2017

Maintaining High Performance Across All Problem Sizes and Parallel Scales Using Microkernel-based Linear Algebra

Md Rakib Hasan
Louisiana State University and Agricultural and Mechanical College, mailtorakib@gmail.com

MAINTAINING HIGH PERFORMANCE ACROSS ALL PROBLEM SIZES AND PARALLEL SCALES USING MICROKERNEL-BASED LINEAR ALGEBRA

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The School of Electrical Engineering and Computer Science
The Division of Computer Science and Engineering

by
Md Rakib Hasan
B.S., Bangladesh University of Engineering and Technology, 2009
August 2017

ACKNOWLEDGMENTS

I want to express deep gratitude to my advisor, Dr. R. Clint Whaley, for his excellent guidance and tremendous effort that enabled my research. I also want to thank Dr. Sukhamay Kundu, Dr. Ramachandran Vaidyanathan, Dr. Qingyang Wang and Dr. Hongchao Zhang for serving on my Ph.D. committee.

I am thankful to Eric Van Hensbergen and Wade Walker for giving me the opportunity to work on ARM architecture and guiding me at every step. I also want to thank Wooseok, Dan, Essan for their help for all the research work on ARM architecture.

This research was supported in part by the National Science Foundation (NSF) grant No. OCI-1149303, the NSF EPSCoR Cooperative Agreement No. EPS-1003897, with additional support from the Louisiana Board of Regents and the Department of Energy and Lawrence Livermore National Security, LLC ("LLNS") under contract number DE-AC52-07NA27344 as part of the Fast Forward 2 ("FF2") program.

Lastly, I am deeply grateful to my parents and my wife for their continuous support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

NOMENCLATURE

Informal descriptions to aid readers outside the area in understanding this dissertation. For topics of greater importance to this work, we provide the page number for more substantive coverage.

Abbreviations

gemmμ	Microkernel for GEMM. See page 5.
trmmμ	Microkernel for handling diagonal blocks of the triangular matrix in TRMM. See page 35.
trsmμ	Microkernel for handling the diagonal blocks of the triangular coefficient matrix in TRSM. See page 52.
ATLAS	Automatically Tuned Linear Algebra Software. Empirical tuning framework used in this research. See page 1.
BLAS	Basic Linear Algebra Subprograms. API and kernel library hiding most architecture-specific optimizations so that higher-level libraries may be written largely independent of the computing platform. See page 1.
DVFS	Dynamic Voltage and Frequency Scaling. See page 135.
FLOP	FLoating point OPeration.
FLOPs	Plural of FLOP.
GEMM	GEneral Matrix-matrix Multiply, a L3BLAS routine. See page 5.
GEMV	GEneral Matrix-Vector multiply, a L2BLAS routine.
GER	GEneral Rank-1 update (update a matrix with the outer product of two vectors), a L2BLAS routine.
GETRF	GEneral TRiangular Factorization, a LAPACK routine for performing the LU factorization. Factorizes rectangular matrix into PLU, where P performs pivoting, and L and U are Upper and Lower triangular matrices, respectively. See page 79.
GFLOPS	Billions of floating point operations per second, a measure of speed.
HEMM	HErmitian Matrix-matrix Multiply, a L3BLAS routine.
HER2K	HErmitian Rank-2K update, a L3BLAS routine.

HERK	HErmitian Rank-K update version of matrix-matrix multiply, a L3BLAS routine.
IAMAX	Index of Absolute-value MAXimum within a vector, a L1BLAS routine.
L1BLAS	Level 1 BLAS, comprised mainly of routines for performing vector-vector computations. Includes many routines, but only IAMAX, SCAL, and SWAP mentioned explicitly in this paper. See page 1.
L2BLAS	Level 2 BLAS routines, comprised of routines for performing matrix-vector computations. Includes many routines, but only GEMV, GER, TRSV mentioned explicitly in this paper. See page 1.
L3BLAS	Level 3 BLAS, comprised of routines for performing matrix-matrix computations. Includes GEMM, HEMM, SYMM, HERK, SYRK, HER2K, SYR2K, TRMM and TRSM. See page 2.
LAPACK	Linear Algebra PACKage. See page 2.
LASWP	An auxiliary routine in LAPACK to apply a series of row-swaps.
POTRF	POsitive definite TRiangular Factorization, a LAPACK routine for performing the Cholesky factorization. Cholesky is the LU factorization specialized for symmetric positive definite matrices; when this is known we can exploit the facts $U = L^T$ and that no pivoting is required to perform LU in roughly half the number of computations. See page 88.
SCAL	SCALe a vector by a scalar, a L1BLAS routine.
SIMD	Single Instruction, Multiple Data.
SWAP	SWAP values held in two vectors, a L1BLAS routine.
SYMM	SYmmetric Matrix-matrix Multiply, a L3BLAS routine.
SYR2K	SYmmetric Rank-2K update, a L3BLAS routine.
SYRK	SYmmetric Rank-K update version of matrix-matrix multiply, a L3BLAS routine.
TRMM	TRiangular Matrix-matrix Multiply, a L3BLAS routine. See page 19 and 25.
TRMV	TRiangular Matrix-Vector multiply, a L2BLAS routine
TRSM	TRiangular Solve of a Matrix of right-hand sides, a L3BLAS routine. See page 49.

Definitions

Back solve	Triangular solve (TRSM) when coefficient matrix is LOWER triangular.
Column-panel	A group of columns in a matrix. The number of columns typically depend on the block factor along the N-dimension.
Core	Independent CPU (ALUs+FPUs+vector units, etc) sharing a chip/die with other cores.
Forward solve	Triangular solve (TRSM) when coefficient matrix is UPPER triangular.
Kernel	Performance-centric code section operating on arbitrarily-sized data that may perform high-overhead optimizations like operand copy. Mostly refers to BLAS routines in this paper. See page 1.
Loop Unrolling	Replicating the body of a loop some number of times (unroll factor). An optimization technique to reduce loop overhead, increase instruction-level parallelism, etc.
Microkernel	Performance-centric code section operating on cache-blocked data that does only part of a kernel computation, and applies only low-overhead optimizations. See page 2.
Nanokernel	Performance-centric code section doing $O(1)$ computations used in the building of a microkernel. Usually a `cpp` macro or `inline` function to avoid function call overhead. See page 55.
Parallel scale	Measure of explicit parallelism. Usually either core count, or core count $\times$ SIMD vector length.
Row-panel	A group of rows in a matrix. The number of rows typically depend on the block factor along the M-dimension.
SIMD vectorization	Using instructions that operate on a vector of values, rather than only a scalar. At the hardware level, relatively easy to increase parallelism by duplicating functional units. Typical vector lengths vary between 2-32 depending on machine and scalar data size.
Unroll & Jam	Unrolling of an outer-loop and fusing the copies of the inner-loop back together.

ABSTRACT

Linear algebra underlies a large proportion of computational problems. With the continuous increase of scale on modern hardware, performance of small sized linear algebra has become increasingly important. To overcome the shortcomings of conventional approaches, we employ a new approach using a microkernel framework provided by ATLAS to improve the performance of a few linear algebra routines for all problem sizes. Our initial research consists of improving the performance of parallel LU factorization in ATLAS for which we were able to achieve up to 2.07x and 2.66x speedup for small problems, up to 91% and 87% of theoretical peak performance for asymptotic problems on a 12-core Intel Xeon and a 32-core AMD Opteron machine, respectively, outperforming all the state-of-the-art libraries at the time. Such performance was achieved via an exhaustive search of all the tuning parameters, which could take days. This motivated us to try to develop a computational model for our LU factorization that could predict those parameters by combining some basic empirical timings and a theoretical model based on the amount of required computations. While our model provided good prediction for mid-to-asymptotic sized problems, there were some unknown factors for small problems that could possibly be answered by extending the ATLAS tuning framework. While this extension is underway, we decided to pursue the model research using simpler serial BLAS-based approach. We investigated and implemented two Level-3 BLAS routines: TRSM and TRMM that are widely used primarily by LAPACK operations like the aforementioned LU factorization. With the microkernel-based approach, we were able to improve the performance of both routines by up to 15% and 73% for square and fat problems, respectively, over prior ATLAS implementations on modern hardware. Finally, with a collaborative research with ARM Inc., we improved the performance of the most important Level-3 BLAS operation GEMM in ATLAS by up to 53% via implementing microkernels

for two 64-bit ARM architectures. This automatically improves other BLAS and LAPACK routines that rely on GEMM for high performance.

CHAPTER 1
INTRODUCTION

1.1 Basic Introduction to the Libraries/APIs Optimized by this Research

The goal of this research is to provide linear algebra computational kernels that maintain high efficiency for a wide range of problem sizes and parallel scales. We performed this work using the ATLAS [71, 72, 73, 76] (Automatically Tuned Linear Algebra Software) empirical tuning research framework.

The ATLAS project uses automated timings and code transformations to produce portably and persistently optimal linear algebra libraries for scientists and engineers worldwide [76]. It provides support for two widely used HPC (High Performance Computing) APIs (Application Programming Interfaces), the BLAS [35, 51, 23, 24, 22] (Basic Linear Algebra Subprograms), and LAPACK [3] (Linear Algebra PACKage).

The BLAS is a basic linear algebra *kernel* library. A *kernel* is a mathematically simple routine whose main purpose is to provide a high performance building block operation for higher level algorithms (e.g. computing the Eigenvalues). The BLAS are split into three levels, indicating the type of operation they do. The Level 1 BLAS [35, 51] (L1BLAS) perform vector-vector operations (e.g. dot product), and thus they perform $O(N)$ computations on $O(N)$ data. The L2BLAS [23, 24] handle matrix-vector computations, and thus perform $O(N^2)$ operations on $O(N^2)$ data. The L2BLAS include things like matrix-vector multiplication (GEMV), rank-1 update of a matrix (GER), forward- and back-solve on a vector (TRMV), etc.

Because the data and computation is of the same order for the L1 and L2BLAS, their performance is overwhelmingly limited by the speed of memory, which is multiple orders of magnitude slower than compute speed on almost all modern computing systems. Due to their inherently low performance, these kernels are only used when there are no other ways of doing the operation, and so they are not the focus of our research here.

The Level 3 BLAS (L3BLAS) perform matrix-matrix operations [22], which can be categorized as doing $O(N^3)$ operations on $O(N^2)$ data. This allows them to be cache blocked, and when well-tuned they can get very close to the theoretical peak of the hardware on many machines. The most important L3BLAS operation is GEMM (GEneral Matrix-matrix Multiply), which updates an optionally scaled output matrix with the optionally scaled product of two matrices. Due to its lack of strong dependencies and high data reuse potential, a tuned matrix multiply is one of the most efficient operations that the architecture can perform. For this reason, its performance underlies the majority of high performance linear algebra.

Probably the next most important L3BLAS routine is TRSM (TRiangular Solve to a Matrix of right-hand sides), which performs a highly stable and efficient triangular solve (e.g. forward- and back-solve using an upper or lower triangular matrix). Section 2.3 will discuss TRSM in detail. TRMM (TRiangular Matrix-matrix Multiply) is also an important L3BLAS kernel, and is particularly important for the QR (Householder) factorization [8, 9, 28]. Section 2.2 will discuss TRMM in detail.

The average computational scientist will usually be calling the BLAS directly only when doing fairly low level development. Most will instead call LAPACK [3], which provides higher level operations such as matrix factorization, inversion, and a host of routines related to Eigenvalues. The performance of LAPACK is largely determined by the BLAS performance on the target machine. In this research, we mainly concentrated on the LAPACK LU & Cholesky factorizations (LAPACK API names GETRF and POTRF, respectively). GETRF is essentially Gaussian factorization for general rectangular matrices, while POTRF is the same but optimized for symmetric positive definite matrices. These factorizations underlie a host of linear algebra operations.

1.2 Introducing the Microkernel Concept

One major way that ATLAS was different from its empirical tuning predecessor PHiPAC [7] was that it used a *microkernel* strategy. The idea is that even kernels that are *mathematically*

simple like GEMM are *computationally* quite complex due to the number of modern architectural features that must be addressed to achieve high performance. ATLAS's approach was to hide the vast majority of code transformations inside a much simpler routine that could be tuned in isolation under known conditions. This simple *microkernel* could hide all optimizations that require actually rewriting the code, so that only a small section of code need be heavily adapted to the target architecture. Since the microkernel is much simpler, it is easier to generate, time, and hand-tune than a more general kernel like GEMM.

Optimizations that require only changing run- or compile-time constants, such as blocking, can be handled with fixed code implementing the higher-level kernel, while the microkernel is generated and handles transformations like loop unrolling, register exploitation, scheduling, efficient looping and indexing that require changes to the compiled code.

A modern GEMM performs a host of optimizations to achieve high performance. Cache blocking is the most important, and on many machines cache blocking actually requires copying to more cache-friendly formats to consistently achieve maximal performance. As long as the problem size is large, this $O(N^2)$ data copy overhead more than pays for itself during the course of the $O(N^3)$ computation, but if the kernel is instead repeatedly called with small problems, this low-order overhead can become vitally important.

Due to architectural trends, ATLAS has become even more heavily oriented towards microkernels. Whereas the original ATLAS had one primary microkernel, the current releases have hundreds of them. One of the primary drivers of this is increasing parallel scale, which has repercussions that have greatly increased the importance of microkernels.

As parallel scale increases, problems that were considered large are distributed over enough cores that the individual problems all become small. This means that problems must be scaled by the increased core count just to achieve the same efficiency as they did on prior machines. Since clock rates have stagnated, users with fixed-size problems may actually run *slower* than they did on older machines!

The second problem with increasing parallel scale is that it becomes much harder to keep cores in sync: it takes longer for all cores to become active, longer for them to report finishing, and the OS often fails to manage them efficiently. This leads to the need to dynamically schedule all parallel operations, but as you increase the job size to ensure that low-order costs like data copy are overcome, the job granularity becomes too great for effective dynamic scheduling.

Therefore, for extreme-scale computing, we need a way to directly control overheads, so that we incur them the minimum number of times, rather than repeatedly across many calls. The ATLAS microkernels have been designed to facilitate this control. In particular, all of the data copying, which can cause implicit performance effects (e.g. cache flushing) in addition to the explicit cost of the copy computation, are done above the microkernel layer. This allows the knowledgeable user to manage this high-overhead optimization manually, and thus potentially reduce it to the theoretical minimum. Reducing this overhead, and carefully managing the cache for maximum parallel benefit was a major feature of this research, and is discussed in detail in Chapter 4.

1.2.1 General Features of Microkernels

Kernel libraries like the BLAS try to provide high performance for a broad class of use-cases and user experience. They feature easy-to-understand APIs, error checking for safety, and general usefulness.

A microkernel on the other hand can afford almost no overhead, and so its APIs are dictated by performance concerns, its usefulness is very narrow, and its successful use requires a sophisticated understanding of both the particular microkernel and some of the features typically hidden within a kernel library like the BLAS (e.g. blocking and its related data copying). So choosing a microkernel over kernel usage is usually trading ease-of-use and error checking for increased expert control. Because modern parallel scale has made *many* low-

overhead costs important, this has resulted in a hugely expanded need to control all details of jobs, which has produced our greatly expanded reliance on microkernels within ATLAS.

1.2.2 ATLAS Main Microkernel Family: the GEMM Microkernel, *gemm*μ

The BLAS routine GEMM performs the operation $C \leftarrow \alpha AB + \beta C$, where A is an $M \times K$ matrix, B is a $K \times N$ matrix, and C is an $M \times N$ matrix. This BLAS kernel is itself built out of a much simpler *microkernel* whose efficiency dictates the performance of the entire L3BLAS and most of LAPACK. We call this simplified microkernel *gemm*μ, and it assumes that the matrix operands fit in some level of cache, and are stored in cache-friendly access-major storage. A *gemm*μ implementation will be compiled three times in order to support differing β cases:

$$\beta = 0\text{:} \quad C \leftarrow AB$$

$$\beta = 1\text{:} \quad C \leftarrow AB + C$$

$$\beta = -1\text{:} \quad C \leftarrow AB - C$$

In this discussion, it will be necessary to differentiate the *matrix* from the *storage array* (or more simply, the *array*). The matrix is a mathematical entity, and is used in the operation definitions above. However, the *array* is how we store the numbers from the matrix in the memory of the computer, and as we will see this is not simple for *gemm*μ. Most Fortran programmers are familiar with column-major arrays, which store columns of the matrix contiguously in memory, while matrix rows are strided, while C-family programmers may be more used to row-major arrays. The arrays used by *gemm*μ have a more complex storage pattern, where the matrix has been permuted so that all arrays are naturally accessed in a purely sequential fashion when the computation is being performed. Completely sequential access allows us to minimize cache line conflicts, maximize cache line packing & hardware prefetch accuracy, and ensures that our bus access is as "smooth" as possible (i.e. it minimizes the number of cache misses that happen at any one time). The earliest discussion of this rough idea is probably [38]. To facilitate such sequential access, each *gemm*μ is associated with copy microkernels for all three A, B, and C matrices. These copy microkernels can

optionally scale the matrices with α and/or β, and for A or B, it can also optionally transpose (and for complex types, conjugate or conjugate-transpose) the data during the copy process. ATLAS provides copy microkernels for both directions: standard to access-major format and access-major to standard format for copying-in the input and copying-back the result, respectively.

More specifically, to understand what the *gemmμ* copy microkernels can do in detail, assume A_c refers to any general column-major array, while W_μ refers to any optimized microkernel workspace. Assume α and β are scalars, and the $op(X)$ produces X, or X^T. For complex numbers, $op(X)$ can also produce $\overline{X}$ (conjugate of X, or X with negated imaginary components) or $\overline{X^T} = X^H$ (Hermitian transpose of X). With this notation all A_c arrays have the same data structure (column-major array), while the structure of any two W_μ's may be unrelated. We can therefore say that all three operands to GEMM are in A_c format, while all three operands to *gemmμ* are in (possibly differing) W_μ formats. Note also that A_c^T is essentially a row-major array. Given this notation, Table 1.1a shows the supported copy operations for *gemmμ*'s C (output) array, while Table 1.1b shows the supported copy operations for the input (A or B) arrays.

1.2.3 Implementation Overview and Loop Ordering for *gemmμ*

Figure 1.1a shows pseudocode for a simple GEMM implementation (there are many ways to write GEMM, this is MNK loop order). The compute cost of GEMM is $2 \times M \times N \times K$ floating point operations, which can legally be computed in any order. We often simplify this analysis by considering the "square" case, where $M = N = K$, which allows us to say it performs $2N^3$ FLOPs, and is therefore an $O(N^3)$ algorithm. To understand this FLOP count notice that if $M = N = K$, then the body of the innermost loop is executed N^3 times, and in each iteration it does 2 floating point operations (1 add and 1 multiply). One of the interesting features of GEMM is that its correctness does not depend on the order of the

Table 1.1: Actual operation performed α, β, *op*(), and direction settings for *gemm*μ's copy routines for the (a) output array C (left) and (b) input arrays A or B (right). 'X' means set to any value not indicated in prior element of this column. Note that $\alpha = 0$ is not supported.

β	α	$W_\mu \leftarrow A_c$ Operation	$A_c \leftarrow W_\mu$ Operation
0	1	$W_\mu \leftarrow A_c$	$A_c \leftarrow W_\mu$
1	1	$W_\mu \leftarrow W_\mu + A_c$	$A_c \leftarrow A_c + W_\mu$
-1	1	$W_\mu \leftarrow A_c - W_\mu$	$A_c \leftarrow W_\mu - A_c$
X	1	$W_\mu \leftarrow \beta W_\mu + A_c$	$A_c \leftarrow \beta A_c + W_\mu$
0	-1	$W_\mu \leftarrow -A_c$	$A_c \leftarrow -W_\mu$
1	-1	$W_\mu \leftarrow W_\mu - A_c$	$A_c \leftarrow A_c - W_\mu$
-1	-1	$W_\mu \leftarrow -A_c - W_\mu$	$A_c \leftarrow -W_\mu - A_c$
X	-1	$W_\mu \leftarrow \beta W_\mu - A_c$	$A_c \leftarrow \beta A_c - W_\mu$
0	X	$W_\mu \leftarrow \alpha A_c$	$A_c \leftarrow \alpha W_\mu$
1	X	$W_\mu \leftarrow W_\mu + \alpha A_c$	$A_c \leftarrow A_c + \alpha W_\mu$
-1	X	$W_\mu \leftarrow \alpha A_c - W_\mu$	$A_c \leftarrow \alpha W_\mu - A_c$
X	X	$W_\mu \leftarrow \beta W_\mu + \alpha A_c$	$A_c \leftarrow \beta A_c + \alpha W_\mu$

(a)

op()	α	$W_\mu \leftarrow A_c$ Operation	$A_c \leftarrow W_\mu$ Operation
N	1	$W_\mu \leftarrow A_c$	$A_c \leftarrow W_\mu$
N	-1	$W_\mu \leftarrow -A_c$	$A_c \leftarrow -W_\mu$
N	X	$W_\mu \leftarrow \alpha A_c$	$A_c \leftarrow \alpha W_\mu$
T	1	$W_\mu \leftarrow A_c^T$	$A_c \leftarrow W_\mu^T$
T	-1	$W_\mu \leftarrow -A_c^T$	$A_c \leftarrow -W_\mu^T$
T	X	$W_\mu \leftarrow \alpha A_c^T$	$A_c \leftarrow \alpha W_\mu^T$
C	1	$W_\mu \leftarrow \overline{A_c}$	$A_c \leftarrow \overline{W_\mu}$
C	-1	$W_\mu \leftarrow \overline{-A_c}$	$A_c \leftarrow \overline{-W_\mu}$
C	X	$W_\mu \leftarrow \alpha\overline{A_c}$	$A_c \leftarrow \alpha\overline{W_\mu}$
H	1	$W_\mu \leftarrow \overline{A_c^T}$	$A_c \leftarrow \overline{W_\mu^T}$
H	-1	$W_\mu \leftarrow \overline{-A_c^T}$	$A_c \leftarrow \overline{-W_\mu^T}$
H	X	$W_\mu \leftarrow \alpha\overline{A_c^T}$	$A_c \leftarrow \alpha\overline{W_\mu^T}$

(b)

operations, so it is perfectly legal to reorder the loop nesting. Therefore, Figure 1.1b shows a mathematically equivalent implementation using the KNM loop ordering.

In addition to all the combinatoric loop orders, you can intermix them, by for instance, doing multiple iterations at once (this corresponds to loop unrolling), or only going part way through K (or any other dimension) in one set of loops, and then doing another set of loops later to finish the computation off (this is the basis of *blocking*).

So, we see we have enormous degrees of freedom for this simple operation; all of these varying implementations are equivalent mathematically, but they can have very different performance aspects due mainly to their different memory access profiles. To reduce memory

```
for (i=0; i < M; i++)
   for (j=0; j < N; j++)
       for (k=0; k < K; k++)
          C(i,j) = C(i,j) + A(i,k) * B(k,j);
```

(a)

```
for (k=0; k < K; k++)
   for (j=0; j < N; j++)
      for (i=0; i < M; i++)
          C(i,j) = C(i,j) + A(i,k) * B(k,j);
```

(b)

Figure 1.1: Mathematically equivalent pseudocodes for GEMM $\alpha = \beta = 1$ case: (a) MNK loop order (b) KNM loop order

costs, the innermost loop will always be over the K dimension. To understand why, we have to look at the access of each of the operands as dictated by the above loops.

Examining this code, we can see that, at least on a machine without registers, all three operands (A, B, and C) are all also accessed $O(N^3)$ times. There are N^3 reads of A, B and C, but C additionally experiences (N^3) writes. When registers are considered, the loop order has a strong effect on the number of memory accesses required by the algorithm. In particular, registers can be used to reduce one (and only one) of the array access from $O(N^3)$ to $O(N^2)$. Which array is reduced to $O(N^2)$ memory accesses is determined by which loop is placed innermost. Notice that the input matrices are only read, while C is both read *and* written (due to cache effects, C will likely be read even in the $C \leftarrow AB$ formulation). If we make the naive assumption that all accesses count the same (not true; due to architectural issues, some writes are cheaper than reads, and some are much more expensive), this gives us the estimate that C has twice the access cost as either A or B, and so we should choose the innermost loop that reduces C's access to $O(N^2)$, and leave the less expensive input arrays at the original $O(N^3)$ access cost. Iterating over the K dimension in the innermost loop accomplishes this.

