. Math. Softw.* 44.2 (Sept. 2017), 16:1–16:19. DOI: [10.1145/3061664](#).
- [2] Edoardo Di Napoli, Elmar Peise, Markus Hrywniak, and Paolo Bientinesi. “High-performance generation of the Hamiltonian and Overlap matrices in FLAPW methods”. In: *Computer Physics Communications* 211 (Feb. 2017). High Performance Computing for Advanced Modeling and Simulation of Materials, pages 61–72. DOI: [10.1016/j.cpc.2016.10.003](#).
- [3] Elmar Peise and Paolo Bientinesi. *The ELAPS Framework: Experimental Linear Algebra Performance Studies*. Technical report. Acceptor for publication in The International Journal of High Performance Computing Applications. AICES, RWTH Aachen University, Nov. 2016. arXiv: [1504.08035 \[cs.PF\]](#).
- [4] Rodrigo Canales, Elmar Peise, and Paolo Bientinesi. *Large Scale Parallel Computations in R through Elemental*. Technical report. Under review for the Journal of Statistical Software. AICES, RWTH Aachen University, Oct. 2016. arXiv: [1610.07310 \[stat.CO\]](#).
- [5] Elmar Peise and Paolo Bientinesi. “A Study on the Influence of Caching: Sequences of Dense Linear Algebra Kernels”. In: *High Performance Computing for Computational Science – VECPAR 2014: 11th International Conference*. Volume 8969. Lecture Notes in Computer Science. Springer International Publishing, Apr. 2015, pages 245–258. DOI: [10.1007/978-3-319-17353-5_21](#).

- [6] Elmar Peise, Diego Fabregat-Traver, and Paolo Bientinesi. “On the Performance Prediction of BLAS-based Tensor Contractions”. In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014*. Volume 8966. Lecture Notes in Computer Science. Springer International Publishing, Apr. 2015, pages 193–212. DOI: [10.1007/978-3-319-17248-4_10](https://doi.org/10.1007/978-3-319-17248-4_10).
- [7] Elmar Peise, Diego Fabregat-Traver, and Paolo Bientinesi. “High Performance Solutions for Big-data GWAS”. in: *Parallel Computing* 42.C (Feb. 2015). Parallelism in Bioinformatics, pages 75–87. DOI: [10.1016/j.parco.2014.09.005](https://doi.org/10.1016/j.parco.2014.09.005).
- [8] Elmar Peise and Paolo Bientinesi. *Cache-aware Performance Modeling and Prediction for Dense Linear Algebra*. Technical report. AICES, RWTH Aachen University, Nov. 2014. arXiv: [1409.8602](https://arxiv.org/abs/1409.8602) [cs.PF].
- [9] Elmar Peise, Diego Fabregat-Traver, Yurii S. Aulchenko, and Paolo Bientinesi. “Algorithms for Large-scale Whole Genome Association Analysis”. In: *Proceedings of the 20th European MPI Users’ Group Meeting*. EuroMPI ’13. ACM, 2013, pages 229–234. DOI: [10.1145/2488551.2488577](https://doi.org/10.1145/2488551.2488577).
- [10] Matthias Petschow, Elmar Peise, and Paolo Bientinesi. “High-Performance Solvers for Dense Hermitian Eigenproblems”. In: *SIAM Journal on Scientific Computing* 35.1 (Jan. 2013), pages C1–C22. DOI: [10.1137/110848803](https://doi.org/10.1137/110848803).
- [11] Elmar Peise and Paolo Bientinesi. “Performance Modeling for Dense Linear Algebra”. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. SCC ’12. IEEE Computer Society, Nov. 2012, pages 406–416. DOI: [10.1109/SC.Companion.2012.60](https://doi.org/10.1109/SC.Companion.2012.60).
- [12] Elmar Peise. “Hierarchical Performance Modeling for Ranking Dense Linear Algebra Algorithms”. Master’s thesis. Aachen Institute for Computational Engineering Science, RWTH Aachen, May 2012. arXiv: [1207.5217](https://arxiv.org/abs/1207.5217) [cs.PF].

References

- [13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”. In: (Mar. 2016). arXiv: [1603.04467](https://arxiv.org/abs/1603.04467) [cs.DC].