To see how, assume that any scalar variable gets assigned to a register (a register is a special storage area on a computer that has room to hold only a single value, and is the fastest way to store and access variables; you can think of it as essentially free when compared to memory or cache access), while any matrix access in the pseudo-code will be understood to use memory reads and writes. Figure 1.2a shows how the memory accesses of C can be reduced to only $2MN$ references (the theoretical minimum of 1 read & 1 write per element). The total number of C references is unchanged, but now innermost-loop accesses to the memory locations of C have been transformed into register accesses, so we have $2MNK$ accesses of the register `c00` (ignoring the zeroing of `c00`), but only $2MN$ accesses of the elements of C.

```
1       for (i=0; i < M; i++)
2           for (j=0; j < N; j++)
3           {
4               register c00 = 0.0;
5               for (k=0; k < K; k++)
6                   c00 = c00 + A(i,k) * B(k,j);
7               C(i,j) = C(i,j) + c00;
8           }
```

(a)

```
1       for (i=0; i < M; i++)
2           for (j=0; j < N; j++)
3           {
4               register c00 = 0.0;
5               for (k=0; k < K; k += 4)
6               {
7                   c00 += A(i,k) * B(k,j);
8                   c00 += A(i,k+1) * B(k+1,j);
9                   c00 += A(i,k+2) * B(k+2,j);
10                  c00 += A(i,k+3) * B(k+3,j);
11              }
12              C(i,j) += c00;
13          }
```

(b)

Figure 1.2: GEMM with register blocking for C: (a) K-loop rolled and (b) K-loop unrolled to 4 ($u_k = 4$)

This simple analysis is enough to understand why K is the innermost loop. With this fixed, we could choose for the microkernel to use NMK or MNK loop order. ATLAS's *gemm*μ mostly use the MNK order for slightly improved cache effects in some particularly important *gemm*μ use cases.

1.2.4 Reducing Loop Overhead Through Loop Unrolling

We can reduce the overhead of looping via loop unrolling (loop overhead includes the cost of updating the loop variable, doing the loop-exit comparison, and executing the branch). Figure 1.2b shows our C register-blocked GEMM where the loop over the K dimension has been unrolled four times. For any dimension D, we will use u_d to indicate the unroll factor, so this listing shows $u_k = 4$ (for simplicity, we assume K is a multiple of 4).

We can see that this reduces the number of branches, k updates and comparisons by a factor of 4; if branches are costly, we can increase the unroll factor until they are no longer significant. There are other optimization benefits to loop unrolling, mostly due to the fact that it results in extra instructions in the loop body, which leads to greater opportunity for instruction scheduling as well as a host of related techniques. In Chapter 5, we will see examples of such optimization techniques for implementing *gemm*μ for ARM 64-bit architectures.

1.2.5 Reducing A and B Access Using Unroll and Jam with Register Blocking

The outer loops can be unrolled as readily as the innermost. If an outer loop is unrolled mechanically it would duplicate any loop inside of it, which is not usually helpful. For increased performance we therefore *jam* the contents of the outer loop into any inner loop(s), performing an operation called *unroll and jam* [5]. Figure 1.3a shows pseudocode for an unroll and jammed GEMM implementation where the M loop has been unrolled by 3 (i.e. $u_m = 3$), while the N loop is unrolled by 2 ($u_n = 2$), assuming M is a multiple of 3 and N is a multiple of 2.

The interesting thing about this is that our innermost loop now has additional opportunities for reducing memory access. We can use register assignment to reduce the access of C as before, but if we look at this loop, we see that elements of both A and B are used multiple times, which means we can save time by storing them in registers. To see this explicitly, Figure 1.3b shows register blocked code with unroll & jam performed as before.

Assuming register access is free, we can now compute the reduced memory cost of this algorithm. Our C access is unchanged, and is still at the minimum, though this may not be obvious at first glance. To show this, note that C is no longer accessed inside the K loop, so we need only worry about the first 2 loops. Since we have unrolled them, we don't execute them as many times as before. We execute the M loop only $\frac{M}{3}$ times, and the N-loop only $\frac{N}{2}$ times. At the top of the N loop we read 6 elements of C (due to unrolling and jamming of the outer two loops), and at the end of the N loop we write those same 6 elements back out. Therefore, total access costs is: $\frac{M}{3} \times \frac{N}{2} \times (6 + 6) = 2 \times M \times N$, just as before.

Let us now count the reads of A. For A we now have 3 reads in the innermost loop, giving as total accesses of: $\frac{M}{3} \times \frac{N}{2} \times K \times 3 = \frac{M \times N \times K}{2}$, which for $M = N = K$ means $\frac{N^3}{2}$, meaning we have cut our A accesses in half.

```
for (i=0; i < M; i += 3)
  for (j=0; j < N; j += 2)
    for (k=0; k < K; k++)
    {
      C(i,j)   += A(i,k)*B(k,j);
      C(i+1,j) += A(i+1,k)*B(k,j);
      C(i+2,j) += A(i+2,k)*B(k,j);

      C(i,j+1) += A(i,k)*B(k,j+1);
      C(i+1,j+1) += A(i+1,k)*B(k,j+1);
      C(i+2,j+1) += A(i+2,k)*B(k,j+1);
    }
```

(a)

```
for (i=0; i < M; i += 3)
  for (j=0; j < N; j += 2)
  {
    register c00,c10,c20,c01,c11,c21;
    c00=c10=c20=c01=c11=c21=0.0;
    for (k=0; k < K; k++)
    {
      register a0=A(i,k), a1=A(i+1,k);
      register a2=A(i+2,k);
      register b0=B(k,j), b1=B(k,j+1);

      c00 += a0 * b0;
      c10 += a1 * b0;
      c20 += a2 * b0;

      c01 += a0 * b1;
      c02 += a1 * b1;
      c03 += a2 * b1;
    }
    C(i,j)     += c00;
    C(i+1,j)   += c10;
    C(i+2,j)   += c20;
    C(i,j+1)   += c01;
    C(i+1,j+1) += c11;
    C(i+2,j+1) += c21;
  }
```

(b)

Figure 1.3: $u_m = 3, u_n = 2$ unroll & jammed GEMM: (a) no register blocking and (b) with register blocking for A, B, and C.

Finally we count the reads of B. For B we now have 2 reads in the innermost loop, giving as total accesses of: $\frac{M}{3} \times \frac{N}{2} \times K \times 2 = \frac{M \times N \times K}{3}$, which for $M = N = K$ means $\frac{N^3}{3}$, implying we have made only one third as many B accesses as our original implementation performed.

1.2.6 Generalizing Register Blocking Understanding

We now overview terminology to allow us to talk about the various unrolling factors. For each dimension, we indicate the unrolling factor with u subscripted by the dimension, so Figure 1.3a uses $u_m = 3$, $u_n = 2$, and $u_k = 1$ (K is not unrolled). From the simple counts given in the previous section, it is clear register blocking can reduce memory access strongly. Prior discussions should suffice to understand C, but to what degree can A or B access be improved?

The reason total A and B could be reduced it was that in jamming the M and N unrollings into the innermost loop, we exposed the opportunity to reuse a given element of these matrices multiple times, which allows us to load it to a register (requiring our normal memory

read) and then for every subsequent use, reuse it from the register (which we are counting as free).

Inside our innermost loop, each element of A is used u_n times, while each B element is used u_m times. Therefore, unrolling B's loop allows us to reuse elements of A, and vice versa. If $M = N$, then A and B have the same number of elements, and therefore it is typically true that the best register blocks are ones that are roughly square ($u_m \approx u_n$).

Note that $u_k > 1$ has no effect on memory usage, since we moved the accesses of C (that are controlled by the K loop) out of the innermost loop in a prior step!

With all this in mind, we can say that a register blocked GEMM requires $2N^2$ C memory accesses, while A requires $\frac{N^3}{u_n}$ memory reads, and B requires $\frac{N^3}{u_m}$ reads.

1.2.7 Example of a Complete and Correct *gemm*μ Implementation

Figure 1.4 shows a full and valid *gemm*μ implementation where $u_m = 3$, $u_n = 2$ and $u_k = 1$. One major difference between this implementation and our prior pseudo-code is that the loops indexed the rows and columns of the matrix in our pseudocode, but in *gemm*μ we are simply accessing all three storage locations in a purely sequential fashion. This is because prior to the invocation, the original column- (or row-) major array has been copied into a storage format to allow for purely sequential access.

We note that the framework automatically defines `TYPE` to be `float` for single precision real and complex, and `double` otherwise. The framework will also automatically compile each microkernel three times with differing `BETA` macro definitions in order to generate the required *gemm*μ variants, as shown in Table 1.2. We see that we use these compile-time

Table 1.2: *gemm*μ operation dictated by compile-time macro definition

CPP MACRO DEFINED	KERNEL RESULT
`BETA0`	$C \leftarrow AB$
`BETA1`	$C \leftarrow AB + C$
`BETAN1`	$C \leftarrow AB - C$

```
#define SZT size_t
#define CTYPE const TYPE
#define RTYPE register TYPE
void ATL_USERMM(SZT nmu, SZT nnu, SZT K, CTYPE *pA, CTYPE *pB, TYPE *pC,
                CTYPE *pAn, CTYPE *pBn, CTYPE *pCn)
{
   CTYPE *pB0 = pB;
   int m, n;
   const int incA = 3*K;    /* um*K */

   for (m=0; m < nmu; m++)
   {
      for (n=0; n < nnu; n++, pC += 6)
      {
         RTYPE rC00=0.0, rC10=0.0, rC20=0.0;
         RTYPE rC01=0.0, rC11=0.0, rC21=0.0;
         for (k=0; k < K; k++, pA += 3, pB += 2)
         {
            RTYPE rA0=(*pA), rA1=pA[1], rA2=pA[2];
            RTYPE rB0=(*pB), rB1=pB[1];
            rC00 += rA0 * rB0;
            rC10 += rA1 * rB0;
            rC20 += rA2 * rB0;
            rC01 += rA0 * rB1;
            rC11 += rA1 * rB1;
            rC21 += rA2 * rB1;
         }
         pA -= incA;  /* rewind 3 rows traversed in K-loop */
         #ifdef BETA0
            *pC   = rC00; pC[1] = rC10; pC[2] = rC20;
            pC[3] = rC01; pC[4] = rC11; pC[5] = rC21;
         #elif defined BETA1
            *pC   += rC00; pC[1] += rC10; pC[2] += rC20;
            pC[3] += rC01; pC[4] += rC11; pC[5] += rC21;
         #else /* BETA = -1 */
            *pC = C00 - *pC;  pC[1]=rC10-pC[1]; pC[2]=rC20-pC[2];
            pC[3]=rC00-pC[3]; pC[4]=rC10-pC[4]; pC[5]=rC20-pC[5];
         #endif
      }
      pB = pB0;    /* rewind pB to beginning */
      pA += incA; /* done with these 3 rows of A */
   }
}
```

Figure 1.4: Complete *gemmμ* with $u_m = 3$, $u_n = 2$ and $u_k = 1$.

macros in lines 29-38 of Figure 1.4. In the rest of this document, we refer to the computational components of the loop body of the of the innermost jammed loop (lines 19-26 of Figure 1.4) as the "unroll block".

1.3 Sequence of Research

In this dissertation the topics are ordered roughly by dependence. So, since the BLAS are used by everything we do, we discuss our research on BLAS first in Chapter 2. This seems to be the best approach for reader understanding, but it can be confusing to understand how things tie together since this was not the order in which we undertook the research. This section therefore serves to explain both our selection of topics, and what motivated the actual sequence of our investigation.

The main topic of my PhD research is using microkernels for increased parallelism, with particular emphasis on small- and medium-range problems. Therefore, the first topic investigated was parallel LU factorization, originally published in [37] and expanded on here in Chapter 4. While I was doing this research, my PhD advisor (Dr. Whaley) was working on redesigning the ATLAS tuning infrastructure for increased scale. The LU work was a proof-of-concept code to show the promise of increased microkernel usage for extreme-scale parallelism, and these two prongs of research had to be undertaken somewhat together, as our research results from LU provided guidance to keys and pitfalls for the general tuning framework.

We were next invited to extend our IPDPS paper for a special issue the Journal of Parallel and Distributed Computing (JPDC). We agreed, and looked to extend our parallel approach to the Cholesky factorization (we did not complete this parallel Cholesky research, but the fact that we started it is the reason for the Cholesky-related work reported in Section 3.2 and Appendix B.5), while also formalizing our microkernel definition and tuning so that others could use it. We were almost forty pages into this new paper when we ran into results we did not understand. In the time since the original paper, we had acquired a 24-core count Intel

Haswell-EP machine (X24 as described in Section 2.4). On this machine for the first time our algorithm did not outperform the commercial library MKL. The question then became why? All our work is based on *gemm*μ, and on this machine our *gemm*μ seemed slower than MKL, though it was hard to be sure since our performance results had huge variance. In the end, we had to pull the paper, because we could not publish a journal article with results we ourselves couldn't explain; we needed more time to investigate them than could be had with the special issue deadline.

It took quite some time to understand that the problematic results were coming from a complex combination of factors. The instability of our timings were mostly coming from the power savings that Haswell-EP does regardless of the user settings. We eventually realized that the timing instability that kept us from repeatable experiments could be reduced (though not eliminated) by substantially underclocking the machine.

Once this was taken care of, it was clear that our tuning framework had to be updated to better handle parallel scale. Our original framework tried to use serial timings to find the best case, and this was no longer adequate. This along with some lessons from my parallel LU research caused Dr. Whaley to significantly change the ongoing tuning redesign.

We needed a proven design before we could finish our interrupted JPDC paper, so in the meantime I began research on one of our most important future work ideas from the parallel LU effort. For our parallel LU, we statically block the matrix and distribute its blocks across the cores in a 2-D grid in order to fully exploit the cache. This requires us to choose a $(r \times c)$ process grid, and a block factor B. In our first publication, we simply wrote an almost exhaustive empirical search to find the best settings. This produces good results, but for large scale it could take several days of tuning, which would be untenable when added to ATLAS's already substantial tuning time.

Therefore the next area I worked extensively on was trying to build a model to predict the best (r, c, B) settings based on ATLAS's microkernel timings plus a rough computational

model. Our initial work on building computational models for microkernel-based routines is presented in Appendix B. Our first work in this area was for our parallel LU, and it is presented in Section B.1. As our results in this section show, our initial model was largely successful for asymptotic problems, but at smaller sizes where lower order terms like thread synchronization and communication time are important it could fail. Part of the parallel tuning framework redesign plan was to empirically measure and tune these parallel overheads, but that work had been put on hold for the $gemm\mu$ parallel tuning our previous research had mandated. Since this new parallel tuning required the fundamental redesign to finish before it could be added, the finishing of the parallel model had to be delayed until the package could provide the underlying empirically determined parallel communication cost estimates.

Microkernels can produce large wins for serial computation too, and so our next thought was to explore the modeling idea first in serial, which would allow us to build up the expertise necessary to tackle the parallel model with increased confidence. In the much simpler serial case, we could attempt to address a second problem highlighted by our initial work from Section B.1: In addition to areas where we suspected low-order parallel costs caused mispredicts, we had significant performance loss by modeling the affects of nearing the cache boundary too poorly. The problem is that most hardware uses pseudo-random replacement, and this cannot be modeled from first principles given easily established methods. However, it may be possible to do much better job with some combination heuristics and empirical timings, and we could investigate these cache affects in the simpler serial case. This would allow us to work with only one set of unknowns at a time, and we could return to the attempt to model parallel once we had established to what extent we could improve our models of behaviour around cache boundaries.

The only problem with this idea is that we had done all our main microkernel prototyping using parallel code, and didn't have a serious effort underway in serial. However, the ATLAS project provides widely-used serial BLAS, and this was the obvious place to investigate fully

using microkernels and then predicting the best block factor based on install-time timings. The serial BLAS could be written using our *gemm*μ microkernel using the GEMM-based BLAS [45, 46, 18, 32, 33, 47] approach. However, to address the small-case problems we were interested in, we would need to show that we could develop a series of BLAS-specific microkernels that could be so strongly based on our *gemm*μ work that this approach would be practical from a tuning and code maintenance perspective.

This then led to the chronologically last part of our research. Chapter 2 demonstrates the effectiveness of this approach, while Appendix B discusses some of our initial modeling work. Our goal in this research was to establish the feasibility of the research, and so we started with the hardest BLAS to write efficiently using a GEMM-based approach, which is TRSM (Section 2.3). TRSM has more dependencies and stability restriction than any other BLAS, and so if we could be affective here, we were sure this approach would work in general. Also, the exact TRSM case we were most optimizing here would be of enormous benefit to our ongoing parallel LU work when we had the tuning framework in place to finish it. We came up with an approach that requires almost no extra microkernel support, and yet still produces the extremely promising results reported in 2.4.

TRSM is very different from the rest of the BLAS, and so the microkernel approach we adopted for TRSM was qualitatively different than what we would want for the rest of the BLAS. As our last step we therefore did the work on TRMM (Section 2.2). With these two critical BLAS supported we believe we have demonstrated the validity of this new approach, and as soon as the new tuning framework research is complete, we should be able to present it in a publication.

This chronology has contextualized all the research activity presented here with the exception of Chapter 5. This work was a result of my taking an internship at ARM for a summer. ARM was interested in ATLAS performance on ARM processors, and determining if ATLAS-style tuning could be in the face of dynamic voltage and frequency scaling

(DVFS). These are topics of great interest for ATLAS research in general as well. This work was mostly done at ARM, and resulted in a poster presentation at SC15 [36].

1.4 Organization of Paper

The remainder of this paper is organized in the following way:

- Chapter 2 discusses how we can utilize ATLAS's *gemm*μ framework to optimize the performance of two Level-3 BLAS routines (TRMM and TRSM).
- Chapter 3 provides an introductory overview for optimizing the performance of two LAPACK routines (LU and Cholesky factorizations).
- Chapter 4 provides the details for our approach to improving the performance of parallel LU factorization using the *gemm*μ framework.
- Chapter 5 discusses implementing *gemm*μ for ARM 64-bit architectures, improving ATLAS performance for heterogeneous architectures, and an attempt to reliably auto-tuning in the presence of CPU frequency scaling.

Finally, we conclude this paper by summarizing the findings and the potential impact of this research on the computational sciences.

CHAPTER 2
PERFORMANCE OPTIMIZATION OF BLAS ROUTINES

In this chapter, we will discuss how we can optimize Level-3 BLAS routines using the GEMM microkernel (*gemm*μ) framework discussed in Section 1.2.2. Recall that the research presented in this chapter is not the first that we worked on, rather it serves as the basic block for understanding our overall research on microkernel-based linear algebra.

2.1 GEMM-based BLAS

From the mathematical point of view, all Level-3 BLAS routines e.g. triangular matrix-matrix multiply (TRMM), triangular solve (TRSM), etc. can be implemented using GEMM. The idea is to reuse the highly optimized GEMM routine to optimize other BLAS routines with minimum effort. This approach is known as GEMM-based BLAS which first appeared in a library called Superscalar BLAS [46, 47, 33]. We will overview the GEMM-based BLAS approach with TRMM and will later see that all Level-3 BLAS routines work in a similar fashion. As described in BLAS documentation [11], TRMM is a Level-3 BLAS routine that performs one of the following matrix-matrix operations:

$$B = alpha \times op(A) \times B, \quad \text{for } (SIDE = \text{Left})$$

or

$$B = alpha \times B \times op(A), \quad \text{for } (SIDE = \text{Right})$$

Where *alpha* is a scalar, A is a unit or non-unit, lower or upper triangular matrix and B is an $(M \times N)$-sized general matrix. The size of A is $M \times M$ in the first case or $N \times N$ in the second case and $op(A)$ is one of the following: $op(A) = A$ or $op(A) = A^T$ or $op(A) = A^H$ (only for complex data types), where A^T is the transpose of A and A^H is the conjugate-transpose of A. The multiplication result is written to the storage for B (i.e. the storage for B is used for both input and output). $SIDE$ is one of four parameters that denote all these variants of the TRMM operation. The four parameters are (descriptions from BLAS documentation):

1. $SIDE$: Specifies whether op(A) is on the left or right side of B.

   If $SIDE = L$, then the equation is: $B = alpha \times op(A) \times B$.

   If $SIDE = R$, then the equation is: $B = alpha \times B \times op(A)$.

2. $UPLO$: Specifies whether the matrix A is an upper or lower triangular.

   If $UPLO = U$, then A is an upper triangular matrix.

   If $UPLO = L$, then A is a lower triangular matrix.

3. $TRANSA$: Specifies the form of $op(A)$ to be used in the multiplication.

   If $TRANSA = N$, then $op(A) = A$.

   If $TRANSA = T$, then $op(A) = A^T$.

   If $TRANSA = C$, then $op(A) = A^H$ (only for complex datatype).

4. $DIAG$: Specifies whether the diagonal of A is unit or not.

   If $DIAG = U$, then A is assumed to be unit triangular (diagonal elements are assumed to be unit).

   If $DIAG = N$, then A is not unit triangular (diagonal elements must be read during computation).