- [14] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. “Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects”. In: *Journal of Physics: Conference Series* 180.1 (2009), page 012037. DOI: [10.1088/1742-6596/180/1/012037](https://doi.org/10.1088/1742-6596/180/1/012037).
- [15] Pedro Alonso, Sandra Catalán, Francisco D. Igual, Rafael Mayo, Rafael Rodríguez-Sánchez, and Enrique S. Quintana-Ortí. “Time and energy modeling of high-performance Level-3 BLAS on x86 architectures”. In: *Simulation Modelling Practice and Theory* 55 (2015), pages 77–94. DOI: [10.1016/j.simpat.2015.04.003](https://doi.org/10.1016/j.simpat.2015.04.003).
- [16] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users’ Guide*. Third. Society for Industrial and Applied Mathematics, 1999. DOI: [10.1137/1.9780898719604](https://doi.org/10.1137/1.9780898719604).
- [17] Brett W. Bader and Tamara G. Kolda. “Algorithm 862: MATLAB Tensor Classes for Fast Algorithm Prototyping”. In: *ACM Trans. Math. Softw.* 32.4 (Dec. 2006), pages 635–653. DOI: [10.1145/1186785.1186794](https://doi.org/10.1145/1186785.1186794).
- [18] Rosa M. Badia, José R. Herrero, Jesús Labarta, Josep M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. “Parallelizing dense and banded linear algebra libraries using SMPs”. In: *Concurrency and Computation: Practice and Experience* 21.18 (July 2009), pages 2438–2456. DOI: [10.1002/cpe.1463](https://doi.org/10.1002/cpe.1463).
- [19] Grey Ballard, Dulceneia Becker, James Demmel, Jack Dongarra, Alex Drusinsky, Inon Peled, Oded Schwartz, Sivan Toledo, and Ichitaro Yamazaki. “Communication-Avoiding Symmetric-Indefinite Factorization”. In: *SIAM Journal on Matrix Analysis and Applications* 35.4 (Nov. 2014), pages 1364–1406. DOI: [10.1137/130929060](https://doi.org/10.1137/130929060).
- [20] R. H. Bartels and G. W. Stewart. “Solution of the Matrix Equation $AX + XB = C$ [F4]”. In: *Commun. ACM* 15.9 (Sept. 1972), pages 820–826. DOI: [10.1145/361573.361582](https://doi.org/10.1145/361573.361582).
- [21] Rodney J. Bartlett and Monika Musiał. “Coupled-cluster theory in quantum chemistry”. In: *Rev. Mod. Phys.* 79 (1 Feb. 2007), pages 291–352. DOI: [10.1103/RevModPhys.79.291](https://doi.org/10.1103/RevModPhys.79.291).
- [22] Haym Benaroya, Seon Mi Han, and Mark Nagurka. *Probability Models in Engineering and Science*. Taylor & Francis, 2005. ISBN: 9780824723156.
- [23] Peter Benner, Pablo Ezzatti, Enrique S. Quintana-Ortí, Alfredo Remón, and Juan P. Silva. “Tuning the Blocksize for Dense Linear Algebra Factorization Routines with the Roofline Model”. In: *Algorithms and Architectures for*