Each BLAS routine supports two datatypes: real and complex and for each datatype, there are two precisions: single and double. For TRMM, there are 16 variants for each real precision and 24 variants for each complex precision. We will refer to each of these variants using the values of these parameters. For example, the LUNN variant of TRMM will denote $op(A)$ is on the Left side of B, A is Upper triangular, No-transposed ($op(A) = A$), and Non-unit-triangular. Since the B matrix is overwritten with the multiplication result, to avoid any confusion, we will use Z to specify the storage containing the result. The basic idea of GEMM-based TRMM in the Superscalar BLAS library is to statically partition the input

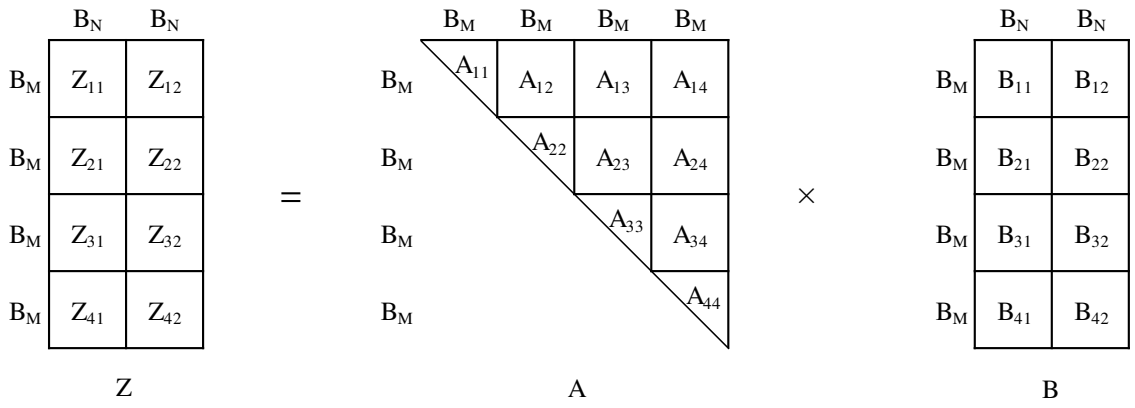


Figure 2.1: An example of TRMM with upper-triangular Matrix

matrices into blocks and compute the result one block at a time. Figure 2.1 shows an example of such blocking for LUNN-variant of TRMM. Blocks in matrix A are of size $B_M \times B_M$ and blocks in matrix B are of size $B_M \times B_N$. Note that A is partitioned as square blocks (i.e. $B_K = B_M$). If we use non-square blocking for A, we end up with trapezoidal diagonal blocks which significantly complicates the implementation. These blocking factors B_M and B_N in the Superscalar BLAS are selected based on the cache size of the system to maximize data reuse. We will later see that for the best performance, these block factors should be selected based on the combination of the features including the parallel scale, the cache size and the given problem size. Using such partitioning, the basic computational steps for the first column-panel of Z (i.e. $[Z_{11}\ Z_{21}\ Z_{31}\ Z_{41}]^T$) are shown in Figure 2.2. In steps 1, 3, 5, and 7, the operation is TRMM itself but only on a block. These operations can be done by using *reference* BLAS implementations (i.e. an unblocked, element-wise multiplication) or can be separately optimized (discussed in Section 2.2.2). Considering the GEMM updates in steps 2, 4, and 6, note that we use the block B_{41} in all the GEMM updates. Similarly, we use the block B_{31} in steps 2 and 4. A highly optimized GEMM routine usually copies the input operands to a cache-friendly storage format and then performs the multiplication. As shown in the above steps, the blocks B_{31} and B_{41} will possibly be copied two and three times, respectively. Typically, such copy overheads are only amortized when the operands are large. In this case, however, the output operand is only one block and the size of these blocks is limited by

1. $Z_{11} = A_{11}B_{11}$ ..... // TRMM
2. $Z_{11} = Z_{11} + A_{12}B_{21} + A_{13}B_{31} + A_{14}B_{41}$ ..... // GEMM

   $\Rightarrow$ may internally copy A_{12}, A_{13}, A_{14}, B_{21}, B_{31}, and B_{41}
3. $Z_{21} = A_{22}B_{21}$ ..... // TRMM
4. $Z_{21} = Z_{21} + A_{23}B_{31} + A_{24}B_{41}$ ..... // GEMM

   $\Rightarrow$ may internally copy A_{23}, A_{24}, B_{31}, and B_{41}
5. $Z_{31} = A_{33}B_{31}$ ..... // TRMM
6. $Z_{31} = Z_{31} + A_{34}B_{41}$ ..... // GEMM

   $\Rightarrow$ may internally copy A_{34} and B_{41}
7. $Z_{41} = A_{44}B_{41}$ ..... // TRMM

Figure 2.2: Computational steps of Superscalar TRMM for first column-panel of Z

the constraint that if block factors are increased with a fixed problem size, the block-sized TRMM performance become more important than the GEMM updates. On the other hand, the block factors cannot be too small either, since decreasing block size beyond a certain point can strongly reduce GEMM performance. Due to such constraints on the block factor, the copy overheads in steps 2, 4, and 6 can be significant and have noticeable impact on the overall performance. Note that all these steps are repeated for each column-panel of B and the GEMM steps will repeatedly copy the blocks of A. As the size of the input for TRMM grows, these repeated copy overheads put a limit on the maximum performance achievable by this GEMM-based TRMM (true for any statically blocked GEMM-based BLAS routines). This limitation can be overcome by using recursive blocking which is used in ATLAS [76, 74, 75]. The idea is to partition the input problem into two sub-problems and recursively multiply the sub-problems as shown in Figure 2.3. If the recursion is continued until the problem size becomes 1, the recursion overhead becomes too high [74]. To minimize this recursion overhead, the stopping criteria is set to a size where the entire problem can fit into some level of cache (e.g. Level-1 cache) but now the performance of these block-sized TRMM

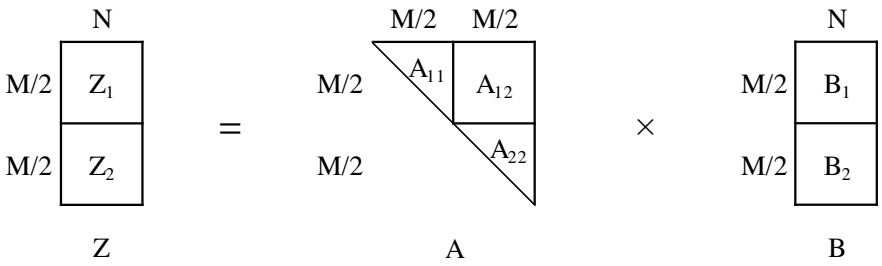


Figure 2.3: An example of recursive TRMM with upper-triangular matrix

will be important for small sized problems. The benefit of this recursive approach over the statically blocked GEMM-based approach is that the GEMM update required at the top level of recursion consists of large operands ($\frac{M}{2} \times N$ for the example) instead of fixed sized blocks and as the input problem size grows, the size of these operands grows as well. For a TRMM of size 10000×10000, the top level GEMM update (5000×10000) performs 50% of the total required computations. The two GEMM updates (2500×10000) in the next recursion level perform another 25% of the total required computations. Due to GEMM's high performance with such large operands, for asymptotic sized problems, the recursive approach outperforms the statically blocked iterative approach. However, both these approaches suffer from low performance for small sized problems due to significant copy overhead. Our approach to improve the performance of BLAS routines for all problem sizes is similar to GEMM-based statically blocked approach, except that instead of calling GEMM (that includes the copy), we will manage the copy ourselves to avoid duplicating it and use the ATLAS *gemm*μ for the updates. Figure **??** shows the computational steps for the first column-panel of Z for the example shown in Figure 2.1 but now using *gemm*μ. Note that the GEMM updates in steps 2, 4, and 6 are explicitly broken into sub-steps. In step 2(a), we copy the blocks A_{12}, A_{13}, and A_{14} only if we are working on the first column-panel of B. In step 2(b), we copy the blocks B_{21}, B_{31}, and B_{41}. Then we call the *gemm*μ instead of GEMM routine on the copied storage to apply the GEMM update. Note that the step of copying the result back to Z after the multiplication may or may not be needed depending on the implementation and the selected microkernels. Since different BLAS libraries implement GEMM in different

1. $Z_{11} = A_{11}B_{11}$ ..... // TRMM
2. $Z_{11} = Z_{11} + A_{12}B_{21} + A_{13}B_{31} + A_{14}B_{41}$ ..... // GEMM[†]
   (a) copy A_{12}, A_{13}, and A_{14} to *gemm*μ's required storage format
   (b) copy B_{21}, B_{31} and B_{41} to *gemm*μ's required storage format
   (c) three calls to *gemm*μ on the copied storage
   (d) may need to copy-back and accumulate the result to Z_{11}
3. $Z_{21} = A_{22}B_{21}$ ..... // TRMM
4. $Z_{21} = Z_{21} + A_{23}B_{31} + A_{24}B_{41}$ ..... // GEMM
   (a) copy A_{23} and A_{24} to *gemm*μ's required storage format
   (b) two calls to *gemm*μ on the copied storage
   (c) may need to copy-back and accumulate the result to Z_{21}
5. $Z_{31} = A_{33}B_{31}$ ..... // TRMM
6. $Z_{31} = Z_{31} + A_{34}B_{41}$ ..... // GEMM
   (a) copy A_{34} to *gemm*μ's required storage format
   (b) one call to *gemm*μ on the copied storage
   (c) may need to copy-back and accumulate the result to Z_{31}
7. $Z_{41} = A_{44}B_{41}$ ..... // TRMM

Figure 2.4: Computational steps of *gemm*μ-based TRMM for first column-panel of Z

[†]In actual implementation, the copies and the *gemm*μ updates are intermixed for better performance.

ways, this copy-back step may or may not be needed for a GEMM-based approach as well. Going back to the discussion, in step 4(a), we copy the next row-panel (i.e. blocks A_{23} and A_{24}) required for the GEMM. Note that for this update, we don't need to copy the blocks B_{31} and B_{41} since we already copied those in *gemm*μ's required format in step 2(b). Therefore, we can directly call the *gemm*μ using the newly copied A blocks and our previously copied B blocks. Similarly, in step 6, no copy is needed for B blocks. After we are done with the first column-panel of B (i.e. $[B_{11}\ B_{21}\ B_{31}\ B_{41}]^T$), we move on to the next column-panel and repeat the above steps with the exception that copy steps for the A blocks are now not needed since the all these blocks were copied during the operations for the first column-panel of Z. Figure 2.5 shows the steps for computing the second column-panel of Z. As we can see, that the steps 9, 11, and 13, only copy the current column-panel and no blocks of A, since they can reuse the previously copied blocks of A for the *gemm*μ computations. Note that these steps would be the same for any later column-panels of Z.

Since each Level-3 BLAS routine can be implemented using GEMM, we can use a similar approach to implement any Level-3 BLAS routine using *gemm*μ. In this chapter, we discuss two of such implementations in detail. Section 2.2 provides details about how each variant of TRMM is implemented using the *gemm*μ framework. In Section 2.3, we discuss how we can similarly improve the performance of triangular solve (TRSM), another widely used Level-3 BLAS operation, using the *gemm*μ framework.

2.2 *gemm*μ-based Triangular Matrix-matrix Multiply (TRMM)

In Section 2.1, we discussed how one variant of triangular matrix-matrix multiply (TRMM) works and how we can use the *gemm*μ framework to reduce overhead for small case performance. In this section, we discuss each of its variants in detail and how we can optimize each variant's performance using the *gemm*μ framework. As before, since B is also used for the output, we will use Z to denote the storage containing the multiplication results to avoid any confusion with input B.

8. $Z_{12} = A_{11}B_{12}$ ..... // TRMM
9. $Z_{12} = Z_{12} + A_{12}B_{22} + A_{13}B_{32} + A_{14}B_{42}$ ..... // GEMM
   (a) Copy B_{22}, B_{32} and B_{42} to *gemm*μ's required storage format
   (b) three calls to *gemm*μ on the copied storage
   (c) may need to copy-back and accumulate the result to Z_{12}
10. $Z_{22} = A_{22}B_{22}$ ..... // TRMM
11. $Z_{22} = Z_{22} + A_{23}B_{32} + A_{24}B_{42}$ ..... // GEMM
   (a) two calls to *gemm*μ on the copied storage
   (b) may need to copy-back and accumulate the result to Z_{22}
12. $Z_{32} = A_{33}B_{32}$ ..... // TRMM
13. $Z_{32} = Z_{32} + A_{34}B_{42}$ ..... // GEMM
   (a) one call to *gemm*μ on the copied storage
   (b) may need to copy-back and accumulate the result to Z_{32}
14. $Z_{42} = A_{44}B_{42}$ ..... // TRMM

Figure 2.5: Computational steps of *gemm*μ-based TRMM for second column-panel of Z

2.2.1 Our Statically Blocked Iterative TRMM

In this section, we discuss in detail on how we optimize each variant of TRMM using the *gemm*μ framework to provide high performance for problem sizes. Recall that in the example given in Section 2.1, we still needed an optimized microkernel for performing TRMM with the diagonal blocks. For this section, we assume that we already have an optimized microkernel for our discussion and we refer to those as TRMM microkernels (*trmm*μ). The details of the *trmm*μ are given in Section 2.2.2. Note that since the $DIAG$ parameter only changes the computation for the diagonal blocks for which we use the *trmm*μ, we will ignore this parameter for now and discuss how it is handled in Section 2.2.2.

2.2.1.1 Implementing LLN-variant of *gemm*μ-based TRMM

LLN denotes the operation: $B = alpha \times A \times B$, where A is a lower triangular matrix and B is a general matrix. Recall that we need a *gemm*μ to apply the GEMM updates as described in Section 2.1. Using the *gemm*μ framework, we need to find the best performing *gemm*μ and block factors B_M, B_N, and B_K with a restriction of $B_M = B_K$ (A is triangular: $M = K$). Recall that this restriction is not necessary but it simplifies the implementation (avoiding trapezoidal blocks) while providing very good performance for almost all problems. The selection of the block factors B_M, B_N, and B_K are dictated by the *gemm*μ and architecture features (e.g. parallel scale and cache size) of the system. For now, we can assume that only one combination of B_M, B_N, and B_K are selected. In Section B.3, we will discuss how multiple combinations can be helpful for small sized problems. Now, to perform the TRMM operation, the basic idea is to process one column-panel of B at a time while reusing the previously copied data with *gemm*μ. The basic implementation is shown in Figure 2.6.

To understand the implementation, consider the example shown in Figure 2.7. Note that we assume $alpha = 1$ to simplify the discussion. The triangular matrix A is partitioned into $(B_M \times B_M)$-sized blocks and the matrix B is partitioned into $(B_M \times B_N)$-sized blocks. As before, we will use Z to denote the storage, thus avoiding any confusion with the input B.

- **foreach** column-panel $Bcpan$ in B/Z
  1. copy each block of $Bcpan$ except the bottom-last to $gemm\mu$ storage
  2. **foreach** block $Bblk$ of $Bcpan$ from bottom to top
     $\rightarrow$ requires access of A from bottom row-panel to the top
     (a) call $trmm\mu$ to multiply the diagonal block of current row-panel $Arpan$ to $Bblk$
     (b) if first B-column-panel, copy the $Arpan$ omitting the diagonal block
     (c) call $gemm\mu$ to multiply copied $Arpan$ to copied part of $Bcpan$
     (d) copy back (accumulate) the result to $Bblk$

Figure 2.6: Basic computational steps for the LLN-variant of TRMM

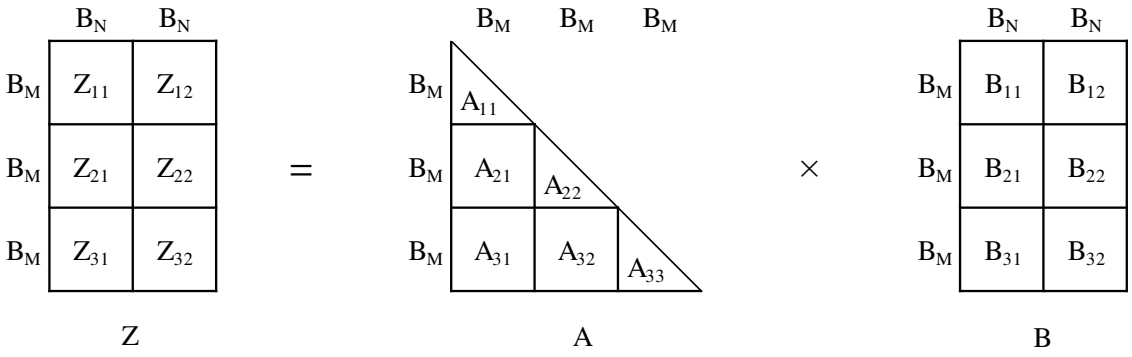


Figure 2.7: An example of TRMM: LLN-variant

The computational steps are shown in Figure 2.8 for the first column-panel of B. Note that in step 4, we reuse the previously copied B_{11}. For next column-panels of B, we repeat the same steps except reuse the previously copied blocks of A, instead of copying it again.

2.2.1.2 Implementing LUN-variant of *gemmμ*-based TRMM

LUN denotes the operation: $B = alpha \times A \times B$, where A is an upper triangular matrix and B is a general matrix. Like LLN case, we need a *gemmμ* to apply the GEMM updates and we use the *gemmμ* framework to find the best performing *gemmμ* and block factors B_M, B_N, and B_K with the same restriction of $B_M = B_K$. The basic implementation steps are shown in Figure 2.9. The implementation is similar to LLN-case except we start with the top block of every column-panel B. We already discussed in detail on how this approach works in Section 2.1.

2.2.1.3 Implementing RLN-variant of *gemmμ*-based TRMM

RLN denotes the operation: $B = alpha \times B \times A$, where A is a lower triangular matrix and B is a general matrix. Like LLN and LUN variants, we use the *gemmμ* framework to find the best performing *gemmμ* and the block factors B_M, B_N, and B_K. In this case, however, we impose a restriction of $B_N = B_K$ since A is on the right side of B and triangular ($N = K$). The idea for this case is to process one row-panel of B at a time. Note that accessing row-panels of B is not optimal for column-major storage (this is what ATLAS uses) but this allows us to only use one row panel of workspace for B, while copying the entire triangular matrix A into workspace. In the cases we are most trying to optimize with this new approach, the size of A is dominated by the size of B, so this approach makes sense. The basic implementation is shown in Figure 2.10.

To understand the implementation, consider the example shown in Figure 2.11. The triangular matrix A is partitioned into $(B_N \times B_N)$-sized blocks and the matrix B is partitioned into $(B_M \times B_N)$-sized blocks. Again, since B is reused for result, we use Z to avoid any confusion. We assume $alpha = 1$ to simplify discussion. The computational steps for the

1. $Z_{31} = A_{33}B_{31}$ ..... // $trmm\mu$
2. $Z_{31} = Z_{31} + A_{31}B_{11} + A_{32}B_{21}$ ..... // GEMM
   (a) copy A_{31} and A_{32} to $gemm\mu$'s required storage format
   (b) copy B_{11} and B_{21} to $gemm\mu$'s required storage format
   (c) two calls to $gemm\mu$ on the copied storage
   (d) may need to copy-back and accumulate the result to Z_{31}
3. $Z_{21} = A_{22}B_{21}$ ..... // $trmm\mu$
4. $Z_{21} = Z_{21} + A_{21}B_{11}$ ..... // GEMM
   (a) copy A_{21} to $gemm\mu$'s required storage format
   (b) one call to $gemm\mu$ on the copied storage
   (c) may need to copy-back and accumulate the result to Z_{21}
5. $Z_{11} = A_{11}B_{11}$ ..... // $trmm\mu$

Figure 2.8: Computational steps of $gemm\mu$-based LLN-variant of TRMM for first column-panel of Z

- **foreach** column-panel $Bcpan$ in B
  1. copy each block of $Bcpan$ except the top block to $gemm\mu$ storage
  2. **foreach** block $Bblk$ of $Bcpan$ from top to bottom
     $\rightarrow$ requires access of A from top row-panel to the bottom
     (a) call $trmm\mu$ to multiply the diagonal block of current row-panel $Arpan$ to $Bblk$
     (b) if first B-column-panel, copy the $Arpan$ omitting the diagonal block
     (c) call $gemm\mu$ to multiply copied $Arpan$ to copied part of $Bcpan$
     (d) copy back (accumulate) the result to $Bblk$

Figure 2.9: Basic computational steps for the LUN-variant of TRMM

- **foreach** row-panel $Brpan$ in B
  1. copy each block of $Brpan$ except the left-most block to $gemm\mu$ storage
  2. **foreach** block $Bblk$ of $Brpan$ from left to right
     $\rightarrow$ this requires access of A from left column-panel to the right
     (a) call $trmm\mu$ to multiply the diagonal block of current column-panel $Acpan$ to $Bblk$
     (b) if first B-panel, copy the $Acpan$ except the diagonal block
     (c) call $gemm\mu$ to multiply copied part of $Brpan$ to copied $Acpan$
     (d) copy back (accumulate) the result to $Bblk$

Figure 2.10: Basic computational steps for the RLN-variant of TRMM

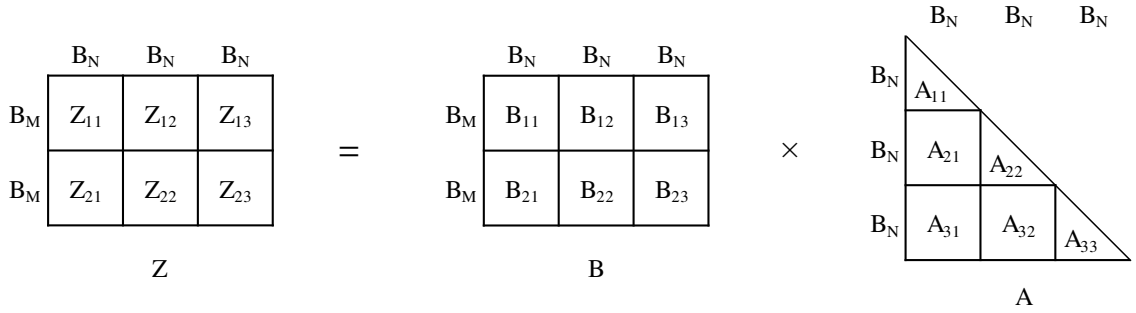


Figure 2.11: An example of TRMM: RLN-variant

first row-panel of B is shown in Figure 2.12. Again, in step 4, we reuse the block B_{13} that we copied in step 2. For next row-panels of B, we repeat the same steps except reuse the previously copied blocks of A, instead of copying it again.

2.2.1.4 Implementing RUN-variant of *gemmμ*-based TRMM

RUN denotes the operation: $B = alpha \times B \times A$, where A is an upper triangular matrix and B is a general matrix. Like RLN case, we use the *gemmμ* framework to find the best performing *gemmμ* and the block factors B_M, B_N, and B_K with the same restriction of $B_N = B_K$. Similar to RLN case, we process one row-panel of B at a time. The basic implementation is shown in Figure 2.13.

To understand the implementation, consider the example shown in Figure 2.14. Similar to RLN case, the triangular matrix A is divided into $(B_N \times B_N)$-sized blocks and B is divided into $(B_M \times B_N)$-sized blocks. Again, to avoid confusion with B, we use Z to denote the storage for the result. Also to simplify discussion, we assume $alpha = 1$. The computational steps for the first row-panel of B is shown in Figure 2.15. For next row-panels of B, we repeat the same steps except reuse the previously copied blocks of A, instead of copying it again.

2.2.1.5 Other Variants of TRMM Using Reflection

Ignoring the $DIAG$ parameter[1], there are eight other variants of TRMM (including complex-only variants). Among these eight variants, the four variants: LLT, LUT, RLT, and RUT denote the same operations as LLN, LUN, RLN, and RUN respectively with the exception that the triangular A is stored in a transposed storage. Since the *gemmμ* copy routines allow us specify the input matrix as stored in a transposed storage, we can utilize the reflecting case (e.g. use LUN implementation for LLT) except that when we copy the blocks of A, we specify that A is stored as transposed. We can use the same technique to support the other four variants: LLC, LUC, RLC, RUC by specifying that A is stored in transpose-conjugate format during copy. Table 2.1 summarizes the variants and the copy settings to use to support

[1]The changes for the $DIAG$ parameter are discussed in Section 2.2.2.

1. $Z_{11} = B_{11}A_{11}$ ..... // *trmm*μ
2. $Z_{11} = Z_{11} + B_{12}A_{21} + B_{13}A_{31}$ ..... // GEMM
   (a) copy A_{21} and A_{31} to *gemm*μ's required storage format
   (b) copy B_{12} and B_{13} to *gemm*μ's required storage format
   (c) two calls to *gemm*μ on the copied storage
   (d) may need to copy-back and accumulate the result to Z_{11}
3. $Z_{12} = B_{12}A_{22}$ ..... // *trmm*μ
4. $Z_{12} = Z_{12} + B_{13}A_{32}$ ..... // GEMM
   (a) copy A_{32} to *gemm*μ's required storage format
   (b) one call to *gemm*μ on the copied storage
   (c) may need to copy-back and accumulate the result to Z_{12}
5. $Z_{13} = B_{13}A_{33}$ ..... // *trmm*μ

Figure 2.12: Computational steps of *gemm*μ-based RLN-variant of TRMM for first row-panel of Z

- **foreach** row-panel *Brpan* in B
  1. copy each block of *Brpan* except the right-most block to *gemm*μ storage
  2. **foreach** block *Bblk* of *Brpan* from right to left
     $\rightarrow$ requires access of A from right column-panel to the left
     (a) call *trmm*μ to multiply the diagonal block of current column-panel *Acpan* to *Bblk*
     (b) if first B-panel, copy the *Acpan* omitting the diagonal block
     (c) call *gemm*μ to multiply copied part of *Brpan* to copied *Acpan*
     (d) copy back (accumulate) the result to *Bblk*

Figure 2.13: Basic computational steps for the RUN-variant of TRMM

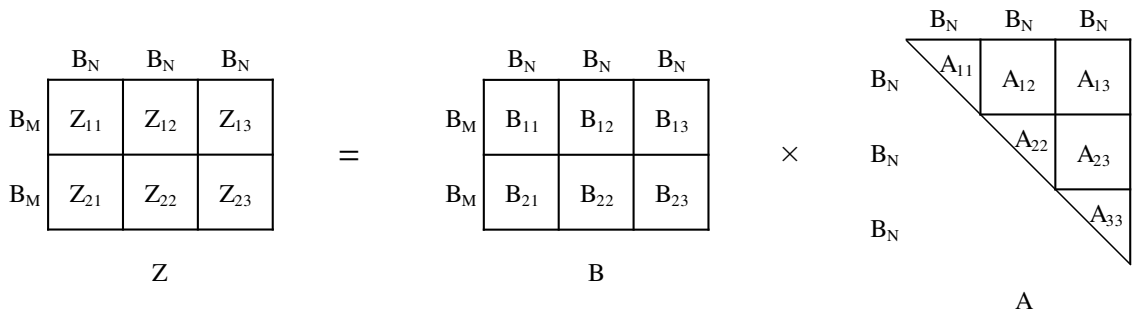


Figure 2.14: An example of TRMM: RUN-variant

1. $Z_{13} = B_{13}A_{33}$ ..... // $trmm\mu$
2. $Z_{13} = Z_{13} + B_{11}A_{13} + B_{12}A_{23}$ ..... // GEMM
   (a) copy A_{13} and A_{23} to $gemm\mu$'s required storage format
   (b) Copy B_{11} and B_{12} to $gemm\mu$'s required storage format
   (c) two calls to $gemm\mu$ on the copied storage
   (d) may need to copy-back and accumulate the result to Z_{13}
3. $Z_{12} = B_{12}A_{22}$ ..... // $trmm\mu$
4. $Z_{12} = Z_{12} + B_{11}A_{12}$ ..... // GEMM
   (a) copy A_{12} to $gemm\mu$'s required storage format
   (b) one call to $gemm\mu$ on the copied storage
   (c) may need to copy-back and accumulate the result to Z_{12}
5. $Z_{11} = B_{11}A_{11}$ ..... // $trmm\mu$

Figure 2.15: Computational steps of $gemm\mu$-based RUN-variant of TRMM for first row-panel of Z

all these variants. In Section 2.2.2, we will see that to support all the variants of *trmm*μ, we can use the same technique thus having only 4 variants of *trmm*μ and supporting all 16 (or 24 for complex) variants by only changing the copy microkernels for the triangular matrix.