- Parallel Processing: ICA3PP 2016 Collocated Workshops: SCDT, TAPEMS, BigTrust, UCER, DLMCS*. Springer International Publishing, 2016, pages 18–29. DOI: [10.1007/978-3-319-49956-7_2](https://doi.org/10.1007/978-3-319-49956-7_2).
- [24] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. “The Science of Deriving Dense Linear Algebra Algorithms”. In: *ACM Trans. Math. Softw.* 31.1 (Mar. 2005), pages 1–26. DOI: [10.1145/1055531.1055532](https://doi.org/10.1145/1055531.1055532).
 - [25] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users’ Guide*. Society for Industrial and Applied Mathematics, 1997. DOI: [10.1137/1.9780898719642](https://doi.org/10.1137/1.9780898719642).
 - [26] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. “DAGuE: A generic distributed DAG engine for High Performance Computing”. In: *Parallel Computing* 38.1-2 (Jan. 2012). Extensions for Next-Generation Parallel Programming Models, pages 37–51. DOI: [10.1016/j.parco.2011.10.003](https://doi.org/10.1016/j.parco.2011.10.003).
 - [27] Gerth Stølting Brodal. “Cache-Oblivious Algorithms and Data Structures”. In: *Algorithm Theory – SWAT 2004: 9th Scandinavian Workshop on Algorithm Theory*. Springer Berlin Heidelberg, 2004, pages 3–13. DOI: [10.1007/978-3-540-27810-8_2](https://doi.org/10.1007/978-3-540-27810-8_2).
 - [28] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. “A Portable Programming Interface for Performance Evaluation on Modern Processors”. In: *The International Journal of High Performance Computing Applications* 14.3 (2000), pages 189–204. DOI: [10.1177/109434200001400303](https://doi.org/10.1177/109434200001400303).
 - [29] Richard L. Burden, Douglas J. Faires, and Annette M. Burden. *Numerical Analysis*. 10th edition. Cengage Learning, 2015. ISBN: 9781305465350.
 - [30] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoeffler, I. Karlin, M. Schulz, and F. Wolf. “Fast Multi-parameter Performance Modeling”. In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2016, pages 172–181. DOI: [10.1109/CLUSTER.2016.57](https://doi.org/10.1109/CLUSTER.2016.57).
 - [31] Alexandru Calotoiu, Torsten Hoeffler, Marius Poke, and Felix Wolf. “Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’13. ACM, 2013, 45:1–45:12. DOI: [10.1145/2503210.2503277](https://doi.org/10.1145/2503210.2503277).
 - [32] Sandra Catalán, Francisco D. Igual, Rafael Mayo, Rafael Rodríguez-Sánchez, and Enrique S. Quintana-Ortí. “Time and energy modeling of a high-performance multi-threaded Cholesky factorization”. In: *The Journal of Supercomputing* 73.1 (2017), pages 139–151. DOI: [10.1007/s11227-016-1654-6](https://doi.org/10.1007/s11227-016-1654-6).

- [33] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. “Supermatrix Out-of-order Scheduling of Matrix Operations for SMP and Multi-core Architectures”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’07. June 2007, pages 116–125. DOI: [10.1145/1248377.1248397](https://doi.org/10.1145/1248377.1248397).
- [34] Jiří Čížek. “On the Correlation Problem in Atomic and Molecular Systems. Calculation of Wavefunction Components in Ursell-Type Expansion Using Quantum-Field Theoretical Methods”. In: *The Journal of Chemical Physics* 45.11 (1966), pages 4256–4266. DOI: [10.1063/1.1727484](https://doi.org/10.1063/1.1727484).
- [35] Don Coppersmith and Shmuel Winograd. “Matrix multiplication via arithmetic progressions”. In: *Journal of Symbolic Computation* 9.3 (1990), pages 251–280. DOI: [10.1016/S0747-7171\(08\)80013-2](https://doi.org/10.1016/S0747-7171(08)80013-2).
- [36] Krister Dackland and Bo Kågström. “An hierarchical approach for performance analysis of ScaLAPACK-based routines using the distributed linear algebra machine”. In: *Applied Parallel Computing Industrial Computation and Optimization: Third International Workshop, PARA ’96*. Springer Berlin Heidelberg, 1996, pages 186–195. DOI: [10.1007/3-540-62095-8_20](https://doi.org/10.1007/3-540-62095-8_20).
- [37] Edoardo Di Napoli, Diego Fabregat-Traver, Gregorio Quintana-Ortí, and Paolo Bientinesi. “Towards an efficient use of the BLAS library for multilinear tensor contractions”. In: *Applied Mathematics and Computation* 235 (2014), pages 454–468. DOI: [10.1016/j.amc.2014.02.051](https://doi.org/10.1016/j.amc.2014.02.051).
- [38] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users’ Guide*. Society for Industrial and Applied Mathematics, 1979. DOI: [10.1137/1.9781611971811](https://doi.org/10.1137/1.9781611971811).
- [39] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. “A Set of Level 3 Basic Linear Algebra Subprograms”. In: *ACM Trans. Math. Softw.* 16.1 (Mar. 1990), pages 1–17. DOI: [10.1145/77626.79170](https://doi.org/10.1145/77626.79170).
- [40] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. “An Extended Set of FORTRAN Basic Linear Algebra Subprograms”. In: *ACM Trans. Math. Softw.* 14.1 (Mar. 1988), pages 1–17. DOI: [10.1145/42288.42291](https://doi.org/10.1145/42288.42291).
- [41] Jeremy J. Du Croz and Nicholas J. Higham. “Stability of Methods for Matrix Inversion”. In: *IMA Journal of Numerical Analysis* 12.1 (1992), pages 1–19. DOI: [10.1093/imanum/12.1.1](https://doi.org/10.1093/imanum/12.1.1).
- [42] E. Elmroth and F.G. Gustavson. “Applying recursion to serial and parallel QR factorization leads to better performance”. In: *IBM Journal of Research and Development* 44.4 (July 2000), pages 605–624. DOI: [10.1147/rd.444.0605](https://doi.org/10.1147/rd.444.0605).