2.2.2 Implementing All Variants of *trmm*μ

A naive implementation of *trmm*μ would be to zero-pad the triangular block to make it a full block and just use *gemm*μ to multiply it to the block of B. However with this approach, 50% of the computations done by the *gemm*μ are useless because they are known to be zero. Note that a microkernel can be made for each Level-3 BLAS operation in a similar way (i.e. incurring useless computations) with the exception of triangular solve (TRSM), discussed in Section 2.3. Our idea is to achieve high performance from these microkernels with much reduced zero-padding and useless computations. As described in Section 1.2.2, ATLAS has a *gemm*μ generator that produces architecturally optimized microkernels that ATLAS uses to build its full GEMM. We adapted this generator to produce all variants of *trmm*μ. These microkernels are then used to build the full TRMM as described in Section 2.2. This section provides implementation details of our *trmm*μ generator. We will describe in detail how we adapt the *gemm*μ generator and the data copy microkernel generator to support all variants of TRMM. Recall that the $DIAG$ parameter denotes whether A is a unit or non-unit triangular matrix. In our *trmm*μ for unit-triangular variants, we put ones on the diagonal

Table 2.1: Summary for supporting transpose and conjugate-transpose variants of TRMM through reuse of no-transpose variants

TRMM Variants	Variant to use	Transpose settings for copy
LLT	LUN	Transpose
LUT	LLN	Transpose
RLT	RUN	Transpose
RUT	RLN	Transpose
LLC	LUN	Conjugate-transpose
LUC	LLN	Conjugate-transpose
RLC	RUN	Conjugate-transpose
RUC	RLN	Conjugate-transpose

elements during copy and treat them as non-unit elements during multiplication. Although this simplification incurs an extra multiplication for each diagonal element, it also halves the number of different microkernel variants required. Due to this trivial difference among these unit and non-unit variants (e.g. LLNN vs. LLNU), we will ignore the $DIAG$ parameter in our discussions for all variants of *trmm*μ. Note that all the microkernels discussed here are supposed to perform block-sized TRMM operation i.e. the output is at most an $(B_M \times B_N)$-sized matrix.

2.2.2.1 *trmm*μ for LLN-variant

LLN denotes the operation: $B = alpha \times A \times B$, where A is the lower triangular matrix and B is a general matrix. For the *trmm*μ, we need to select a combination of u_m, u_n and u_k which denotes the unroll factor of the microkernel in the M, N, and K dimensions (respectively) as ATLAS does for the *gemm*μ. This selection can be done using a smart search that can exploit architectural features and limitations e.g. the number of registers available, the type of the available floating point unit (FPU). Since A is a triangular matrix, $B_K = B_M$. To limit the number of corner cases for keeping track of the diagonals, we impose a restriction that either u_m is a multiple of u_k or u_k is a multiple of u_m. We will discuss the case where u_m is a multiple of u_k i.e. $u_m = i \times u_k$ where $i \geq 1$. The implementation of the other case (i.e. $u_k = i \times u_m$ where $i > 1$) is slightly more complicated as it requires handling trapezoidal unroll-blocks and is discussed in Appendix A.

Copy microkernel changes: Since B is a general matrix, we can generate B-copy microkernels to copy B to GEMM's required storage format (we will generally call this GEMM-storage) using GEMM's own copy microkernel generator where the microkernel takes u_n-columns of B at a time and packs them together to facilitate SIMD vectorized GEMM. For the triangular A, we need to adapt GEMM's copy microkernel generator for A where the microkernel takes u_m-rows at a time and packs them together. The adaptation is required since the number of elements is varying in consecutive rows of A. One way would be to pad

the rows with zeros so that each row will have same number of elements but as we discussed before that this approach incurs nearly 50% useless storage. Instead, we will pad the rows at the end only up to a number where it is a multiple of u_m. Consider the example shown in Figure 2.16. The triangular matrix A is divided into u_m-sized row-panels, and each row-panel consists of a triangular unroll-block (e.g. T_1, T_2, etc.) and a rectangular row-panel (e.g. A_2, A_3, etc.). For each row-panel, we copy the rectangular row-panel (if any) just like a *gemm*μ copy routine does. However, for the triangular unroll-blocks, we copy them specially to make it a $(u_m \times u_m)$-sized unroll-block with zero-padding. To maximize the cache utilization, these varying (increasing) sized row-panels are packed together. So, in our example, the data are copied in the following order: padded-T_1, A_2, padded-T_2, A_3, and padded-T_3. Note that in the actual implementations, the order we do the copy of these unroll-blocks may not be the same as they are laid out spatially in memory (to minimize possible page faults accessing A).

***trmm*μ changes**: Once we copy the triangle A to microkernel-friendly compact storage using our own triangular copy routine, the *trmm*μ changes are trivial. The pseudocode outlining the changes required from a *gemm*μ is shown in Algorithm 2.1. The key considerations are:

- since row-panels (also known as K-panels) of A have increasing number of unroll-blocks, the loop bound for K-loop also increases for each panel (lines 11 and 12).

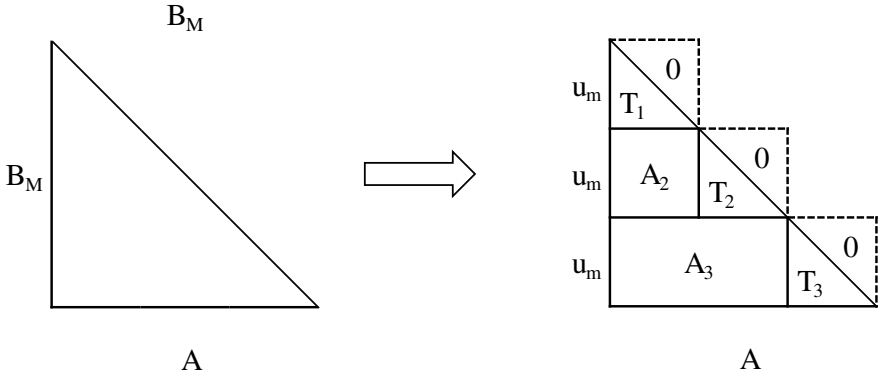


Figure 2.16: LLN-variant: partitioning and zero-padding the lower-triangular A

Algorithm 2.1: Pseudocode of looping and pointer updates for LLN-variant of *trmmμ* (changes from a *gemmμ* are highlighted in bold)

```
1  Function trmmμ-LLN(nmu, nnu, K, pA, pB, pC):
2  {
       arguments: nmu : no. of mu-sized row-panels in A matrix
                  nnu : no. of nu-sized column-panels in B matrix
                  K : size of the dimension common to A and B
                  pA : pointer to the triangular matrix
                  pB : pointer to the general matrix
                  pC : pointer to the result matrix
3      pA0 ← pA
4      pB0 ← pB
       /* Set the skips needed                                          */
5      Askip ← mu × mu
6      Bskip ← nu × (K − mu)
7      for i ← 1 to nmu do
8       {
9          for j ← 1 to nnu do
10          {
11             Kbound ← min(K, i × mu)
12             for k ← 1 to Kbound do
13              {
14                 ⟹ perform one unroll-block multiplication
15                 pA ← pA + mu
16                 pB ← pB + nu
17              }
18             pB ← pB + Bskip          /* skip to correct column-panel */
19             pA ← pA0
20             pC ← pC + (mu × nu)
21          }
22         pA0 ← pA0 + Askip              /* skip to correct row-panel */
23         pA ← pA0
24         pB ← pB0
           /* update skips for next row-panel                           */
25         Askip ← Askip + (mu × mu)
26         Bskip ← Bskip − (mu × nu)
27      }
28 }
```

- at the end of each K-loop, we need to advance the B-pointer to skip the known-zero unroll-blocks of $A \times B$, so it points to the beginning of the correct column-panel of B (line 18).

- at the end of each N-loop, we need to move the pointer to A to the beginning to the correct row-panel (line 22), and we also need to update the amount of skip needed for the next K-loop (lines 25 and 26).

In the example shown in Figure 2.17, to compute the result of Z_{11}, only one unroll-block multiplication ($T_1 \times B_{11}$) is needed in the K-loop. Note that after zero-padding T_1 is treated as a regular unroll-block during multiplication. After Z_{11} is computed, the pointer to A is moved back to T_1 and the pointer to B is skipped to B_{12}. Note that the magnitude for this skip starts with two unroll-blocks. After the end of first N-loop, we start using the second row-panel (i.e. A_2,T_2) to compute Z_{21} and Z_{22}. Note that for each of these unroll-blocks require two unroll-blocks multiplications. So, the magnitude of the skip needed at the end of next K-loop is decreased by one unroll-block.

2.2.2.2 *trmmμ* for LUN-variant

The operation is similar to LLN-variant except that A is an upper triangular matrix.

Copy microkernel changes: Like LLN-variant, B-copy microkernel can be generated using GEMM's own B-copy microkernel generator. Unlike LLN-variant, the triangle A has decreasing number of elements on consecutive rows. Consider the example shown in Fig-

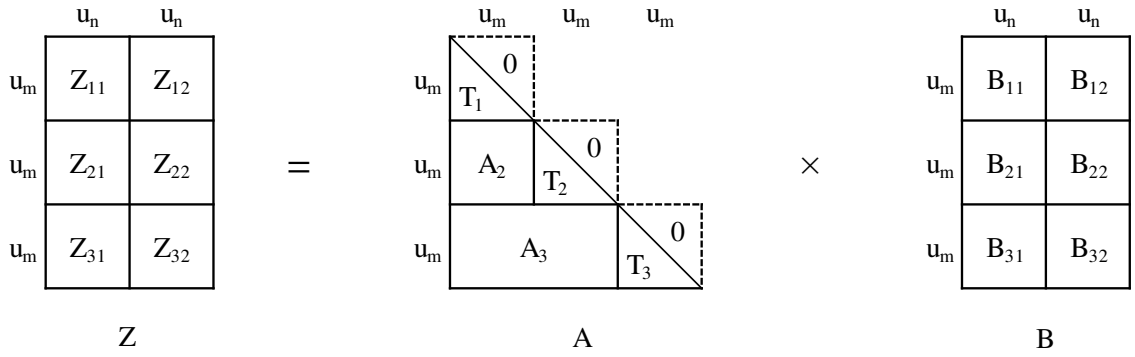


Figure 2.17: An example for LLN-variant *trmmμ*

ure 2.18. The triangular A is divided into u_m-sized row-panels and each row-panel is padded at the beginning to the smallest multiple of u_k to minimize zero computations. Then, these decreasing sized row-panels are packed together (i.e. padded-T_1, A_1, padded-T_2, A_2 and padded-T_3) to maximize the cache utilization. Note, we imposed the same restriction on u_m and u_k as the LLN-variant to help keep track of the diagonal unroll-blocks.

***trmmμ* changes**: Once we copy the triangle A to GEMM-friendly compact storage using our own triangular copy routine, the *trmmμ* changes are trivial. The pseudocode outlining the changes required from a *gemmμ* is shown in Algorithm 2.2. The key considerations are:

- since row-panels (also known as K-panels) of A have decreasing number of unroll-blocks, the loop bound for K-loop also decreases for each panel (lines 11 and 12)

- at the end of each K-loop, we need to advance the B-pointer to to skip the known-zero unroll-blocks of $A \times B$, so it points to the beginning of the correct column-panel of B (line 18).

- at the end of each N-loop, we need to advance the A-pointer to the beginning to the correct row-panel (line 22), update the amount of skip needed for the next K-loop (lines 25 and 26), and advance the B-pointer to the correct row-block for the next K-loop (line 27).

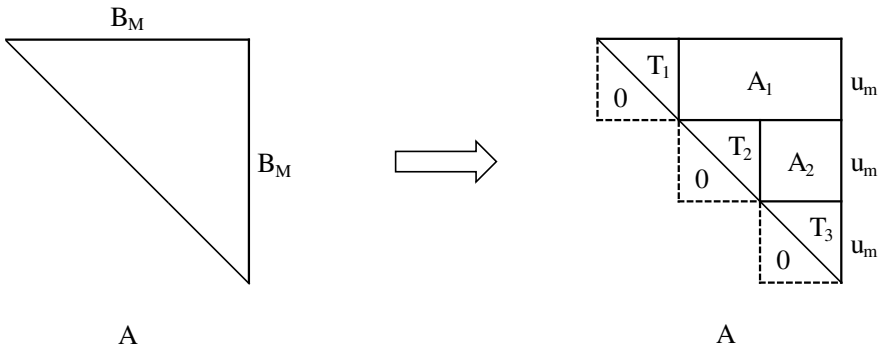


Figure 2.18: LUN-variant: partitioning and zero-padding the upper-triangular A

Algorithm 2.2: Pseudocode of looping and pointer updates for LUN-variant of *trmmμ* (changes from a *gemmμ* are highlighted in bold)

```
1  Function trmmμ-LUN(nmu, nnu, K, pA, pB, pC):
2  {
       arguments: nmu : no. of mu-sized row-panels in A matrix
                  nnu : no. of nu-sized column-panels in B matrix
                  K : size of the dimension common to A and B
                  pA : pointer to the triangular matrix
                  pB : pointer to the general matrix
                  pC : pointer to the result matrix
3      pA0 ← pA
4      pB0 ← pB
       /* Initialize the skips needed                                  */
5      Askip ← mu × K
6      Bskip ← 0
7      for i ← 1 to nmu do
8       {
9          for j ← 1 to nnu do
10          {
11             Kstart ← (i − 1) × mu
12             for k ← Kstart to K do
13              {
14                 ⟹ perform one unroll-block multiplication
15                 pA ← pA + mu
16                 pB ← pB + nu
17              }
18             pB ← pB + Bskip          /* skip to correct column-panel */
19             pC ← pC + (mu × nu)
20             pA ← pA0
21          }
22         pA0 ← pA0 + Askip              /* skip to correct row-panel */
23         pA ← pA0
24         pB ← pB0
           /* update skips for next row-panel and advance pB to correct
              block                                                     */
25         Askip ← Askip − (mu × mu)
26         Bskip ← Bskip + (mu × nu)
27         pB ← pB + Bskip
28      }
29 }
```

In the example shown in Figure 2.19, to compute the result of Z_{11}, three unroll-blocks multiplications ($T_1 \times B_{11} + A_1 \times \ [B_{21}\ B_{31}]^T$) are needed in the K-loop. After that the pointer to A is moved back to T_1 and the pointer to B in this case is already at the start of the next column-panel (B_{12}). So, the magnitude for the skip needed starts with zero and later increased. After the end of first N-loop, like LLN, we start working on the second row-panel (i.e. T_2,A_2) and each unroll-block of the result (i.e. Z_{21} and Z_{22}) require two unroll-block computations. Therefore, the magnitude of the skip needed at the end of next K-loop is increased by one unroll-block.

2.2.2.3 *trmmμ* for RLN-variant

RLN denotes the operation: $B = alpha \times B \times A$, where A is the lower triangular matrix and B is a general matrix. Like other *trmmμ*, we need to select a combination of u_m, u_n and u_k which denotes the unroll factor of the microkernel in M, N, and K dimensions (respectively). The selection can be done by using a search similar to LLN-variant's smart search. Since A is a triangular matrix, $B_K = B_N$. So, in this case, since triangular A is treated as B for GEMM, we impose a restriction that either u_n is a multiple of u_k or u_k is a multiple of u_n to limit the number of corner cases for keeping track of the diagonal unroll-blocks.

Copy microkernel changes: Since B is a general matrix and it is treated as the A operand for GEMM, we can generate A-copy microkernels to copy it to GEMM's required storage using GEMM's own copy microkernel generator where the microkernel takes u_m-

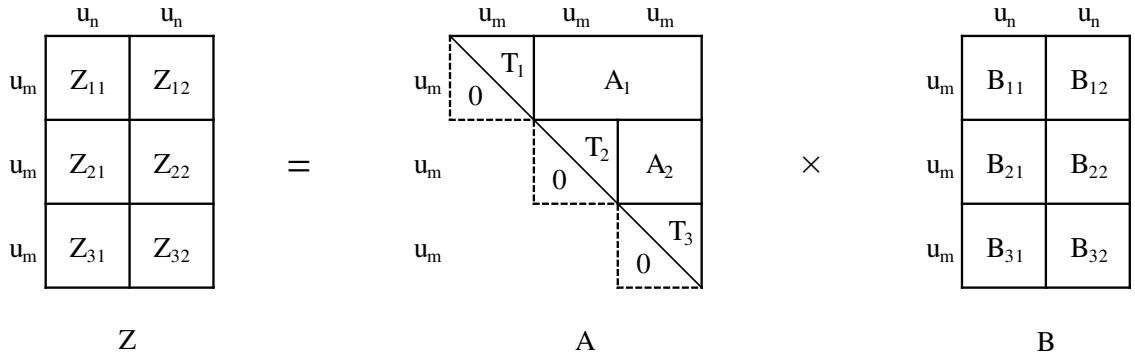


Figure 2.19: An example for LUN-variant *trmmμ*

rows of B at a time and packs them together to facilitate SIMD vectorized GEMM. For the triangular A (it is treated as B operand for GEMM), we need to adapt GEMM's copy microkernel generator for B where the microkernel takes u_n-columns at a time and packs them together. The adaptation is required since the number of elements is varying (decreasing) on consecutive columns of A. Analogous to LLN-variant, one way would be to pad the columns with zeros so that each column will have same number of elements but as discussed before, this incurs nearly 100% extra(zero) computations. Instead, we will pad at the beginning of the columns only up to a number where it is a multiple of u_k. Consider the example shown in Figure 2.20. The triangular matrix A is divided into u_n-sized column-panels, and each column-panel is padded at the beginning to the smallest multiple of u_k to minimize the zero computations. All these varying (decreasing) sized column-panels are packed together to maximize the cache utilization.

***trmmμ* changes**: Once we copy the triangle A to GEMM-friendly compact storage (treated as B) using our own triangular copy routine, the *trmmμ* changes are trivial. The pseudocode outlining the changes required from a *gemmμ* is shown in Algorithm 2.3. The key considerations are:

- since column-panels (also known as K-panels) of triangular A (treated as B) have decreasing number of unroll-blocks, the loop bound for K-loop also decreases for each panel (line 11 and 12).

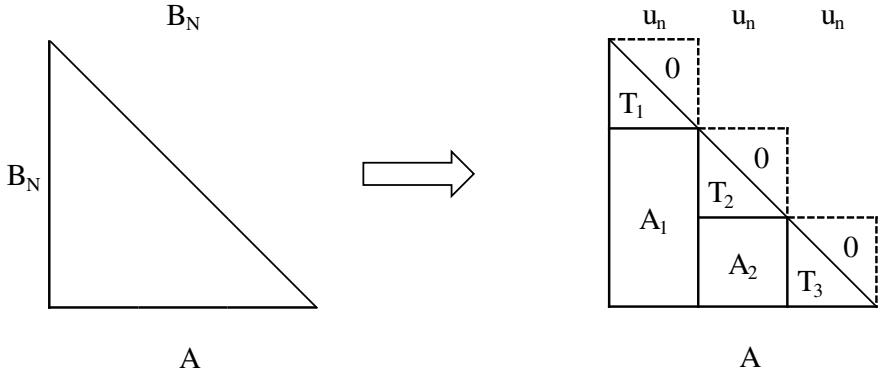


Figure 2.20: RLN-variant: partitioning and zero-padding the lower-triangular A

Algorithm 2.3: Pseudocode of looping and pointer updates for RLN-variant of *trmmμ* (changes from a *gemmμ* are highlighted in bold)

```
1  Function trmmμ-RLN(nmu, nnu, K, pA, pB, pC):
2  {
       arguments: nmu : no. of mu-sized row-panels in A matrix
                  nnu : no. of nu-sized column-panels in B matrix
                  K : size of the dimension common to A and B
                  pA : pointer to the general matrix
                  pB : pointer to the triangular matrix
                  pC : pointer to the result matrix
3      pA0 ← pA
4      pB0 ← pB
5      Apan ← mu × K                                  /* size of one row-panel */
6      for i ← 1 to nmu do
7       {
          /* Set the skips needed                                              */
8         Askip ← 0
9         for j ← 1 to nnu do
10         {
11           Kstart ← (j − 1) × nu
12           for k ← Kstart to K do
13            {
14              ⟹ one unroll-block multiplication
15              pA ← pA + mu
16              pB ← pB + nu
17            }
             /* update skip for next column-panel                             */
18           Askip ← Askip + (mu × nu)
19           pA ← pA0 + Askip                    /* skip to correct block */
20           pC ← pC + (mu × nu)
21        }
22        pB ← pB0
23        pA0 ← pA0 + Apan                      /* move to next row-panel */
24        pA ← pA0
25     }
26 }
```

- at the end of each K-loop, the pointer to B (pointing to the triangle) is already at the correct location since the triangle is stored in a compact storage, so no skips needed for pointer to B. However, the pointer to A (pointing to the general B matrix) cannot move back to the original start of the panel. It needs to skip some unroll-blocks to the correct part of the panel that are needed for the next computations (line 18). We also need to update the amount of skip needed for the next iteration (line 19).

In the example shown in Figure 2.21, to compute the result of Z_{11}, three unroll-blocks multiplications ($B_{11} \times T_1 + \ [B_{12}\ B_{13}]^T \ \times A_1$) are needed in the K-loop. After that the pointer to A is moved back to B_{11} and skipped to B_{12} and the pointer t o B (pointing to the triangle A) is unchanged since it already points to T_2 which is what we need for the next multiplication. The skip magnitude for the pointer to A also needs to be increased by one unroll-block here as well for the next iteration. After the end of first N-loop, the skip magnitude is reset for the next row panel of Z.

2.2.2.4 *trmmμ* for RUN-variant

The operation is similar to RLN-variant except that A is an upper triangular matrix.