- [43] Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyayev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I. Krylov. “New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations”. In: *Journal of Computational Chemistry* 34.26 (2013), pages 2293–2309. DOI: [10.1002/jcc.23377](https://doi.org/10.1002/jcc.23377).
- [44] Diego Fabregat-Traver and Paolo Bientinesi. “Automatic Generation of Loop-Invariants for Matrix Operations”. In: *2011 International Conference on Computational Science and Its Applications*. June 2011, pages 82–92. DOI: [10.1109/ICCSA.2011.41](https://doi.org/10.1109/ICCSA.2011.41).
- [45] Diego Fabregat-Traver and Paolo Bientinesi. “Knowledge-Based Automatic Generation of Partitioned Matrix Expressions”. In: *Computer Algebra in Scientific Computing: 13th International Workshop, CASC 2011*. Springer Berlin Heidelberg, 2011, pages 144–157. DOI: [10.1007/978-3-642-23568-9_12](https://doi.org/10.1007/978-3-642-23568-9_12).
- [46] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. “Cache-Oblivious Algorithms”. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. FOCS ’99. IEEE Computer Society, Oct. 1999, pages 285–297. DOI: [10.1109/SFFCS.1999.814600](https://doi.org/10.1109/SFFCS.1999.814600).
- [47] Burton S. Garbow. “EISPACK: A package of matrix eigensystem routines”. In: *Computer Physics Communications* 7.4 (1974), pages 179–184. DOI: [10.1016/0010-4655\(74\)90086-1](https://doi.org/10.1016/0010-4655(74)90086-1).
- [48] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. “The Scalasca performance toolset architecture”. In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pages 702–719. DOI: [10.1002/cpe.1556](https://doi.org/10.1002/cpe.1556).
- [49] K. Georgiev and J. Waśniewski. “Numerical Analysis and Its Applications: Second International Conference, NAA 2000”. In: Springer Berlin Heidelberg, 2001. Chapter Recursive Version of LU Decomposition, pages 325–332. DOI: [10.1007/3-540-45262-1_38](https://doi.org/10.1007/3-540-45262-1_38).
- [50] Kazushige Goto and Robert A. van de Geijn. “Anatomy of High-performance Matrix Multiplication”. In: *ACM Trans. Math. Softw.* 34.3 (May 2008), 12:1–12:25. DOI: [10.1145/1356052.1356053](https://doi.org/10.1145/1356052.1356053).
- [51] Kazushige Goto and Robert Van De Geijn. “High-performance Implementation of the Level-3 BLAS”. in: *ACM Trans. Math. Softw.* 35.1 (July 2008), 4:1–4:14. DOI: [10.1145/1377603.1377607](https://doi.org/10.1145/1377603.1377607).
- [52] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. “Gprof: A Call Graph Execution Profiler”. In: *SIGPLAN Not.* 17.6 (June 1982), pages 120–126. DOI: [10.1145/872726.806987](https://doi.org/10.1145/872726.806987).