Copy microkernel changes: Like RLN-variant, B-copy microkernel can be generated using GEMM's own A-copy microkernel generator. Unlike RLN-case, the triangle A has increasing number of elements on consecutive columns. Consider the example shown in Figure 2.22. The triangular A is divided into u_n-sized column-panels and each column-panel is padded at the end to the smallest multiple of u_k to minimize zero computations. Then, these

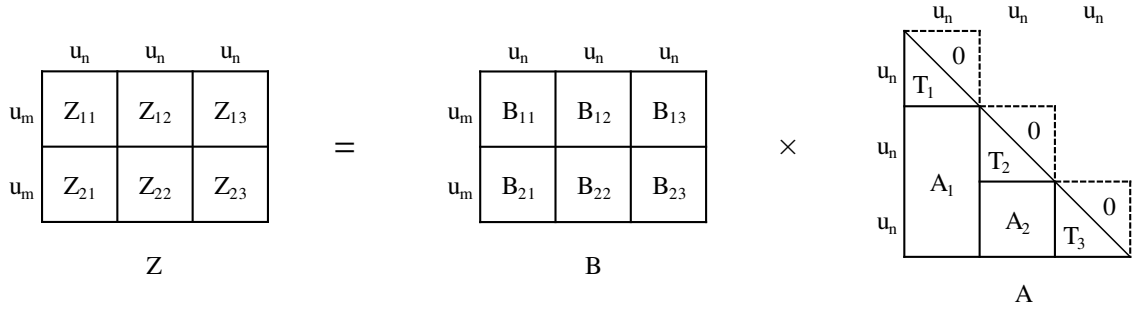


Figure 2.21: An example for RLN-variant *trmmμ*

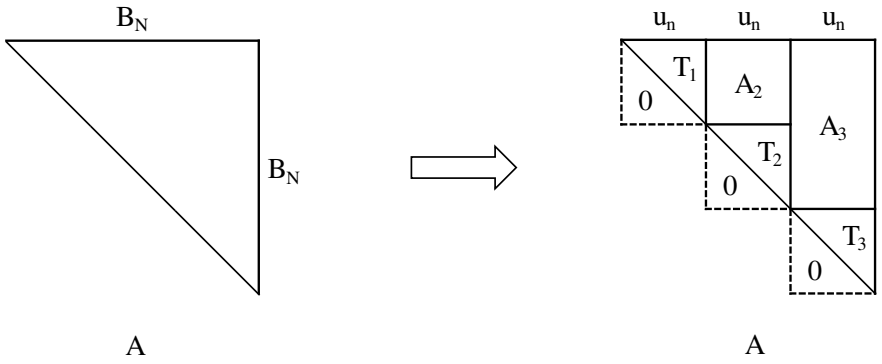


Figure 2.22: RUN-variant: partitioning and zero-padding the upper-triangular A

increasing sized column-panels are packed together to maximize the cache utilization. We imposed the same restriction on u_n and u_k as RLN-variant to help keep track of the diagonal unroll-blocks.

***trmmμ* changes**: Once we copy the triangle A to GEMM-friendly i compact storage (treated as B) using our own triangular copy routine, the *trmmμ* changes are trivial. The pseudocode outlining the changes required from a *gemmμ* is shown in Algorithm 2.4. The key considerations are:

- since column-panels (also known as K-panels) of triangular A (treated as B) have different number of unroll-blocks, the loop bound for K-loop is different for each panel (line 10 and 11).

- at the end of each K-loop, the pointer to B (pointing to the triangle) is already at the correct location since triangle is stored in a compact storage. So, no skips needed for pointer to B. The pointer to A, in this case, can move back to the original start of the panel without any skip. So, no changes needed for this either.

In the example shown in Figure 2.23, to compute the result of Z_{11}, only one unroll-block multiplication ($B_{11} \times T$) is needed in the K-loop. After that the pointer to A is moved back to B_{11} and the pointer to B is unchanged since it already points to A_2. Since no skip is needed for any pointers in this case, pointer to A is moved to the next row-panel where pointer to B is moved to the start of the triangle (like a regular *gemmμ*).

Algorithm 2.4: Pseudocode of looping and pointer updates for RUN-variant of *trmmμ* (changes from a *gemmμ* are highlighted in bold)

```
1  Function trmmμ-RUN(nmu, nnu, K, pA, pB, pC):
2  {
       arguments: nmu : no. of mu-sized row-panels in A matrix
                  nnu : no. of nu-sized column-panels in B matrix
                  K : size of the dimension common to A and B
                  pA : pointer to the general matrix
                  pB : pointer to the triangular matrix
                  pC : pointer to the result matrix
3      pA0 ← pA
4      pB0 ← pB
5      Apan ← mu × K                          /* size of one row-panel */
6      for i ← 1 to nmu do
7       {
8          for j ← 1 to nnu do
9           {
10             Kbound ← min(K, j × nu)
11             for k ← 1 to Kbound do
12              {
13                 ⟹ perform one unroll-block multiplication
14                 pA ← pA + mu
15                 pB ← pB + nu
16              }
17             pC ← pC + (mu × nu)
18             pA ← pA0
19          }
20         pB ← pB0
21         pA0 ← pA0 + Apan                   /* move to next row-panel */
22         pA ← pA0
23      }
24 }
```

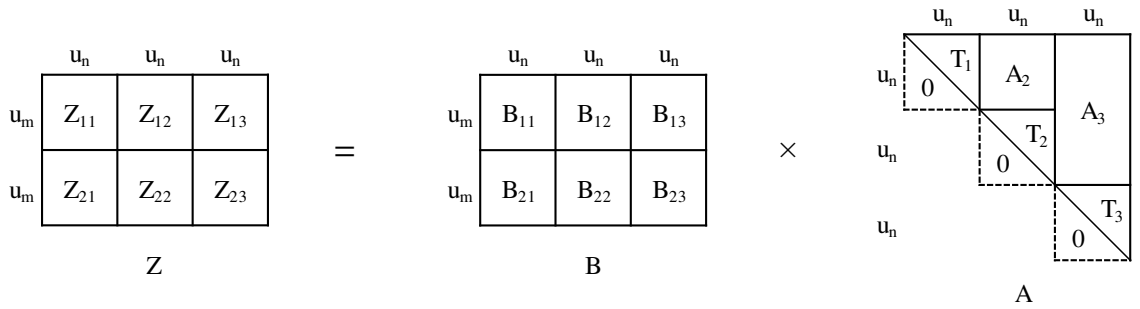


Figure 2.23: An example for RUN-variant *trmmμ*

2.2.2.5 Other *trmmμ* Variants Using Reflection

As discussed before, the unit-triangular cases are handled by putting one as the diagonal elements and using the microkernels for non-unit-triangular cases. For the transpose and conjugate-transpose variants, we will reuse the previously discussed four variants. For LLT-variant, the operation is the same as the LLN-variant except that A is stored in a transposed storage. Since A is in transposed storage, instead of rows of A, we need to copy and pad columns of A. Recall that this is exactly what we did for the RLN-variant. As a result, to copy the data, we can reuse the triangular copy generator for RLN-variant. Next, keep in mind that A is a lower triangular but transposed matrix. If we could transpose the matrix A, it would become a no-transpose, upper triangular matrix. Because of this, to do the actual computations, we can reuse the *trmmμ* for the LUN-variant. So, the LLT-variant can be implemented using the copy from RLN-variant and the *trmmμ* for the LUN-variant. Like LLT-variant, LUT-variant can be implemented using the copy from RUN-variant and the *trmmμ* for the LLN-variant. To support all the conjugate variants (i.e. LLC, LUC, RLC, and RUC), a trivial sign change of the complex part of each element in the copy generators is enough (similarly trivial to $DIAG$ parameter changes). Table 2.2 summarizes which copy microkernels and *trmmμ* can be reused for all the transpose and conjugate-transpose variants.

Table 2.2: Summary for supporting transpose and conjugate-transpose variants of *trmmμ* through reuse of no-transpose variants

trmmμ Variants	Copy Microkernel to Use	*trmmμ* to Use
LLT	RLN	LUN
LUT	RUN	LLN
RLT	LLN	RUN
RUT	LUN	RLN
LLC	RLC	LUN
LUC	RUC	LLN
RLC	LLC	RUN
RUC	LUC	RLN

Note that the copy microkernels RLC, RUC, LLC, and LUC (mentioned in the table) are the trivially modified versions of RLN, RUN, LLN, and LUN copy microkernels, respectively.

2.3 *gemmμ*-based Triangular Solve of Matrix (TRSM)

TRiangular Solve for Matrix (TRSM) is a widely-used Level-3 BLAS operation. It is used to solve a triangular system of equations. It is also used by LAPACK operations like matrix inversions, LU and Cholesky factorizations. As described by the BLAS documentation [11], TRSM solves one of the matrix equations below for X:

$$op(A) \times X = alpha \times B, \quad \text{for } (SIDE = \text{Left})$$

or

$$X \times op(A) = alpha \times B, \quad \text{for } (SIDE = \text{Right})$$

Where *alpha* is a scalar, X and B are $(M \times N)$-sized general matrices and A is a unit or non-unit, lower or upper triangular matrix. The size of A is $M \times M$ for $SIDE = Left$ or or $N \times N$ for $SIDE = Right$ and $op(A)$ is one of the following: $op(A) = A$ or $op(A) = A^T$ or $op(A) = A^H$ (only for complex data types), where A^T is the transpose of A and A^H is the conjugate-transpose of A. The matrix X and B share the storage space, storing B on entry and X on exit of the routine. B is often referred to as the right-hand-side matrix.

All these variations of TRSM operation are determined by the following four parameters to the routine (description from BLAS documentation):

1. $SIDE$: Specifies whether op(A) is on the left or right side of X.

   If $SIDE = L$, then the equation is: $op(A) \times X = alpha \times B$.

   If $SIDE = R$, then the equation is: $X \times op(A) = alpha \times B$.

2. $UPLO$: Specifies whether the matrix A is an upper or lower triangular.

   If $UPLO = U$, then A is an upper triangular matrix.

   If $UPLO = L$, then A is a lower triangular matrix.

3. $TRANSA$: Specifies the form of $op(A)$ to be used in the solve.

   If $TRANSA = N$, then $op(A) = A$.

   If $TRANSA = T$, then $op(A) = A^T$.

   If $TRANSA = C$, then $op(A) = A^H$ (only for complex datatype).

4. $DIAG$: Specifies whether the diagonal of A is unit or not.

   If $DIAG = U$, then A is assumed to be unit triangular (diagonal elements are assumed to be unit).

   If $DIAG = N$, then A is not unit triangular (diagonal elements must be read during computation).

Like TRMM, there are 16 variants for each real precision and 24 variants for each complex precision. We will refer to each of these variants of TRSM using the values of these parameters. So, the LUTN variant of TRSM will denote $op(A)$ is on Left side of X, A is Upper triangular, Transposed ($op(A) = A^T$), and Non-unit-triangular.

TRSM can be implemented using either forward or backward substitution method for lower or upper triangular A, respectively. This is exactly what the original BLAS implementation does which is often called the *reference* implementation. Consider the example in Equation 2.1 for LLNN-variant of TRSM where $alpha = 1$, $M = 3$, $N = 1$:

$$\begin{bmatrix} A_{11} & 0 & 0 \\ A_{21} & A_{22} & 0 \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} \tag{2.1}$$

We can use the forward substitution method to solve for X for the problem shown in Equation 2.1. Since the storage for B is reused to store the result X, we will use Z to denote this common storage in our discussions. The computation consists of the steps shown in Figure 2.24. If all above operands (i.e. A_{11}, Z_1, etc.) are blocks rather than elements

1. $Z_1 = Z_1/A_{11}$ ..... // Solve for X_1
2. $Z_2 = Z_2 - A_{21} \times Z_1$ ..... // Subtract contribution of X_1
3. $Z_2 = Z_2/A_{22}$ ..... // Solve for X_2
4. $Z_3 = Z_3 - A_{31} \times Z_1 - A_{32} \times Z_2$ ..... // Subtract contribution of X_1 and X_2
5. $Z_3 = Z_3/A_{33}$ ..... // Solve for X_3

Figure 2.24: Computational steps for forward substitution method for LLNN-variant of TRSM

(cache-blocking optimization), then the above operations still work[2]. This statically blocked GEMM-based TRSM is substantially the same as the original Superscalar BLAS, as discussed in Section 2.1. Then the computation consists of the steps shown in Figure 2.25. One idea for the divisions of blocks in steps 1, 3, and 5 is to invert A blocks and multiply by the corresponding Z block, but this approach introduces high backward error [60]. Instead, notice that these steps are TRSM operations themselves but on block sized inputs. With this approach, optimized BLAS libraries like Superscalar BLAS or ATLAS can achieve high performance while maintaining low backward error. In our discussion, we will still use the mathematical notation of inverting the blocks as short-hand notation of block sized TRSM with the proviso that the inverse is not explicitly formed. As before, to overcome the draw-

[2]As long as the steps 1, 3, and 5 are done via TRSM, not matrix inversion.

1. $Z_1 = A_{11}^{-1} \times Z_1$ ..... // Solve for X_1 (TRSM)
2. $Z_2 = Z_2 - A_{21} \times Z_1$ ..... // Subtract contribution of X_1 (GEMM)
   $\Rightarrow$ may internally copy A_{21} and Z_1
3. $Z_2 = A_{22}^{-1} \times Z_2$ ..... // Solve for X_2 (TRSM)
4. $Z_3 = Z_3 - A_{31} \times Z_1 - A_{32} \times Z_2$ .... // Subtract contribution of X_1 and X_2 (GEMM)
   $\Rightarrow$ may internally copy A_{31}, A_{32}, Z_1 and Z_2
5. $Z_3 = A_{33}^{-1} \times Z_3$ ..... // Solve for X_3 (TRSM)

Figure 2.25: Computational steps for LLNN-variant of TRSM of Superscalar BLAS

back of statically blocked Superscalar BLAS or even the recursively blocked ATLAS, we will use the *gemm*μ framework to implement our optimized *gemm*μ-based TRSM. With our *gemm*μ-based TRSM, the computational steps for the problem given in Equation 2.1 is shown in Figure **??**. Note that in step 4, we are reusing the copy of Z_1 from step 2. For problems with multiple column panels of Z, we will be able to reuse all the copied A blocks for each additional column panel.

In steps 1, 3, and 5, we need a routine to perform the block sized TRSM. Unlike TRMM, we cannot zero-pad the triangular blocks. Our idea is to use *gemm*μ in a similar way to the approach outlined in Figure **??** to perform the block-sized TRSM which we will refer to as TRSM microkernels (*trsm*μ). We will discuss the implementation details of all *trsm*μ-variants in Section 2.3.1. In Section 2.3.2, we use our high performance *trsm*μ to build *gemm*μ-based TRSM for all variants.

2.3.1 Implementing *trsm*μ Using *gemm*μ

This section describes how to build *trsm*μ using our existing *gemm*μ framework. One obvious way to do that would be to invert the diagonal blocks, and then essentially do a *trmm*μ with the inverted matrix. However, inverting triangular matrices of more than one element introduces the condition number of the inverted matrix into the backwards error [60]. This means that this increased instability is observed in most cases, and so this method cannot generally be used.

We will instead rely on a second level of static blocking to produce a *gemm*μ-based *trsm*μ. Section 2.3.1.1 will explain the basic idea using the Left, Lower, No-transpose, Non-unit diagonal case of TRSM, and Section 2.3.1.2 will describe how we handle the TRSMs from this second level of blocking. Section 2.3.1.3 will then overview important information for all variants, while Sections 2.3.1.5-2.3.1.8 discuss details of the *trsm*μ variants. Finally, Section 2.3.1.9 discusses how the remaining cases are handled through reflection during the copy step.

1. $Z_1 = A_{11}^{-1} \times Z_1$ ..... // Solve for X_1 (TRSM)
2. $Z_2 = Z_2 - A_{21} \times Z_1$ ..... // Subtract contribution of X_1 (GEMM[†])
   (a) copy A_{21} to *gemm*μ's required storage format
   (b) copy Z_1 to *gemm*μ's required storage format
   (c) one call to *gemm*μ on the copied storage
   (d) subtract the result from Z_2 during copy-back
3. $Z_2 = A_{22}^{-1} \times Z_2$ ..... // Solve for X_2 (TRSM)
4. $Z_3 = Z_3 - A_{31} \times Z_1 - A_{32} \times Z_2$ .... // Subtract contribution of X_1 and X_2 (GEMM)
   (a) copy A_{31} and A_{32} to *gemm*μ's required storage format
   (b) copy Z_2 to *gemm*μ's required storage format
   (c) two calls to *gemm*μ on the copied storage
   (d) subtract the result from Z_3 during copy-back
5. $Z_3 = A_{33}^{-1} \times Z_3$ ..... // Solve for X_3 (TRSM)

Figure 2.26: Computational steps for *gemm*μ-based LLNN-variant of TRSM for one column panel of Z

[†]In actual implementation, like *gemm*μ-based TRMM, the copies and the *gemm*μ updates are intermixed for better performance.

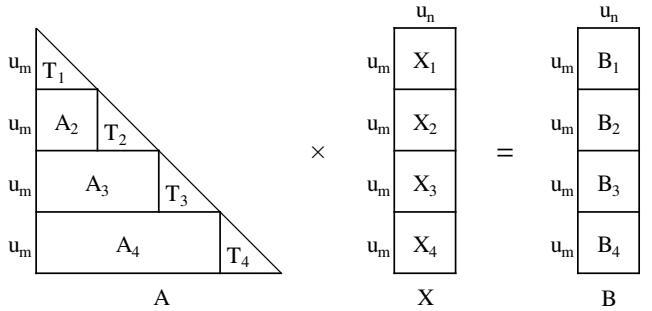
2.3.1.1 Overview of Implementing *trsmμ* Using *gemmμ* with LLN Example Case

Our main goal is to leverage the existing *gemmμ* so that our *trsmμ* does not require a whole new tuning framework. One simple way to achieve this is to simply do another step of GEMM-based blocking, as shown in Figure 2.27a.

Note that in Figure 2.27a, the total size of the A matrix is bounded by B_M, and that the blocks are bounded by the unrolling factor! Note that this means our *gemmμ* is invoked with $M = u_m, N = u_n, K = i \times u_m$, with $(1 \leq i < \frac{B_M}{u_m})$ for this LLN TRSM case. Note that only the K loop will iterate more than once in this scheme, and it is therefore likely this approach will achieve best performance for large block sizes where $i < \frac{B_M}{u_m}$ is large enough to amortize the cost of calling *gemmμ* and accessing the $u_m \times u_m$ portion of C. Building the *trsmμ* will therefore require us to specially tune a *gemmμ* to have a run-time variable K loop, and to force $u_m = i \times u_k, i \geq 1$.

The basic steps for this picture are shown in Figure 2.27b. The first thing to note is that if $N > u_n$, we can just iterate over all $\left\lceil \frac{N}{u_n} \right\rceil$ u_n-wide column panels, each done as shown above, to get an algorithm that can support any $M \leq B_M$ and any N.

The first question is: how do we perform steps 1, 3, 5, and 7, of Figure 2.27b? These steps all require a TRSM with $M = u_m, N = u_n$, which we will handle by generating a TRSM *nanokernel*, as discussed in Section 2.3.1.2. The second question is: how can we perform steps 2, 4, and 6, given that they clearly require a GEMM call with $\beta = 1, alpha = -1$,



(a)

1. $X_1 = T_1^{-1} B_1$
2. $Z_2 = B_2 - A_2 X_1$
3. $X_2 = T_2^{-1} Z_2$
4. $Z_3 = B_3 - A_3 \times X_{1:2}$
5. $X_3 = T_3^{-1} Z_3$
6. $Z_4 = B_4 - A_4 \times X_{1:3}$
7. $X_4 = T_4^{-1} Z_4$

(b)

Figure 2.27: Example unroll-blocked *gemmμ*-based *trsmμ* (a) picture and (b) steps

while *gemm*μ always does $\alpha = 1$? In our current code, the blocks of A shown in this figure are scaled by α during the copy step.

2.3.1.2 The TRSM Nanokernel

Since we cannot invert the diagonal blocks, we still need some method to do the solves for the T blocks of Figure 2.27a. Note that both u_m and u_n are compile-time constants discovered during the *trsm*μ's *gemm*μ tuning step, and so we could actually generate straightline code fully solving the problem without branches. Currently, we fully unroll the loop over u_m so that the triangle is handled without iteration, but keep the right-hand sides (known to be u_n) as a runtime loop. This allows the nanokernel to be used safely for cleanup code (when $(N \bmod u_n) > 0$), and minimizes code size. The main thing that distinguishes what we are calling a *nanokernel* from a *microkernel* is the amount of work it expects to perform, and thus the overhead it can amortize. The nanokernel is declared `inline`, and has only 1 loop for a two dimensional operation. We can't afford much overhead at this level, as we are doing only $O(1)$ operations. The values of u_m and u_n will vary by architecture, but on present machines $1 \leq u_m \leq 24$ and $1 \leq u_n \leq 4$ are typical ranges.

2.3.1.3 Handling Unit/Non-Unit Diagonal for All Variants

In this section, we describe the implementation details of four *trsm*μ that can support all variants of TRSM. Recall that the DIAG parameter specifies whether A is unit or non-unit triangular matrix. In our implementation, we will treat the unit diagonal as a non-unit diagonal and perform the extra division by one[3]. Note that by doing this extra computation, we halve the number of variants to implement. Due to this trivial implementation differences between these variants (e.g. LLNN vs. LLNU), we will ignore this parameter in our discussion of all variants of *trsm*μ.

[3]In actual implementation, the division of the diagonal is done once and the inverted diagonal is stored to avoid repeating the expensive division operation.

2.3.1.4 Increasing *gemm*μ Exploitation via Transposition

From Section 2.3.1.1 it is clear that *trsm*μ's performance will be a combination of the performance of *gemm*μ and the TRSM nanokernel. On almost all systems *gemm*μ's efficiency will be much greater than that of the nanokernel. The nanokernel has low overhead, but the level of optimization it achieves will be dictated by the native compiler (it will not even be vectorized unless the compiler autovectorizes it).

The proportion of TRSM vs. GEMM is set by the block factor of the diagonal blocks: as it increases, the percentage of TRSM goes up. To see how this can be problematic, consider ATLAS's best-case microkernel for a Intel Sandy Bridge architecture, which uses $u_m = 12$ and $u_n = 3$ If we assume $B_M = 36$, this results in the A matrix shown in Figure 2.28a. Using this partitioning results in *gemm*μ doing only around 66% of the computation for this case.

Now, imagine instead we could block A by $u_n = 3$ instead, as in Figure 2.28b. In this case *gemm*μ would perform more than 90% of the computation, and almost certainly result in a much higher performing *trsm*μ regardless of native compiler used. Fortunately, this can be accomplished in a fairly straightforward manner using matrix transposition.

Assume we are doing step 6 from Figure 2.27b, and that we are putting the results of the *gemm*μ call into the $u_m \times u_n$ matrix, C which we will then subtract from Z_4 later. Therefore, the *gemm*μ call would compute: $C = A_4 \times X_{1:3}$. Mathematically, its perfectly legal to transpose both sides of this equation, leading to $C^T = X_{1:3}^T \times A_4^T$. We are now computing an $u_n \times u_m$ product, with the u_m and u_n blockings getting transposed as well, leading to the picture shown in Figure 2.28b. Note that the result we are producing is of fixed size, in this example either 12×3 or 3×12. We therefore generate a copy macro that can perform a copy without and without transposition (and scaling) from *gemm*μ C storage back into standard format. Using this, we can utilize this transposition technique anytime the unroll factor that is naturally first (as determined by the TRSM variant) is significantly larger than the unroll factor along the other dimension.

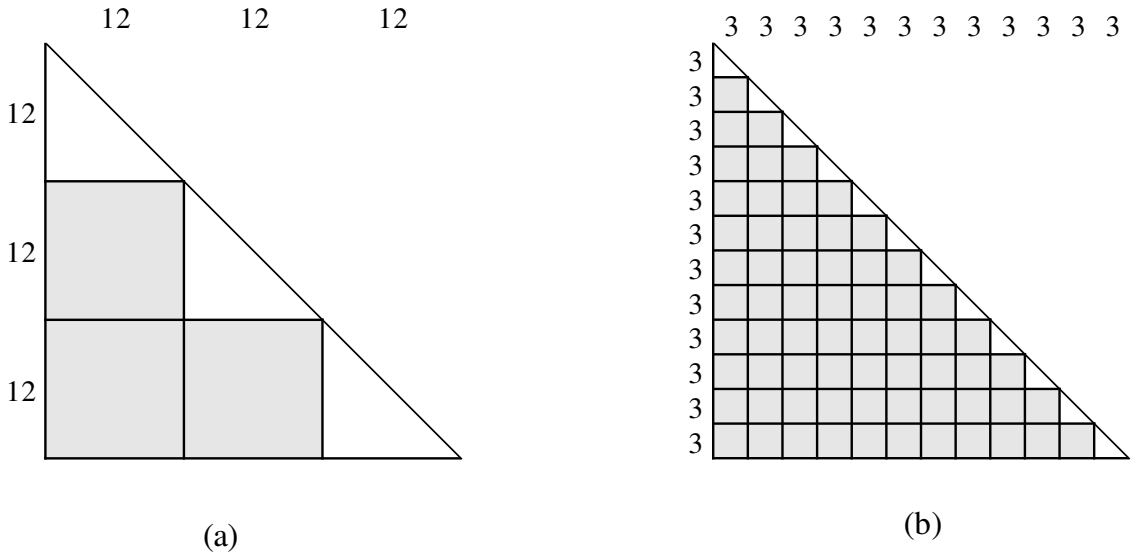


Figure 2.28: Partitioning of the triangular matrix for LLN-variant depicting the data used by *gemm*μ updates (gray blocks): (a) $u_m = 12$ and (b) $u_m = 3$.

2.3.1.5 *trsm*μ for LLN-variant

LLN denotes the following operation: $X = alpha \times op(A)^{-1} \times B$, where A is the lower triangular and B is a general matrix. Since we want to utilize ATLAS's *gemm*μ framework, we need to select a *gemm*μ that we can use in our *trsm*μ implementation. The selected *gemm*μ's unroll factors u_m, u_n and u_k in the M, N, and K dimensions (respectively) will dictate the blocking of the input matrices of our *trsm*μ. Since A is triangular and it appears on the left side of X in this case, we have $B_M = B_K$.

The idea is to partition the triangular A and the general matrix B into u_m-sized row panels and u_n-sized column panels, respectively. For the computation, we apply a *gemm*μ-based approach similar to the one shown in Figure 2.27. To better understand the implementation, consider the example shown in Figure 2.29. For this example, we assume alpha=1 to simplify the discussion. The triangular A is divided into u_m-sized row-panels and the right-hand-side matrix Z is divided into $u_m \times u_n$-sized blocks. Recall that Z initially stores the input B which is transformed into X by the solve.

Each unroll-block of A is of size $u_m \times u_m$ and each unroll-block of Z (B or X) is of size $u_m \times u_n$. Since we work on one u_n-sized column-panel of Z at a time, we can start by solving

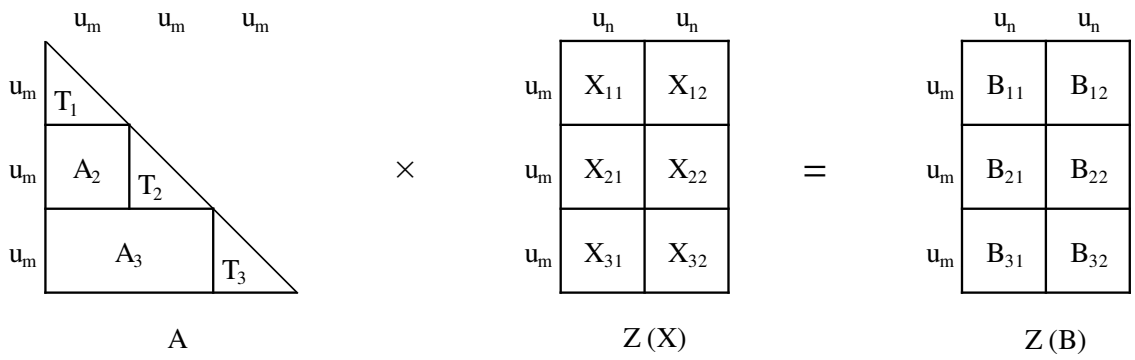


Figure 2.29: An example for LLN-variant of *trsmμ*

the first column panel of $Z(B)$ (i.e. $[\ Z_{11}\ Z_{21}\ Z_{31}\ ]^T$). In the first step, we copy the block T_1 with the diagonal elements inverted and then solve Z_{11} using the copied T_1 which is a TRSM of size $u_m \times u_n$. For the next block of Z (i.e. Z_{21}), we need to copy A_2 and previously solved Z_{11} to *gemmμ*'s A and B storage format, respectively. These copies are done using the copy routines provided by the framework specific to our selected u_m, u_n and u_k combination. After the copy is done, we can call the *gemmμ* once to use the copied A_2 and Z_{11} and subtract the result from Z_{21}. At this point, Z_{21} is ready for the solve using T_2. Again, we first copy T_2 in this step and then do the $(u_m \times u_n)$-sized TRSM. For the next block Z_{31}, we copy the next row panel of A omitting the diagonal unroll-block (i.e. A_3) and the previously solved Z_{21}. Then we call the *gemmμ* once to multiply A_3 and $[\ Z_{11}\ Z_{21}\ ]^T$ and subtract it from Z_{31}. Note that we can reuse the copy of Z_{11} that was done to solve Z_{21}. At this point, Z_{31} is ready to be solved using T_3. Once we are done with the first column-panel of Z, we move to the next column-panel and repeat the same steps. Note that the copy steps of A are needed only for the first column-panel of Z. For the rest of the panels of Z, we will reuse these copies of A. As we can see, that each block is copied exactly once, and reused for as long as possible: reuse of Z blocks of a column-panel ends at the end of the solve for that panel but reuse of A blocks ends at the end of the solve for the entire input matrix. The basic implementation of the LLN-variant of *trsmμ* is shown in Figure 2.30.

- **foreach** u_n-sized column-panel $Zcpan$ in Z
  1. if first Z-column-panel, copy the first $(u_m \times u_m)$-sized diagonal unroll-block of A while inverting the diagonal elements
  2. solve the top unroll-block $Zblk0$ of $Zcpan$ with the copied triangular unroll-block of A (nanokernel)
  3. **foreach** unroll-block $Zblk$ below $Zblk0$ of $Zcpan$ from top to bottom
     $\rightarrow$ requires access of u_m-sized row-panels of A from the top to the bottom
     (a) copy the last solved unroll-block of $Zcpan$ to $gemm\mu$'s B storage format
     (b) if first Z-column-panel, copy the u_m-sized row-panel $Arpan$ omitting diagonal unroll-block $Adiag\_blk$ to $gemm\mu$'s A storage format
     (c) one call to $gemm\mu$ to multiply copied $Arpan$ to copied part of $Zcpan$
     (d) subtract the $gemm\mu$ result from $Zblk$ during copy-back
     (e) solve the $Zblk$ using the copied $Adiag\_blk$ (nanokernel)

Figure 2.30: Basic computational steps for the LLN-variant of $trsm\mu$

2.3.1.6 $trsm\mu$ for LUN-variant

LUN denotes the following operation: $X = alpha \times op(A)^{-1} \times B$, where A is an upper triangular matrix and B is a general matrix. We can select the best $gemm\mu$ thus the u_m, u_n and u_k combination using a search similar to the search for LLN-variant and with the same restriction that u_m is a multiple of u_k (i.e. $u_m = i \times u_k$, where $i \geq 1$). For the microkernel implementation, unlike LLN-variant, we need to start from the last triangle and move upward to solve for LUN-variant.

To understand the implementation, consider the example in Figure 2.31. Again, we assume alpha=1 to simplify the discussion. Each block of A is of size $u_m \times u_m$ and each sub block of Z (B or X) is of size $u_m \times u_n$. Since we work on one u_n-sized column-panel of Z at a time, we can start by solving the first column panel of $Z(B)$ (i.e. $[Z_{11}\ Z_{21}\ Z_{31}]^T$). Since A is upper-triangular, the algorithm follows the order of a backward-substitution method unlike LLN-variant. In the first step, we copy the block T_3 and then solve Z_{31} using the copied T_3

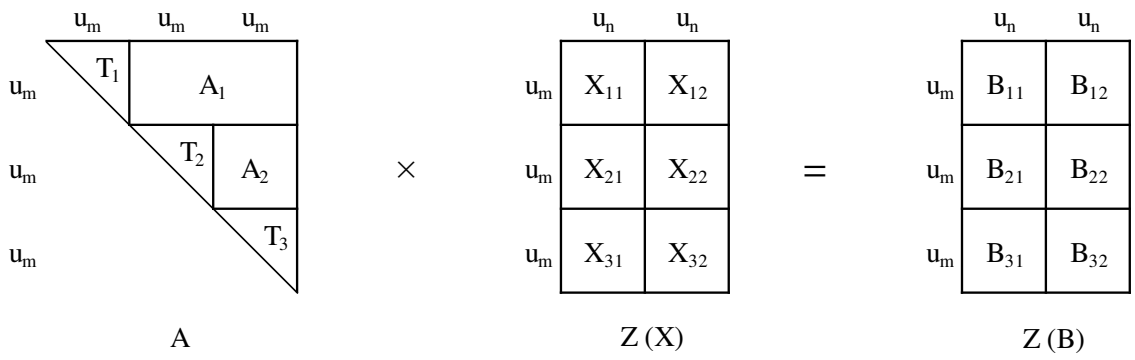


Figure 2.31: An example for LUN-variant of *trsmμ*

which is a TRSM for size $u_m \times u_n$ (nanokernel). For the next block of Z (i.e. Z_{21}), we need to copy A_2 and previously solved Z_{31} to *gemmμ* required storage format. These copies are done using the copy routines provided by the framework specific to our selected u_m, u_n and u_k combination. After the copy is done, we can call the *gemmμ* to use the copied A_2 and Z_{31} and subtract the result from Z_{21}. At this point, Z_{21} is ready for the solve using T_2. Again, we copy T_2 in this step and then do the $(u_m \times u_n)$-sized TRSM. For the next block Z_{11}, we copy the next row panel of A except the diagonal block (i.e. A_1) and the previously solved Z_{21}. Then we call the *gemmμ* to multiply A_1 to $[Z_{21}\ Z_{31}]^T$ and subtract it from Z_{11}. Note that we can reuse the copy of Z_{31} that was done to solve Z_{21}. At this point, Z_{11} is ready to be solved using T_1. Note that like the LLN-variant, the copies of A are needed only for the first column-panel of Z. For the rest of the panels of Z, we will reuse these copies of A. As we can see, that each block is copied exactly once, and reused for as long as possible: reuse of Z blocks of a column-panel ends at the end of the solve for that panel but reuse of A blocks ends at the end of the solve for the entire input matrix. The basic implementation for LUN-variant is shown in Figure 2.32.

2.3.1.7 *trsmμ* for RLN-variant

RLN denotes the following operation: $X = alpha \times B \times op(A)^{-1}$, where A is a lower triangular matrix and B is a general matrix. Like the LLN- and LUN-variants, to utilize the *gemmμ* framework, we need to select the best *gemmμ* that we can use in our *trsmμ* implementation.

- **foreach** u_n-sized column-panel $Zcpan$ in Z

    1. if first Z-column-panel, copy the bottom $(u_m \times u_m)$-sized diagonal unroll-block of A while inverting the diagonal elements
    2. solve the bottom unroll-block $Zblk0$ of $Zcpan$ with the copied triangular unroll-block of A (nanokernel)
    3. **foreach** unroll-block $Zblk$ above $Zblk0$ of $Zcpan$ from bottom to top
       $\rightarrow$ requires access of u_m-sized row-panels of A from the bottom to the top
        (a) copy the last solved unroll-block of $Zcpan$ to $gemm\mu$'s B storage format
        (b) if first Z-column-panel, copy the u_m-sized row-panel $Arpan$ omitting diagonal unroll-block $Adiag\_blk$ to $gemm\mu$'s A storage format
        (c) one call to $gemm\mu$ to multiply copied $Arpan$ to copied part of $Zcpan$
        (d) subtract the $gemm\mu$ result from $Zblk$ during copy-back
        (e) solve the $Zblk$ using the copied $Adiag\_blk$ (nanokernel)

Figure 2.32: Basic computational steps for the LUN-variant of $trsm\mu$

The selected *gemmμ*'s unroll factors u_m, u_n and u_k in the M, N and K dimensions (respectively) will dictate the blocking of the input matrices of our *trsmμ*. Since A is triangle and it is on the right side of X (i.e. A is treated as *gemmμ*'s B) in this case, we have $B_N = B_K$. Therefore, we impose a restriction to the search for the best *gemmμ* for our TRSM that u_n must be a multiple of u_k (i.e. $u_n = i \times u_k$, where $i \geq 1$).

The idea is to partition the triangular matrix A and the general matrix B into u_n-sized column panels and u_m-sized row panels, respectively. Unlike the left-variants (e.g. LLN, LUN, etc.), one row-panel (instead of column-panel) of B is solved at a time for the right-variants. As discussed in Section 2.2.1, accessing a row panel at a time allows to reuse the workspace for only one row panel for $Z(B)$ (instead of having a workspace for the entire matrix). Within each row-panel, for RLN, we start from the last block and move from right to left direction. To understand the implementation details, consider the example shown in Figure 2.33. Again, we assume $alpha = 1$ to simplify the discussion. Each block of A is of size $u_n \times u_n$ and each block of Z (B or X) is of size $u_m \times u_n$. Since we work on one u_m-sized row-panel of Z at a time, we start by solving the first row-panel of Z (i.e. $[Z_{11}\ Z_{12}\ Z_{13}]$). In the first step, we copy the block T_3 and then solve Z_{13} using the copied T_3 which is a TRSM of size $u_m \times u_n$ (nanokernel). For the next block of Z (i.e. Z_{12}), we copy A_2 and previously solved Z_{13} to *gemmμ*'s required storage format. These copies are done using the copy routines provided by the framework specific to our selected u_m, u_n and u_k combination. After the copy is done, we can call the *gemmμ* to multiply the copied A_2 and Z_{13} and then

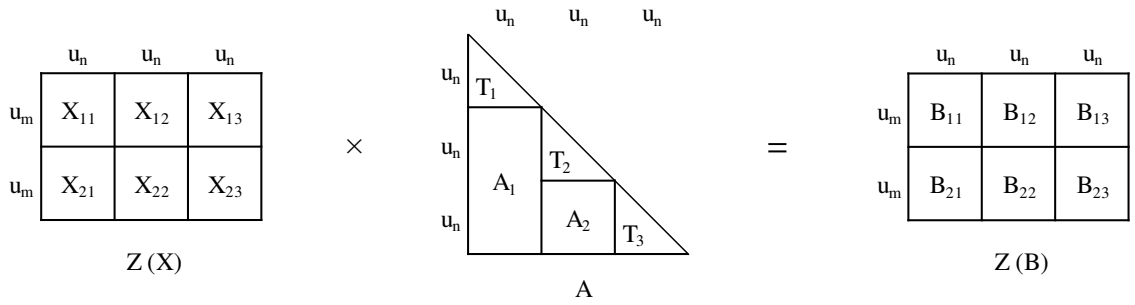


Figure 2.33: An example for RLN-variant of *trsmμ*

subtract the result from Z_{12}. At this point, Z_{12} is ready for the solve using T_2. Again, we start by copying T_2 and then applying $(u_m \times u_n)$-sized TRSM on Z_{12}. For the next block Z_{11}, we copy the next column panel of A except the diagonal block (i.e. A_1) and the previously solved Z_{12}. Then we call the *gemmμ* to multiply $[Z_{12}\ Z_{13}]$ to A_1 and then subtract the result from Z_{11}. Note that just like the left-variants, we reuse our previously copied Z_{13}. At this point, Z_{11} is ready to be solved using T_1. Like the left-variants, we only need to copy the blocks of A when solving the first row-panel of Z. For the other panels, we will reuse the copied A blocks. So, each block is copied exactly once and reused for as long as possible. The basic implementation of the RLN-variant is shown in Figure 2.34.

Recall that the triangle blocks, in this case, are of size *nuxnu*. If $u_n > u_m$, to use the *gemmμ* to do most of the FLOPs needed, we can apply the same transposition technique discussed in Section 2.3.1.4.

2.3.1.8 *trsmμ* for RUN-variant

RUN denotes the operation: $X \times op(A) = alpha \times B$, where A is the upper triangular matrix and B is a general matrix. Like RLN-variant, we need to select the best *gemmμ* (with the unroll factors u_m, u_n and u_k) that we can use in our *trsmμ* implementation with the same restriction to the search that u_n must be a multiple of u_k (i.e. $u_n = i \times u_k$, where $i \geq 1$).

Like RLN-variant, the triangular matrix A and the general matrix B is partitioned into u_n-sized column panels and u_m-sized row panels, respectively. We will solve one row-panel of B at a time but unlike RLN, we start from the leftmost block of the row-panel and move from left to right direction. To understand the implementation in this case, consider the example shown in Figure 2.35. We assume $alpha = 1$ to simplify the discussion. Each block of A is of size $u_n \times u_n$ and each block of Z (B or X) is of size $u_m \times u_n$. Since we work on one u_m-sized row-panel of Z at a time, we start by solving the first row-panel of Z (i.e. $[Z_{11}\ Z_{12}\ Z_{13}]$). In the first step, we copy the block T_1 and then solve Z_{11} using the copied T_1 which is a TRSM of size $u_m \times u_n$. For the next block of Z-panel (i.e. Z_{12}), we copy A_2 and

- **foreach** u_n-sized row-panel $Zrpan$ in Z
  1. if first Z-row-panel, copy the rightmost $(u_n \times u_n)$-sized diagonal unroll-block of A while inverting the diagonal elements
  2. solve the rightmost unroll-block $Zblk0$ of $Zcpan$ with the copied triangular unroll-block of A (nanokernel)
  3. **foreach** unroll-block $Zblk$ on left of $Zblk0$ of $Zcpan$ from right to left
     $\rightarrow$ requires access of u_n-sized column-panels of A from right to left
     (a) copy the last solved unroll-block of $Zrpan$ to $gemm\mu$'s A storage format
     (b) if first Z-row-panel, copy the u_n-sized column-panel $Acpan$ omitting diagonal unroll-block $Adiag\_blk$ to $gemm\mu$'s B storage format
     (c) one call to $gemm\mu$ to multiply copied $Zrpan$ to copied part of $Acpan$
     (d) subtract the $gemm\mu$ result from $Zblk$ during copy-back
     (e) solve the $Zblk$ using the copied $Adiag\_blk$ (nanokernel)

Figure 2.34: Basic computational steps for the RLN-variant of $trsm\mu$

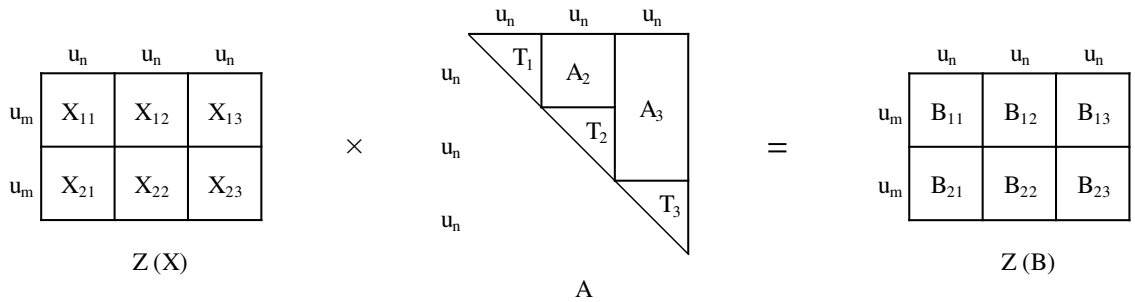


Figure 2.35: An example for RUN-variant of $trsm\mu$

the previously solved Z_{11} to *gemm*μ's required storage format. These copies are done using the copy routines provided by the framework for our selected u_m, u_n and u_k combination. After the copy is done, we can call the *gemm*μ to multiply the copied Z_{11} and A_2 and then subtract the result from Z_{12}. At this point, Z_{12} is ready for the solve using T_2. We copy the diagonal block T_2 and then apply $(u_m \times u_n)$-sized TRSM on Z_{12}. For the next block Z_{13}, we copy the next column-panel of A except the diagonal block (i.e. A_3) and the previously solved Z_{12}. Then we call the *gemm*μ to multiply $[Z_{11}\ Z_{12}]$ to A_3 and then subtract the result from Z_{13}. Note that just like other variants, we can reuse our previously copied Z_{11}. Next, we solve Z_{13} using T_3 and we are done with the current row-panel of Z. Then we move on to the next row panel of Z and repeat the same process. Like in other variants, we copy A only for the first row-panel of Z and then reuse the same copy for later panels. The basic implementation of RUN-variant is given in Figure 2.36.

2.3.1.9 Other *trsm*μ Variants Using Reflection

As discussed in Section 2.2.1.5, the other variants (i.e. LLT, LUT, RLT, etc.) of *trsm*μ can be implemented by using the above discussed 4 variants and selecting the proper copy microkernels. Therefore, Table 2.1 works for *trsm*μ as well. Note that even though we can reuse the variants LLN, LUN, RLN, and RUN to support all other variants of *trsm*μ, we still need 12 variants (ignoring $DIAG$ parameter) of the TRSM nanokernels that are slightly modified forward- and backward- substitution methods.

2.3.2 Our Statically Blocked Iterative TRSM

In this section, we will present our statically blocked iterative TRSM that also utilizes the *gemm*μ framework but on a block level instead of unroll blocks and for its block-sized TRSM on the diagonal blocks, it uses the *trsm*μ we developed in the Section 2.3.1. We only discuss the four variants: LLN, LUN, RLN, & RUN and all other variants can be reduced to one of these four variants like the TRMM variants as discussed in Section 2.2.1.5.

- **foreach** u_n-sized row-panel $Zrpan$ in Z
  1. if first Z-row-panel, copy the leftmost $(u_n \times u_n)$-sized diagonal unroll-block of A while inverting the diagonal elements
  2. solve the leftmost unroll-block $Zblk0$ of $Zcpan$ with the copied triangular unroll-block of A (nanokernel)
  3. **foreach** unroll-block $Zblk$ on right of $Zblk0$ of $Zcpan$ from left to right
     $\rightarrow$ requires access of u_n-sized column-panels of A from left to right
     (a) copy the last solved unroll-block of $Zrpan$ to *gemm*μ's A storage format
     (b) if first Z-row-panel, copy the u_n-sized column-panel $Acpan$ omitting diagonal unroll-block $Adiag\_blk$ to *gemm*μ's B storage format
     (c) one call to *gemm*μ to multiply copied $Zrpan$ to copied part of $Acpan$
     (d) subtract the *gemm*μ result from $Zblk$ during copy-back
     (e) solve the $Zblk$ using the copied $Adiag\_blk$ (nanokernel)

Figure 2.36: Basic computational steps for the RUN-variant of *trsm*μ

2.3.2.1 Implementing LLN-variant of *gemm*μ-based TRSM

The idea is almost identical to the microkernel implementation for LLN-variant. Here, We need to find a *gemm*μ that works with block factors B_M, B_N and B_K (divides the dimensions M, N, and K respectively) where we impose the restriction on the search so that $B_M = B_K$ instead of B_M to be a multiple of B_K like we did for the *trsm*μ. This significantly simplifies the implementation while providing very good performance for most problems. With the selected block factors, the matrix A is divided into $(B_M \times B_M)$-sized blocks and the matrix B is divided into $(B_M \times B_N)$-sized blocks. Like the microkernel implementation, here we solve one B_N-sized column panel of B at a time and for each of these panels, we solve one $B_M \times B_N$ block at a time using the selected *gemm*μ and then calling our *trsm*μ (implemented in Section 2.3.1) for LLN-variant. The basic implementation is shown in Figure 2.37. Recall that we use Z to denote the storage holding B and the result X. The key difference, compared to the implementation in Figure 2.30, is that here we are performing the operations on blocks

instead of unroll blocks of data and the GEMM updates need to be done by calling the *gemm*μ once for each block update (shown in step 3(c)).

2.3.2.2 Implementing LUN-variant of *gemm*μ-based TRSM

Like the LLN-variant, we need to find a *gemm*μ that works with block factors B_M, B_N and B_K where $B_M = B_K$. The matrix is similarly divided as the LLN-variant. However, as described in the LUN *trsm*μ implementation, we start from the last block of each column-panel of B and then move upward on the column panel solving one block at a time. The basic implementation is shown in Figure 2.38.

2.3.2.3 Implementing RLN-variant of *gemm*μ-based TRSM

As we have seen in the microkernel implementation for RLN-variant, in this case, the triangle A appears on the right side of B (X). So, for this case, we need a *gemm*μ where $B_N = B_K$. Then we divide the matrix A into $B_N \times B_N$ blocks and the matrix B into $B_M \times B_N$ blocks. Also, we solve one B_M-sized row-panel of B at a time. In RLN-variant, for each panel, we start by solving the last block and move from right to left direction solving one block at a time. The basic implementation is shown in Figure 2.39. Again, the key difference compared to the microkernel implementation is that here the GEMM update step requires multiple calls to the *gemm*μ (step 3(c)).