- [53] F. G. Gustavson. “Recursion leads to automatic variable blocking for dense linear-algebra algorithms”. In: *IBM Journal of Research and Development* 41.6 (Nov. 1997), pages 737–755. DOI: [10.1147/rd.416.0737](https://doi.org/10.1147/rd.416.0737).
- [54] So Hirata. “Tensor Contraction Engine: Abstraction and Automated Parallel Implementation of Configuration-Interaction, Coupled-Cluster, and Many-Body Perturbation Theories”. In: *The Journal of Physical Chemistry A* 107.46 (2003), pages 9887–9897. DOI: [10.1021/jp034596z](https://doi.org/10.1021/jp034596z).
- [55] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. “Strassen’s Algorithm Reloaded”. In: IEEE Computer Society, 2016, pages 690–701. DOI: [10.1109/SC.2016.58](https://doi.org/10.1109/SC.2016.58).
- [56] Roman Iakymchuk. “Performance Modeling and Prediction for Linear Algebra Algorithms”. PhD thesis. AICES, RWTH Aachen University, Aug. 2012.
- [57] Roman Iakymchuk and Paolo Bientinesi. “Modeling Performance through Memory-Stalls”. In: *SIGMETRICS Perform. Eval. Rev.* 40.2 (Oct. 2012), pages 86–91. DOI: [10.1145/2381056.2381076](https://doi.org/10.1145/2381056.2381076).
- [58] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (Aug. 2008), pages 1–70. DOI: [10.1109/IEEESTD.2008.4610935](https://doi.org/10.1109/IEEESTD.2008.4610935).
- [59] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. “Cache-aware Roofline model: Upgrading the loft”. In: *IEEE Computer Architecture Letters* 13.1 (Jan. 2014), pages 21–24. DOI: [10.1109/L-CA.2013.6](https://doi.org/10.1109/L-CA.2013.6).
- [60] Isak Jonsson and Bo Kågström. “Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference Klagenfurt, Austria, August 26-29, 2003 Proceedings”. In: Springer Berlin Heidelberg, 2003. Chapter RECSY – A High Performance Library for Sylvester-Type Matrix Equations, pages 810–819. DOI: [10.1007/978-3-540-45209-6_111](https://doi.org/10.1007/978-3-540-45209-6_111).
- [61] Lars Karlsson. “Computing explicit matrix inverses by recursion”. Master’s thesis. Umea University, Department of Computing Science, Sweden, Feb. 2006.
- [62] Lawrence E. Kidder, Mark A. Scheel, and Saul A. Teukolsky. “Extending the lifetime of 3D black hole computations with a new hyperbolic system of evolution equations”. In: *Phys. Rev. D* 64 (Aug. 2001). DOI: [10.1103/PhysRevD.64.064017](https://doi.org/10.1103/PhysRevD.64.064017).
- [63] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. “Basic Linear Algebra Subprograms for FORTRAN Usage”. In: *ACM Trans. Math. Softw.* 5.3 (Sept. 1979), pages 308–323. DOI: [10.1145/355841.355847](https://doi.org/10.1145/355841.355847).
- [64] Luis Lehner. “Numerical relativity: a review”. In: *Classical and Quantum Gravity* 18.17 (2001), R25. DOI: [10.1088/0264-9381/18/17/202](https://doi.org/10.1088/0264-9381/18/17/202).