2.3.2.4 Implementing RUN-variant of *gemm*μ-based TRSM

Similar to the RLN-variant, we need a *gemm*μ where $B_N = B_K$. Then we divide the matrix A into $B_N \times B_N$ blocks and the matrix B into $B_M \times B_N$ blocks and we solve one B_M-sized row-panel of B at a time. Unlike RLN-variant, for each row-panel of B, we start by solving the first block and then move from left to right direction solving one block at a time. The implementation is shown in Figure 2.40.

- **foreach** column-panel $Zcpan$ in Z

  1. solve the top-most block $Zblk0$ of $Zcpan$ using the diagonal block of A ($trsm\mu$)
  2. **foreach** block $Zblk$ below $Zblk0$ of $Zcpan$ from top to bottom

     $\rightarrow$ requires access of B_M-sized row panels of A from the top to the bottom

     (a) copy the last solved block of $Zpan$ to $gemm\mu$'s B storage format

     (b) if first Z-panel, copy the B_M-sized row-panel $Arpan$ of A omitting the diagonal block $Adiag\_blk$ to $gemm\mu$'s A storage format

     (c) call $gemm\mu$ once for each block of copied $Arpan$ to multiply with the corresponding block of copied $Bcpan$ and accumulate the result

     (d) subtract the accumulated $gemm\mu$ result from $Zblk$

     (e) solve the $Zblk$ using the $Adiag\_blk$ ($trsm\mu$)

Figure 2.37: Basic computational steps for the LLN-variant of full TRSM

- **foreach** column-panel $Zcpan$ in Z

  1. solve the bottom block $Zblk0$ of $Zcpan$ using the bottom diagonal block of A ($trsm\mu$)
  2. **foreach** block $Zblk$ above $Zblk0$ of $Zcpan$ from bottom to top

     $\rightarrow$ requires access of B_M-sized row panels of A from the bottom to the top

     (a) copy the last solved block of $Zpan$ to $gemm\mu$'s B storage format

     (b) if first Z-panel, copy the B_M-sized row-panel $Arpan$ of A omitting the diagonal block $Adiag\_blk$ to $gemm\mu$'s A storage format

     (c) call $gemm\mu$ once for each block of copied $Arpan$ to multiply with the corresponding block of copied $Bcpan$ and accumulate the result

     (d) subtract the accumulated $gemm\mu$ result from $Zblk$

     (e) solve the $Zblk$ using the $Adiag\_blk$ ($trsm\mu$)

Figure 2.38: Basic computational steps for the LUN-variant of full TRSM

- **foreach** row-panel $Zrpan$ in Z

  1. solve the rightmost block $Zblk0$ of $Zcpan$ using the rightmost diagonal block of A ($trsm\mu$)
  2. **foreach** block $Zblk$ above $Zblk0$ of $Zcpan$ from right to left

     $\rightarrow$ requires access of B_N-sized column panels of A from right to left

     (a) copy the last solved block of $Zpan$ to $gemm\mu$'s A storage format

     (b) if first Z-panel, copy the B_N-sized column-panel $Acpan$ of A omitting the diagonal block $Adiag\_blk$ to $gemm\mu$'s B storage format

     (c) call $gemm\mu$ once for each block of copied $Brpan$ to multiply with the corresponding block of copied $Acpan$ and accumulate the result

     (d) subtract the accumulated $gemm\mu$ result from $Zblk$

     (e) solve the $Zblk$ using the $Adiag\_blk$ ($trsm\mu$)

Figure 2.39: Basic computational steps for the RLN-variant of full TRSM

- **foreach** row-panel $Zrpan$ in Z

  1. solve the leftmost block $Zblk0$ of $Zcpan$ using the leftmost diagonal block of A ($trsm\mu$)
  2. **foreach** block $Zblk$ above $Zblk0$ of $Zcpan$ from left to right

     $\rightarrow$ requires access of B_N-sized column panels of A from left to right

     (a) copy the last solved block of $Zpan$ to $gemm\mu$'s A storage format

     (b) if first Z-panel, copy the B_N-sized column-panel $Acpan$ of A omitting the diagonal block $Adiag\_blk$ to $gemm\mu$'s B storage format

     (c) call $gemm\mu$ once for each block of copied $Brpan$ to multiply with the corresponding block of copied $Acpan$ and accumulate the result

     (d) subtract the accumulated $gemm\mu$ result from $Zblk$

     (e) solve the $Zblk$ using the $Adiag\_blk$ ($trsm\mu$)

Figure 2.40: Basic computational steps for the RUN-variant of full TRSM

2.3.2.5 Other Variants of TRSM Using Reflection

Since our *gemmμ*-based TRSM is almost identical to our approach for *trsmμ*, we can use the same technique presented in Section 2.3.1.9 to support other variants of TRSM. Therefore, as with *trsmμ*, Table 2.1 also applies to our *gemmμ*-based TRSM.

2.4 Performance Results

This section shows some preliminary results that demonstrate the benefits of our new *gemmμ*-based BLAS approach. Note that we need to extend the ATLAS autotuning framework to ensure we utilize the best possible optimization set for our *trmmμ* and *trsmμ* in order to get final results. Unfortunately, ATLAS's tuning framework is currently being redesigned as part of other research efforts, and so we must await completion of that unrelated research prior to finalizing our performance measurements. Therefore, the results presented here should serve as a floor rather than as a ceiling for the performance of this approach.

For this preliminary performance measurement we chose *gemmμ* that are usually selected for ATLAS's GEMM and are compatible with the u_m, u_n, and u_k restrictions of our *gemmμ*-based TRMM and TRSM. As for the block factors B_M and B_N, we selected $B_M = B_N$ and used the best block factors that are selected by ATLAS's GEMM for all our timings presented in this section. We show the performance results for two commonly used test cases: a) square sized inputs ($M = N$) and b) fat inputs with a small triangle and a large number of right-hand sides[4] i.e. $N >> M$ for left variants and $M >> N$ right variants (common for right-looking LU, QR and Cholesky factorizations in LAPACK). For fat cases, we selected $M = 120$ for the left variants ($N = 120$ for the right variants) and vary the right-hand sides. Performance of different libraries can vary significantly depending on their own best block factors. We chose 120 for the size of the degenerate dimension of the fat problems to provide a fair comparison among the libraries as 120 is a multiple of most of the common

[4]Although the phrase "right-hand side" applies only to TRSM, we will use this to refer to the rectangular matrix for both TRSM and TRMM in this section.

register-unrolling factors on modern machines. Note that 120 is not the best block factor for either ATLAS or our *gemm*μ-based approach.

We timed our *gemm*μ-based routines on three different systems each having different architecture for the floating point computations:

1. **O32**: AMD Opteron 6128, 32 cores, CPU Speed: 2.0 GHz,

   Peak Performance / core: 8 GFLOPS

   gemmμ for TRSM used with $B_M = B_N = B_K = 168$

   trsmμ used with $u_m = 3, u_n = 6, u_k = 3$ for left-variants (with transpose technique)

   trsmμ used with $u_m = 6, u_n = 3, u_k = 3$ for right-variants

2. **X12**: Intel Xeon 2620 v2 (Ivy Bridge), 12 cores, CPU Speed: 2.1 GHz,

   Peak Performance / core: 16.8 GFLOPS

   gemmμ for TRSM used with $B_M = B_N = B_K = 192$

   trsmμ used with $u_m = 3, u_n = 12, u_k = 1$ for left-variants (with transpose technique)

   trsmμ used with $u_m = 12, u_n = 3, u_k = 1$ for right-variants

3. **X24**: Intel Xeon 2670 v3 (Haswell-EP), 24 cores, CPU Speed: 1.7 GHz,

   Peak Performance / core: 27.2 GFLOPS

   gemmμ for TRSM and TRMM used with $B_M = B_N = B_K = 192$

   trmmμ used with $u_m = 12, u_n = 4, u_k = 1$

   trsmμ used with $u_m = 4, u_n = 12, u_k = 1$ for left-variants (with transpose technique)

   trsmμ used with $u_m = 12, u_n = 4, u_k = 1$ for right-variants

For performance comparison with our approach, we used ATLAS's current recursively blocked approach using ATLAS 3.11.39. We also used Intel's proprietary Math Kernel Library (MKL 11.2.3) [43] which is considered one of the best linear algebra library for x86 hardware by most computational researchers. All performance results are shown as the percentage of the machine's theoretical peak performance assuming the minimal FLOP count of the

algorithm in question[5]. In all the charts, the performance of recursive ATLAS (ATL), MKL and our *gemm*μ-based approach (μK) are shown as green circle, red diamond and blue square, respectively.

Figure 2.41 and 2.42 shows the double precision TRSM performance for square and fat problems, respectively, for all four variants on O32. As shown in Figure 2.41, for square problems our *gemm*μ-based TRSM outperforms both stock ATLAS and MKL for all problem sizes and all variants by up to 15% and for fat problems, the improvement is up to 39% (shown in Figure 2.42).

Figure 2.43 and 2.44 shows the double precision TRSM performance for square and fat problems, respectively, for all four variants on X12. As shown in Figure 2.43, for square problems our *gemm*μ-based TRSM outperforms prior ATLAS for all problem sizes and all variants by up to 11%. When comparing to MKL, however, we lose for small problems, while usually winning very slightly for large. We believe these small-case losses (which occur on X24 as well) are due to the aforementioned lack of autotuning. This preliminary implementation is using a fixed $B = 192$ for any $M >= 192$, which is highly unlikely to be optimal for small problems. We therefore believe once the full tuning framework is in place, we will be competitive with MKL across the full range.

For fat cases on X12, as shown in Figure 2.44, our approach outperforms prior ATLAS by up to 52% but it is inferior to MKL for left variants and competitive for right variants. Notice that our approach gets roughly the same performance for both left and right variants, but MKL's left-case performance is around 15% higher than its right. It is probably not a coincidence that the left case is used in many more benchmarks in general, and the LINPACK benchmark in particular, while the right case is not. Our best guess is that a lot of

[5]It is therefore theoretically impossible for our algorithms to achieve peak with results reported in this way, since we do extra FLOPs due to zero padding in TRMM and reciprocating the diagonal elements in TRSM. Using the minimal FLOP as the base means we treat these extra computations as a performance loss against theoretical peak.

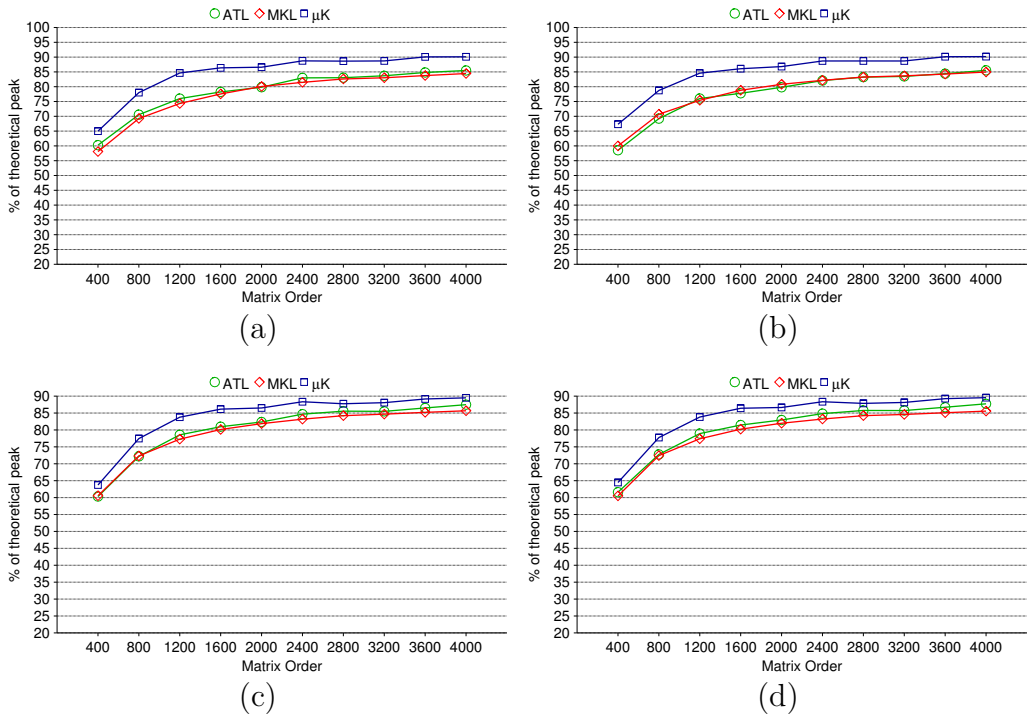


Figure 2.41: Performance of double-precision real TRSM on O32 for square problem sizes for variants: (a) LLNN (b) LUNN (c) RLNN (d) RUNN

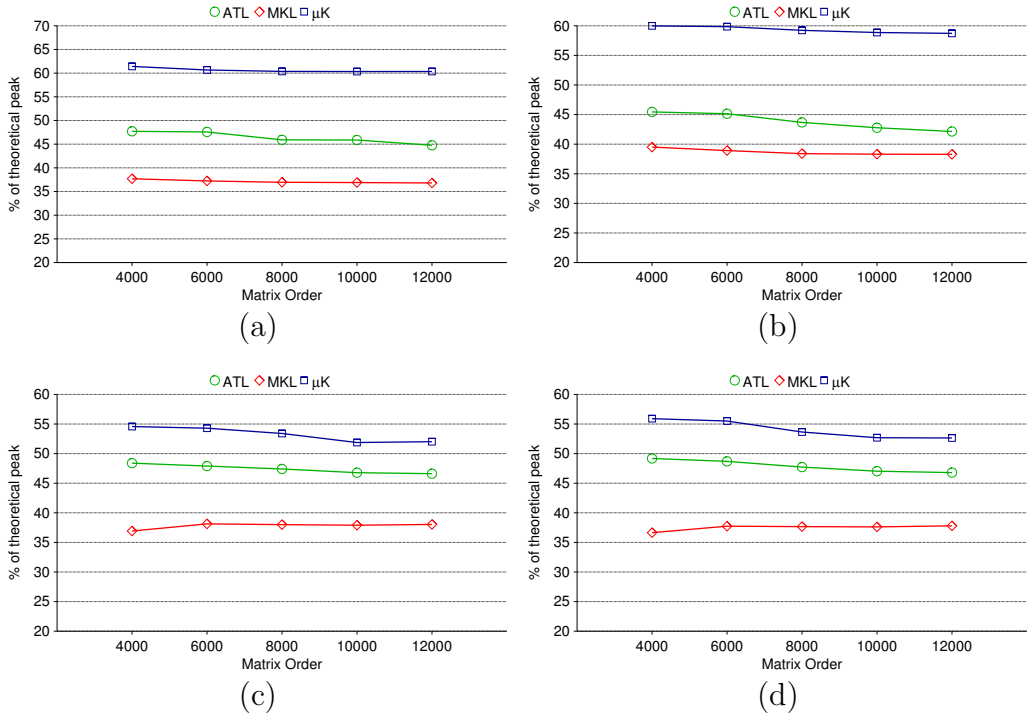


Figure 2.42: Performance of double-precision real TRSM on O32 for fat problems with constant triangle size of (120×120) for variants: (a) LLNN (b) LUNN (c) RLNN (d) RUNN

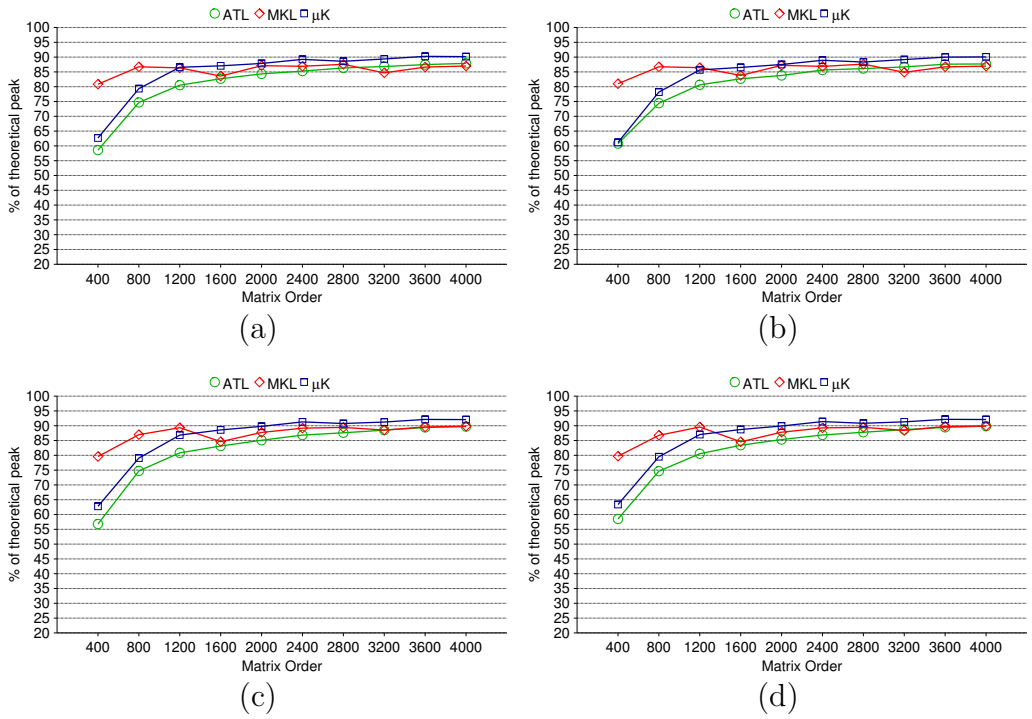


Figure 2.43: Performance of double-precision real TRSM on X12 for square problem sizes for variants: (a) LLNN (b) LUNN (c) RLNN (d) RUNN

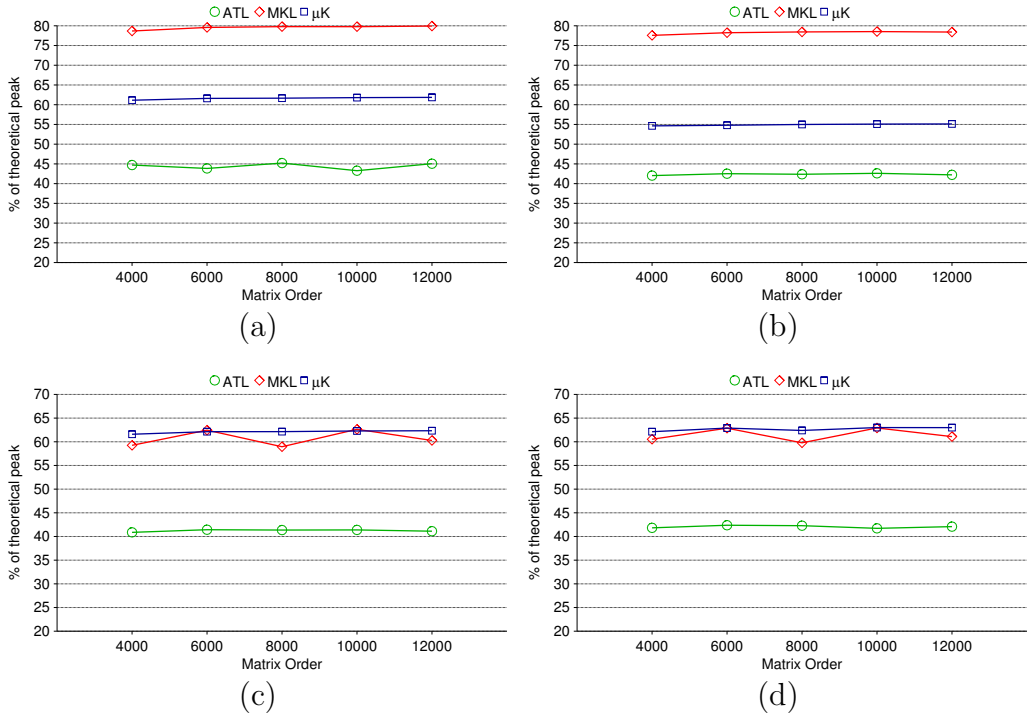


Figure 2.44: Performance of double-precision real TRSM on X12 for fat problems with constant triangle size of (120×120) for variants: (a) LLNN (b) LUNN (c) RLNN (d) RUNN

concentrated hand-tuning has therefore been applied to MKL's left case. This suggests there is plenty of room for improvement for both cases for us (if adding TRSM to our autotuning doesn't already make this improvement). However, this hand-tuned optimization for the left case used by MKL may be difficult and fragile, given that the MKL group did not add it to their right case, despite their manpower advantage.

Figure 2.45 and 2.46 shows the double precision TRSM performance for square and fat problems, respectively, for all four variants on X24. While we see similar speedups over prior ATLAS, this machine is overall our worst-case when comparing to MKL. The main reason for that is probably that our GEMM (and remember that our *trsm*μ is based on our *gemm*μ) is around 2-4% slower than that of MKL's on this architecture. This discrepancy is one of the main reasons for ATLAS's ongoing *gemm*μ tuning framework redesign. It is therefore our expectation that the new framework will allow for improved *gemm*μ performance, which will bring these results closer to those of the X12.

For our *gemm*μ-based TRMM, we have preliminary results for the X24 only[6]. Figure 2.47 shows the performance for square problems. For large problems, we are already faster than MKL, and if we can improve our *gemm*μ performance as we hope, the gap should grow more pronounced. We believe our small problem loss has the same explanation as for TRSM.

For fat problems, as shown in Figure 2.48, our approach outperforms both stock ATLAS and MKL by up to 73%. MKL's relatively poor performance seems to indicate that it has not had near the tuning attention as the more widely used TRSM kernel (theoretically, TRMM should always run at least as fast as TRSM, since TRMM has more degrees of optimization freedom).

[6]TRMM was chronologically the last part of this dissertation research. Between achieving the TRSM and TRMM results, both O32 and X12 machines experienced hard drive crashes brought on by power outages. Both machines are now reinstalled, but we do not yet have our prototype codes reinstalled and correctly autotuning on these machines yet.