- [65] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. “Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis”. In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014*. Springer International Publishing, 2015, pages 129–148. DOI: [10.1007/978-3-319-17248-4_7](https://doi.org/10.1007/978-3-319-17248-4_7).
- [66] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. “Analytical Modeling Is Enough for High-Performance BLIS”. in: *ACM Trans. Math. Softw.* 43.2 (Aug. 2016), 12:1–12:18. DOI: [10.1145/2925987](https://doi.org/10.1145/2925987).
- [67] Piotr Luszczek and Jack Dongarra. “Reducing the Time to Tune Parallel Dense Linear Algebra Routines with Partial Execution and Performance Modeling”. In: *Parallel Processing and Applied Mathematics: 9th International Conference, PPAM 2011*. Springer Berlin Heidelberg, 2012, pages 730–739. DOI: [10.1007/978-3-642-31464-3_74](https://doi.org/10.1007/978-3-642-31464-3_74).
- [68] John D. McCalpin. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pages 19–25. URL: <http://www.cs.virginia.edu/~mccalpin/papers/balance/> (visited on May 1, 2017).
- [69] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. “Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System”. In: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Sept. 2009, pages 261–270. DOI: [10.1109/PACT.2009.22](https://doi.org/10.1109/PACT.2009.22).
- [70] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, and Makrus Püschel. “Applying the Roofline Model”. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2014, pages 76–85. DOI: [10.1109/ISPASS.2014.6844463](https://doi.org/10.1109/ISPASS.2014.6844463).
- [71] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. “Elemental: A New Framework for Distributed Memory Dense Matrix Computations”. In: *ACM Trans. Math. Softw.* 39.2 (Feb. 2013), 13:1–13:24. DOI: [10.1145/2427023.2427030](https://doi.org/10.1145/2427023.2427030).
- [72] Sameer S. Shende and Allen D. Malony. “The TAU Parallel Performance System”. In: *The International Journal of High Performance Computing Applications* 20.2 (May 2006), pages 287–311. DOI: [10.1177/1094342006064482](https://doi.org/10.1177/1094342006064482).
- [73] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. “Anatomy of High-Performance Many-Threaded Matrix

- Multiplication”. In: *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IPDPS '14. IEEE Computer Society, 2014, pages 1049–1059. DOI: [10.1109/IPDPS.2014.110](https://doi.org/10.1109/IPDPS.2014.110).
- [74] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. “Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions”. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IPDPS '13. IEEE Computer Society, May 2013, pages 813–824. DOI: [10.1109/IPDPS.2013.112](https://doi.org/10.1109/IPDPS.2013.112).
- [75] Paul Springer and Paolo Bientinesi. “Design of a high-performance GEMM-like Tensor-Tensor Multiplication”. In: (July 2016). arXiv: [1607.00145 \[cs.MS\]](https://arxiv.org/abs/1607.00145).
- [76] Volker Strassen. “Gaussian elimination is not optimal”. In: *Numerische Mathematik* 13.4 (1969), pages 354–356. DOI: [10.1007/BF02165411](https://doi.org/10.1007/BF02165411).
- [77] Sivan Toledo. “Locality of Reference in LU Decomposition with Partial Pivoting”. In: *SIAM J. Matrix Anal. Appl.* 18.4 (Oct. 1997), pages 1065–1081. DOI: [10.1137/S0895479896297744](https://doi.org/10.1137/S0895479896297744).
- [78] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. “Towards dense linear algebra for hybrid GPU accelerated manycore systems”. In: *Parallel Computing* 36.5-6 (June 2010), pages 232–240. DOI: [10.1016/j.parco.2009.12.005](https://doi.org/10.1016/j.parco.2009.12.005).
- [79] Jan Treibig, Georg Hager, and Gerhard Wellein. “LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments”. In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE Computer Society, Sept. 2010, pages 207–216. DOI: [10.1109/ICPPW.2010.38](https://doi.org/10.1109/ICPPW.2010.38).
- [80] Dan Tsafir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. “System Noise, OS Clock Ticks, and Fine-grained Parallel Applications”. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. ACM, 2005, pages 303–312. DOI: [10.1145/1088149.1088190](https://doi.org/10.1145/1088149.1088190).
- [81] Field G. Van Zee and Robert A. van de Geijn. “BLIS: A Framework for Rapidly Instantiating BLAS Functionality”. In: *ACM Trans. Math. Softw.* 41.3 (June 2015), 14:1–14:33. DOI: [10.1145/2764454](https://doi.org/10.1145/2764454).
- [82] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. van de Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. “The BLIS Framework: Experiments in Portability”. In: *ACM Trans. Math. Softw.* 42.2 (June 2016), 12:1–12:19. DOI: [10.1145/2755561](https://doi.org/10.1145/2755561).
- [83] Jerzy Waśniewski, Bjarne Stig Andersen, and Fred Gustavson. “Recursive Formulation of Cholesky Algorithm in Fortran 90”. In: *Applied Parallel*

- Computing Large Scale Scientific and Industrial Problems: 4th International Workshop*. PARA '98. Springer Berlin Heidelberg, 1998, pages 574–578. DOI: [10.1007/BFb0095384](https://doi.org/10.1007/BFb0095384).
- [84] R. C. Whaley. “Empirically tuning LAPACK’s blocking factor for increased performance”. In: *2008 International Multiconference on Computer Science and Information Technology*. Oct. 2008, pages 303–310. DOI: [10.1109/IMCSIT.2008.4747256](https://doi.org/10.1109/IMCSIT.2008.4747256).
 - [85] R. Clint Whaley and Jack J. Dongarra. “Automatically Tuned Linear Algebra Software”. In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. SC '98. IEEE Computer Society, 1998, pages 1–27. DOI: [10.1109/SC.1998.10004](https://doi.org/10.1109/SC.1998.10004).
 - [86] R. Clint Whaley and Antoine Petit. “Minimizing Development and Maintenance Costs in Supporting Persistently Optimized BLAS”. in: *Softw. Pract. Exper.* 35.2 (Feb. 2005), pages 101–121. DOI: [10.1002/spe.626](https://doi.org/10.1002/spe.626).
 - [87] R. Clint Whaley, Antoine Petit, and Jack J. Dongarra. “Automated empirical optimizations of software and the ATLAS project”. In: *Parallel Computing* 27.1-2 (2001). New Trends in High Performance Computing, pages 3–35. DOI: [10.1016/S0167-8191\(00\)00087-9](https://doi.org/10.1016/S0167-8191(00)00087-9).
 - [88] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Commun. ACM* 52.4 (Apr. 2009), pages 65–76. DOI: [10.1145/1498765.1498785](https://doi.org/10.1145/1498765.1498785).
 - [89] Yasaka Yamamoto. “Performance modeling and optimal block size selection for a BLAS-3 based tridiagonalization algorithm”. In: *Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCA-SIA'05)*. July 2005. DOI: [10.1109/HPCASIA.2005.76](https://doi.org/10.1109/HPCASIA.2005.76).
 - [90] Yusaku Yamamoto. “Performance Modeling and Optimal Block Size Selection for the Small-Bulge Multishift QR Algorithm”. In: *Parallel and Distributed Processing and Applications: 4th International Symposium, ISPA 2006*. Springer Berlin Heidelberg, 2006, pages 451–463. DOI: [10.1007/11946441_44](https://doi.org/10.1007/11946441_44).
 - [91] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. “QUARK Users’ Guide: Queueing and runtime for kernels”. In: *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02* (Apr. 2011). URL: http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf (visited on May 1, 2017).
 - [92] Kamen Yotov, Xiaoming Li, Gang Ren, María Jesús Garzarán, David Padua, Keshav Pingali, and Paul Stodghill. “Is Search Really Necessary to Generate High-Performance BLAS?”. In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pages 358–386. DOI: [10.1109/JPROC.2004.840444](https://doi.org/10.1109/JPROC.2004.840444).

- [93] Field Van Zee. *libflame: The Complete Reference*. Oct. 2011. URL: <http://www.cs.utexas.edu/~flame/web/libflame.pdf> (visited on May 1, 2017).
- [94] Field Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. “Introducing: The libflame Library for Dense Matrix Computations”. In: *Computing in Science Engineering* PP.99 (Sept. 2016). DOI: [10.1109/MCSE.2009.154](https://doi.org/10.1109/MCSE.2009.154).

Online Resources

All websites and repositories were accessible as of March 1, 2018.

- [95] *Automatically Tuned Linear Algebra Software (ATLAS)*. URL: <http://math-atlas.sourceforge.net/>.
- [96] *BLAS (Basic Linear Algebra Subprograms)*. URL: <http://www.netlib.org/blas/>.
- [97] *BLIS: BLAS-like Library Instantiation Software Framework*. GitHub: [flame/blis](https://github.com/flame/blis).
- [98] *clBLAS*. GitHub: [iclMathLibraries/clBLAS](https://github.com/iclMathLibraries/clBLAS).
- [99] *cuBLAS*. NVIDIA Corporation. URL: <https://developer.nvidia.com/cublas>.
- [100] *Cyclops Tensor Framework*. GitHub: [solomonik/ctf](https://github.com/solomonik/ctf). URL: <http://solomon2.web.engr.illinois.edu/ctf/>.
- [101] *EISPACK*. URL: <http://www.netlib.org/eispack/>.
- [102] *ELAPS: Experimental Linear Algebra Performance Studies*. GitHub: [elmarpeise/ELAPS](https://github.com/elmarpeise/ELAPS).
- [103] *Elemental*. GitHub: [elemental/Elemental](https://github.com/elemental/Elemental). URL: <http://libelemental.org/>.
- [104] *Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL*. IBM Corporation. URL: <http://www.ibm.com/systems/power/software/essl/>.
- [105] *Framework Accelerate*. Apple Inc. URL: <https://developer.apple.com/reference/accelerate>.
- [106] *GNU gprof*. URL: <https://sourceware.org/binutils/docs/gprof/>.
- [107] *GotoBLAS2*. URL: <https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2>.
- [108] *Green500*. URL: <https://www.top500.org/green500/>.
- [109] *Intel® Math Kernel Library (Intel® MKL)*. Intel Corporation. URL: <https://software.intel.com/en-us/intel-mkl>.

Bibliography

- [110] *Intel® VTune™ Amplifier 2017*. Intel Corporation. URL: <https://software.intel.com/en-us/intel-vtune-amplifier-xe>.
- [111] *LAPACK: Linear Algebra PACKage*. GitHub: [Reference-LAPACK/lapack](https://github.com/Reference-LAPACK/lapack). URL: <http://www.netlib.org/lapack/>.
- [112] *libFLAME: High-performance object-based library for DLA computations*. GitHub: [flame/libflame](https://github.com/flame/libflame). URL: <https://www.cs.utexas.edu/~flame/web/libFLAME.html>.
- [113] *libtensor: Tensor algebra library for computational chemistry*. GitHub: [epifanovsky/libtensor](https://github.com/epifanovsky/libtensor). URL: <http://iopshell.usc.edu/downloads/tensor/>.
- [114] *LIKWID: Performance monitoring and benchmarking suite*. GitHub: [RRZE-HPC/likwid](https://github.com/RRZE-HPC/likwid).
- [115] *LINPACK*. URL: <http://www.netlib.org/linpack/>.
- [116] *MAGMA: Matrix Algebra on GPU and Multicore Architectures*. URL: <http://icl.cs.utk.edu/magma/>.
- [117] *MATLAB Tensor Toolbox Version 2.6*. URL: <http://www.sandia.gov/~tgkolda/TensorToolbox/>.
- [118] *OpenBLAS: An optimized BLAS library*. GitHub: [xianyi/OpenBLAS](https://github.com/xianyi/OpenBLAS). URL: <http://www.openblas.net/>.
- [119] *PAPI*. URL: <http://icl.cs.utk.edu/papi/software/>.
- [120] *RECSY: High Performance Library for Sylvester-Type Matrix Equations*. URL: <http://www8.cs.umu.se/~isak/recsy/>.
- [121] *ReLAPACK: Recursive LAPACK Collection*. GitHub: [elmar-peise/ReLAPACK](https://github.com/elmar-peise/ReLAPACK).
- [122] *ScaLAPACK: Scalable Linear Algebra PACKage*. URL: <http://www.netlib.org/scalapack/>.
- [123] *Scalasca*. URL: <http://www.scalasca.org/>.
- [124] *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. URL: <http://www.cs.virginia.edu/stream/>.
- [125] *TAU Performance System®*. ParaTools, Inc. URL: <http://www.paratools.com/TAU>.
- [126] *TCE: The Tensor Contraction Engine*. URL: <http://www.csc.lsu.edu/~gb/TCE/>.
- [127] *TOP500*. URL: <https://www.top500.org/>.
- [128] *VAMPIR*. GWT-TUD GmbH. URL: <https://www.vampir.eu/>.

About This Document

This document was written in $\text{\LaTeX 2}_{\epsilon}$ and typeset with pdfTeX Version 3.14159265-2.6-1.40.18 on March 20, 2018.

It relies on the following packages: `microtype` for micro-typography; `listings` and `tcolorbox` for algorithms, listings, and examples; `tikz` and `pgfplots` for graphics and plots; `drawmatrix` for matrix visualizations; `cleveref` and `hyperref` for references and hyperlinks; and `biblatex` for the bibliography.