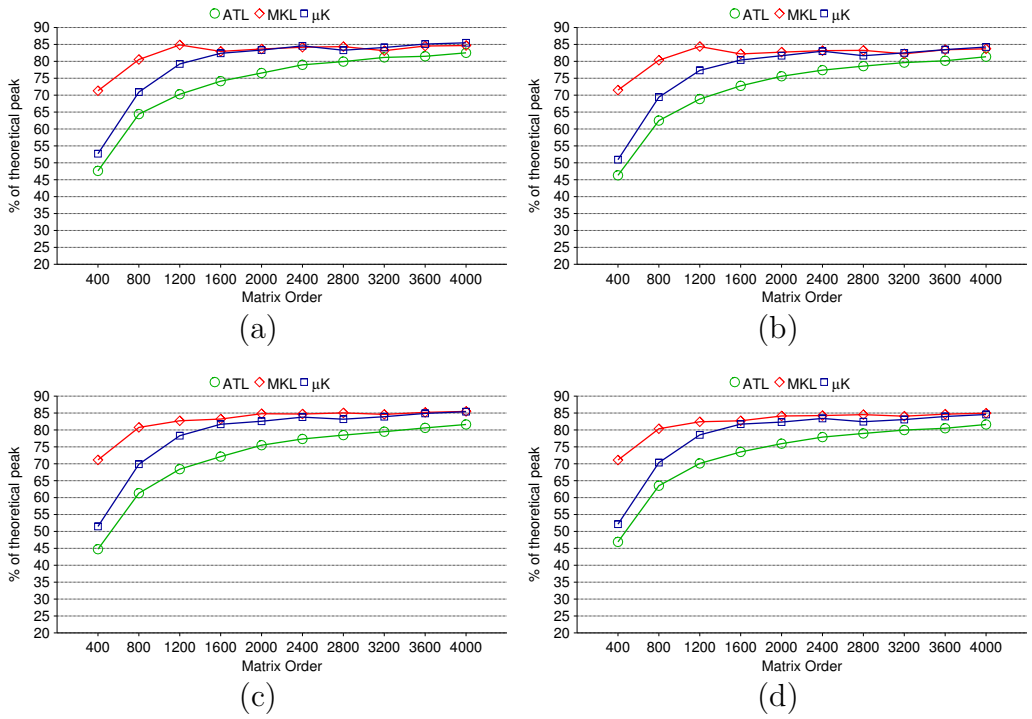


Figure 2.45: Performance of double-precision real TRSM on X24 for square problem sizes for variants: (a) LLNN (b) LUNN (c) RLNN (d) RUNN

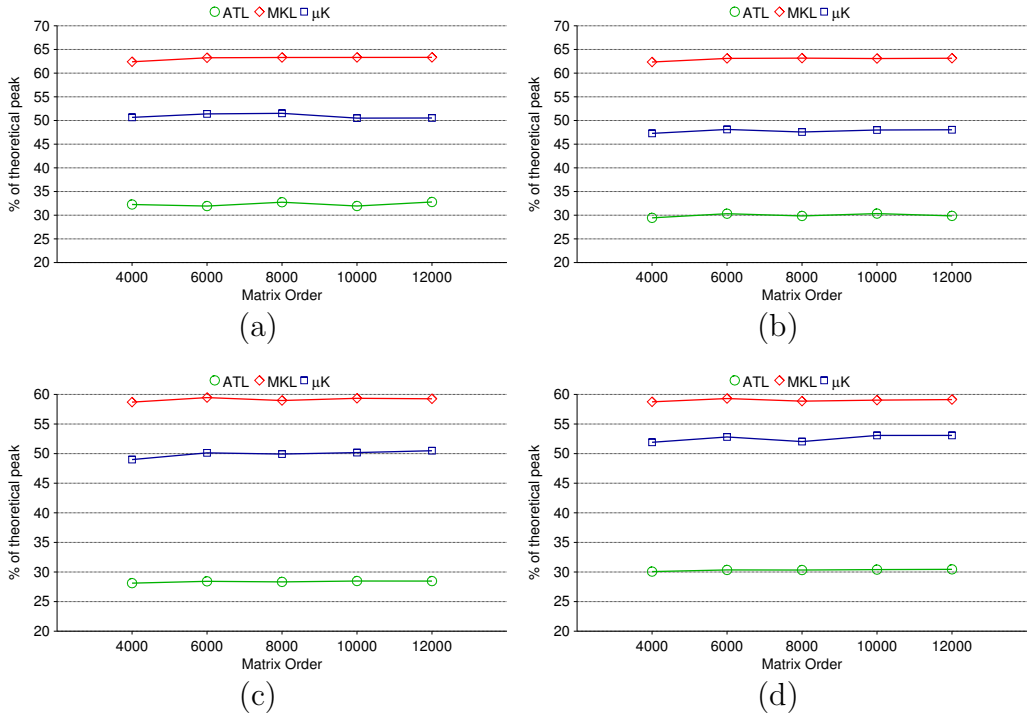


Figure 2.46: Performance of double-precision real TRSM on X24 for fat problems with constant triangle size of (120×120) for variants: (a) LLNN (b) LUNN (c) RLNN (d) RUNN

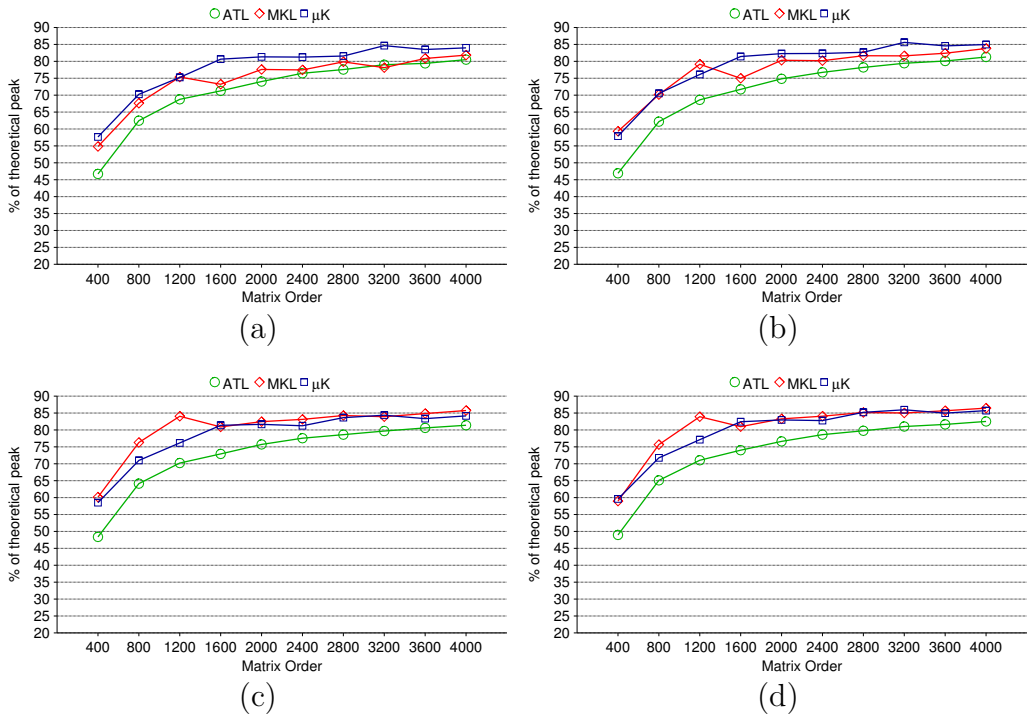


Figure 2.47: Performance of double-precision real TRMM on X24 for square problem sizes for variants: (a) LLNN (b) LUNN (c) RLNN (d) RUNN

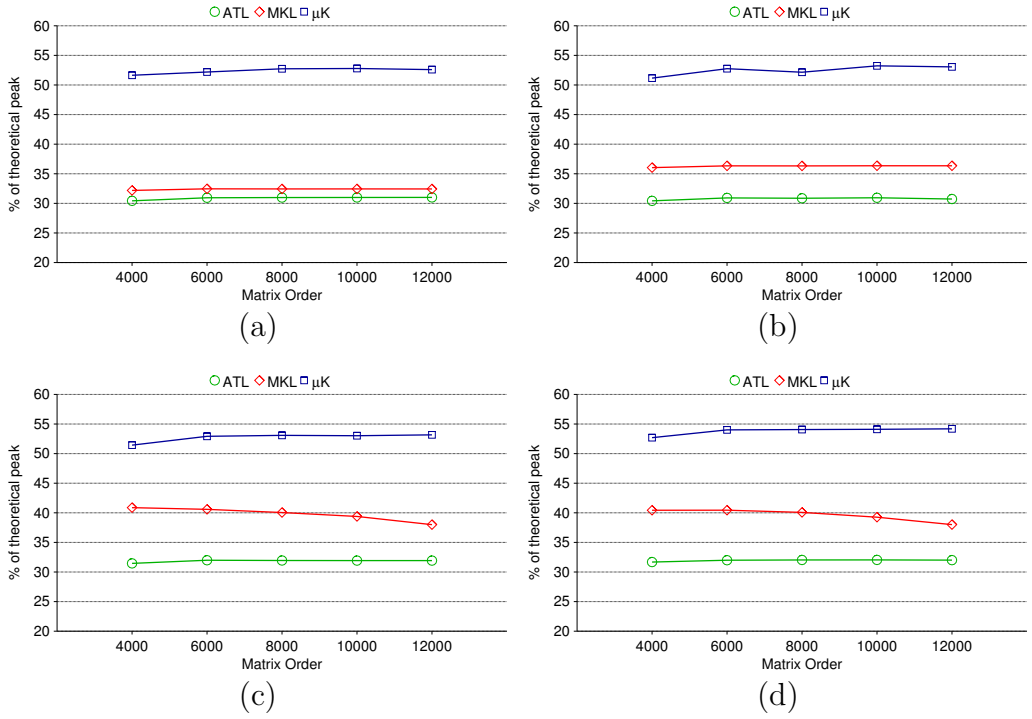


Figure 2.48: Performance of double-precision real TRMM on X24 for fat problems with constant triangle size of (120×120) for variants: (a) LLNN (b) LUNN (c) RLNN (d) RUNN

Given that these results should strongly improve when the autotuning framework is fully online, we view these preliminary TRSM and TRMM results as extremely encouraging, and a strong sign that this approach will ultimately prove effective.

2.5 Summary and Future Research

The purpose of this research is to improve the performance for all cases by reducing the BLAS overheads and to do that we copy the blocks once and reuse the copy multiple times throughout the entire algorithm. However, this optimization causes the approach to require extra memory space to hold the copied data for the entire execution time. For our TRMM and TRSM approach, we need a workspace for the entire triangle matrix and only one panel (column panel for left and row panel for right variants) of the rectangular matrix. If the required amount of memory is not available (which can happen for very large problems) our approach will fail. Note that this limitation can be present in other approaches as well. On the other hand, in a recursive approach like ATLAS, if the top level of recursion cannot allocate the memory space it requires, it can just go into the next (deeper) level of recursion where the problem size is half of that of the original problem size. It can continue recurring until the required memory space is available in the system. Because of this, our plan is to merge our $gemm\mu$-based approach to the recursive approach so that recursion only happens until we can allocate enough memory to use our iterative approach directly. With this hybrid approach: a) for small problems, only our approach will be used thus providing high performance for small problems and b) for asymptotic problems, ATLAS's recursive approach will provide high performance due to lion share of computations done by top level GEMM updates with asymptotic sized operands. Another limitation of our approach is that it is statically blocked which means we need to find and select best the block factors for a given input problem. In Appendix B, we will discuss how we can build a computational model to predict the potentially best block factors for a given problem.

CHAPTER 3
PERFORMANCE OPTIMIZATION OF LAPACK ROUTINES

LAPACK provides various numerical algebra routines including matrix factorizations, eigenvalue problems etc. ATLAS provides optimized versions for some of the LAPACK routines. In this chapter, we will discuss how we can improve the performance of two matrix factorizations that are widely used in computational sciences. Like TRMM and TRSM (discussed in Chapter 2), we will use the *gemm*μ framework to achieve high performance for all problem sizes. In Section 3.1, we will discuss the LU factorization which is widely used for solving systems of linear equations and matrix inversions. In Section 3.2, we will briefly discuss the Cholesky factorization which, unlike LU, only works for symmetric positive definite matrices. Note that this chapter mainly provides an introductory discussion on improving the performance of serial LAPACK routines as an aid in understanding for improving the performance of parallel LAPACK routines, which is discussed in Chapter 4.

3.1 LU Factorization

LU factorization is a method of decomposing a matrix into the product of: a unit-lower-triangular (L) and an upper-triangular (U) matrices. This factorization is done by performing Gaussian elimination on the input matrix. Depending on the pivoting strategy, there are multiple variants of LU factorization. No-pivoting yields the least stable solution while full-pivoting (finding the maximum element of the entire matrix) provides the most stable solution but is too expensive to perform. LAPACK uses partial-pivoting (finding the maximum element of a column) which is sufficiently stable for most usage. In our *gemm*μ-based implementations of LU factorization (both serial and parallel), we used partial-pivoting as well. Even for LU factorization with partial pivoting, there are variants of implementations depending on the order of computations. The two most common variants are right-looking and left-looking. In this section, we will discuss both variants in detail. With partial-pivoting,

the LU factorization of a matrix A is defined as follows:

$$A = PLU$$

where L is a unit-lower-triangular matrix, U is an upper triangular matrix and P (called the permutation matrix) contains the pivoting information (used to reorder the rows of A). Note that L is a unit-triangular matrix (diagonal elements are unit). Omitting these unit values, LAPACK implementation stores both L and U in the same storage as the input A i.e. non-diagonal L is stored below the diagonal of A and U is stored above (and including) the diagonal of A. Also, instead of having a permutation matrix where most of the elements are known to be zero, it stores only the indices of the rows to pivot in the integer array *ipiv*.

To understand how an LU factorization works, consider the example of a 3×3 matrix A shown below.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \tag{3.1}$$

The columns of A are factorized one at a time from left to right. Let A_{31} be the maximum element (also known as the pivot element) of the first column. For partial-pivoting, we need to swap the first row with the third row (therefore the *ipiv* array will hold 3 at index 1). The matrix after the pivoting is shown in below.

$$\begin{bmatrix} A_{31} & A_{32} & A_{33} \\ A_{21} & A_{22} & A_{23} \\ A_{11} & A_{12} & A_{13} \end{bmatrix}$$

Note that finding the maximum element is done by using a Level-1 BLAS operation called IAMAX. Also note that after the swap, the first row has become the first row of U. At this point, all the elements of the first column except the first (the pivot element) are scaled using the pivot element with the Level-1 operation SCAL. Note that in the Gaussian elimination method, the whole column is scaled using the pivot element which causes the diagonal

element to be 1. Since we are using the space for the diagonal element to store the pivot (part of U), we only scale the elements below the diagonal. After the scaling, these elements are transformed to be the first column of L as shown below.

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & A_{22} & A_{23} \\ L_{31} & A_{12} & A_{13} \end{bmatrix}$$

Next, we need to subtract the contribution of these new L and U elements (except the diagonal) from the trailing matrix as shown below.

$$\begin{bmatrix} \hat{A}_{22} & \hat{A}_{23} \\ \hat{A}_{12} & \hat{A}_{13} \end{bmatrix} = \begin{bmatrix} A_{22} & A_{23} \\ A_{12} & A_{13} \end{bmatrix} - \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \times \begin{bmatrix} U_{12} & U_{13} \end{bmatrix}$$

This subtraction is done by a Level-2 routine called GER (GEneral Rank-1 update). After the GER, this updated trailing matrix (i.e. $\hat{A}_{22}$, $\hat{A}_{23}$, etc.) becomes a sub-problem of the original LU factorization and we can repeat the same process on this trailing matrix to factorize the whole A which results the following matrix:

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & L_{32} & U_{33} \end{bmatrix}$$

As shown above, after processing each column, we update all the columns to the right using GER. Therefore, the above approach is called the right-looking variant of LU factorization.

For the left-looking variant, instead of updating the entire trailing matrix in all the steps above, only the current column is updated and then factorized. With this approach, we end up with the following matrix after processing the first two columns of the example input A shown in Equation 3.1.

$$\begin{bmatrix} U_{11} & U_{12} & A_{13} \\ L_{21} & U_{22} & A_{23} \\ L_{31} & L_{32} & A_{33} \end{bmatrix}$$

Note that the third column is untouched at this point. First, we need to apply a series of row-swaps on this column as indicated by the pivot array. Assuming the second column did not need any pivoting, the third column will hold $[A_{33}\ A_{23}\ A_{13}]^T$ after the row-swaps are done. Next we need to perform a triangular solve (TRSV) on the vector $[A_{33}\ A_23]^T$ to transform it into part of U. This triangular solve uses L_{21} with the presumed unit-diagonal in place of U_{11} and U_{22}. The resulting matrix after the solve is shown below.

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & L_{32} & A_{13} \end{bmatrix}$$

Note that after the solve A_{13} is still the unmodified element of the input A. Therefore we need to subtract the contribution of all the columns (i.e. Ls) on the left and the corresponding Us from this unmodified A_{13}. This subtraction is done by another Level-2 operation called GEMV. The operation looks like the following:

$$\begin{bmatrix} \hat{A}_{13} \end{bmatrix} = \begin{bmatrix} A_{13} \end{bmatrix} - \begin{bmatrix} L_{31} & L_{32} \end{bmatrix} \times \begin{bmatrix} U_{13} \\ U_{23} \end{bmatrix}$$

As we can see, we need to access (look at) all the columns to the left to update the current column, hence this approach is known as left-looking. Since we work on one column at a time, at the end of factorizing the last column, we need to apply a series of row-swaps to each column to reflect the required pivoting from all the factorized columns on its right. Note that both the right-looking and the left-looking variants discussed above worked on one column at a time (commonly known as unblocked LU factorization). We can apply both approaches with the cache-blocking optimization technique as discussed in Section 3.1.1.

3.1.1 Statically Blocked LU Factorization

For blocked LU factorization, a column-panel is factorized at a time instead of a column and all the updates are performed with Level-3 operations TRSM and GEMM. After the required TRSM and GEMM updates, we can factorize the trailing panel with the unblocked LU

factorization as discussed in Section 3.1. We will refer to this step as the *panel factorization*. We will only discuss the left-looking variant of statically blocked LU factorization which we use for our *gemm*μ-based serial and parallel LU factorization, presented in this section and Chapter 4, respectively. The basic computational steps of a left-looking variant of GEMM-based LU factorization (LAPACK's implementation) is shown in Figure 3.1. To understand the implementation, consider the example shown in Figure 3.2a where the first two panels are already factorized. Note that the other panels of the input matrix A are unmodified at this point. As we start working on the third panel (in general, we refer to this as the current-active panel *cpan*), we need to apply the updates from all the panels on the left (pan0 and pan1 in this case) before we can perform the panel factorization. The steps required to factorize *cpan* (i.e. pan2) of A are shown in Figure 3.2b-g.

The first step is to apply B_N row-swaps to the top block (e.g. A_{20}) of the *cpan* based on the pivot array of pan0 factorization. After that we use the lower triangular part of the diagonal block of the factorized pan0 (i.e. D_0) to perform TRSM on the pivoted A_{20} to transform it to a part of U (i.e. U_{20} as shown in Figure 3.2b). At this point, we subtract the contribution of pan0 factorization from A_2 using GEMM (i.e. $A_{20} = A_2 L_0 \times U20$) as shown in Figure 3.2c. The next step is to apply the updates for pan1 factorization on the modified A_{20} and we repeat the same steps: apply pivots from pan1 factorization on the top block of A_{20} (note that it is partitioned into a block and a trailing panel A_{21}) and perform TRSM on the pivoted

- **foreach** *active panel* from left to right in matrix:
  1. **foreach** panel to the left of active panel, update the active panel by:
     (a) Apply pivots from diagonal block
     (b) solve (TRSM) using diagonal block, creating block of U
     (c) apply L (GEMM) below diagonal block to rest of panel
  2. Perform panel factorization on remaining non-U blocks of *active panel*
- As last step, apply pivots (from panels to the right) to non-diagonal part of L

Figure 3.1: Basic steps for left-looking LU factorization for square matrices

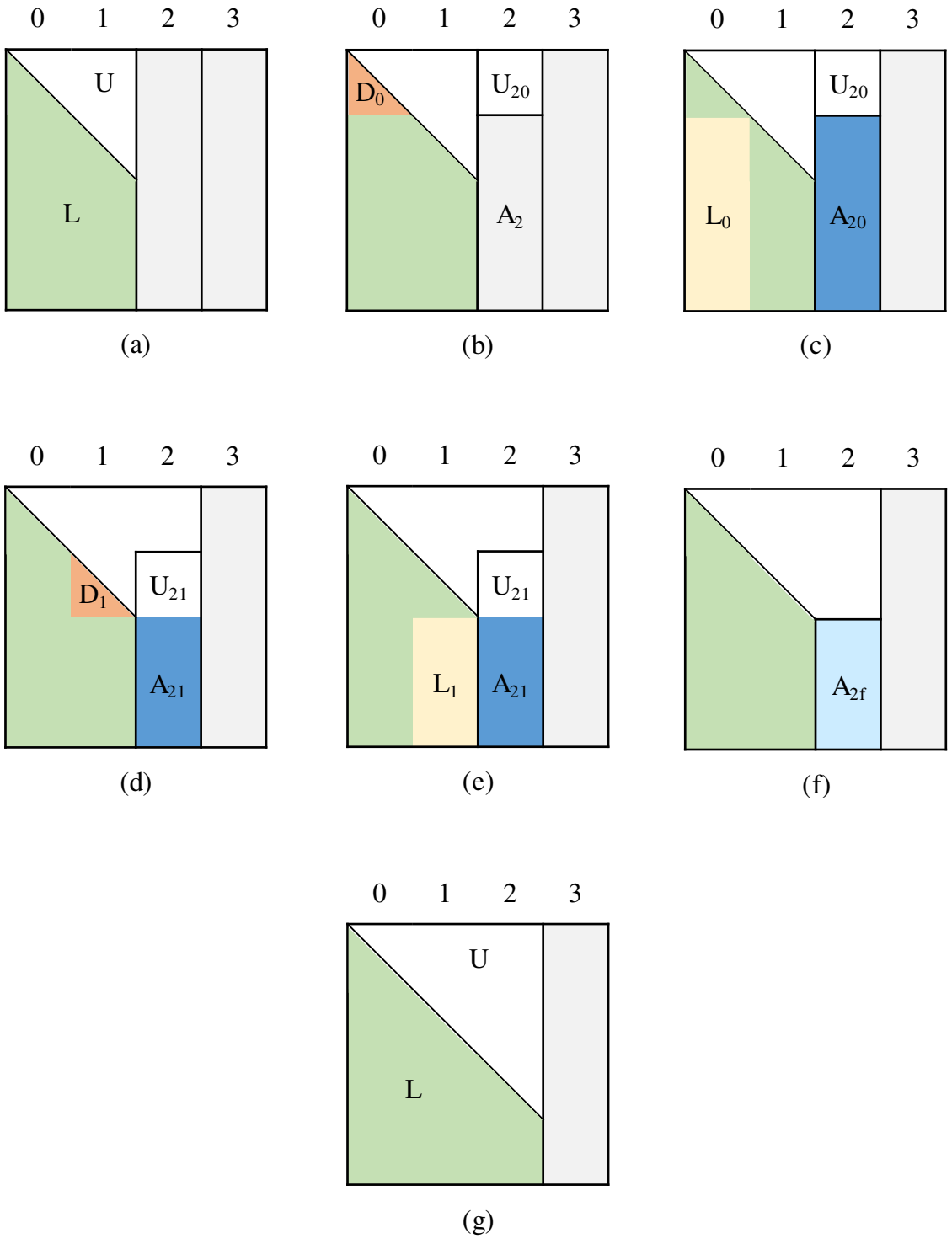


Figure 3.2: Computational steps of a left-looking LU factorization: (a) Initial state of A after pan0 and pan1 are factorized. (b) Apply row-swaps and TRSM using D_0 on top block of pan2 to make it U_{20}. (c) Apply GEMM updates on A_2 using L_0 and U_{20}. (d) Apply row-swaps and TRSM using D_1 on top block of A_{20} to make it U_{21}. (e) Apply GEMM updates on A_{21} using L_1 and U_{21}. (f) Perform the panel factorization on A_{2f}. (g) Final state of A after pan2 is factorized.

top block of A_{20} to transform it into U_{21} as shown in Figure 3.2d and then subtract the contribution of pan2 factorization from A_{21} using GEMM i.e. $A_{2f} = A_{21}L_1 \times U_{21}$. Note that we are referring to this trailing panel as A_{2f} since after the last GEMM update, this trailing panel of pan2 is finally ready to be factorized which can be performed by using an unblocked LU factorization. Later we will see that instead of using an unblocked LU factorization, we can use ATLAS's recursive LU factorization or have another level of static blocking for better performance. Recall that depending on the implementation of the BLAS library used, the GEMM updates may copy the required data (i.e. L_0, L_1, etc.). Since similar steps are needed for each of the later panels, these data may be copied repeatedly. If the number of panels is N_p, then L_0 might be copied $N_p - 1$ times, L_1 might be copied $N_p - 2$ times and so on. We refer to these copies as *Lcopy* for GEMM updates. Note that there is another place where duplicating copy might occur depending on the GEMM implementation. The result of the GEMM updates i.e. A_{20} and A_{21} share common storage except for the block U_{21}. If required, the data in this common storage are possibly copied to GEMM's required storage format and then copied back out at the end of GEMM. We refer to these copies as *Ccopy* of GEMM. These duplicating copies (both *Lcopy* and *Ccopy*) can be a significant overhead, particularly for small sized problems and small block factors. ATLAS provides a recursively blocked LU factorization that provides superior performance for asymptotic problems but still suffers from poor performance for small sized problems due to significant BLAS overhead. We will briefly discuss this approach in Section 3.1.2.

For our *gemmμ*-based approach, to avoid these duplicating copies, we will copy the required data to the *gemmμ* required storage format and directly call the *gemmμ* to apply the GEMM updates. The L blocks within a column panel are copied only once and we copy them (i.e. *Lcopy*) right after the factorization on that panel is done, while they may still be cache resident. Note that after all the required GEMM updates, the portion of the panel that needs to be factorized is copied back to the original storage in column-major format to

avoid the complexity of performing the panel factorization on *gemmμ* storage. This copying back to the original storage would still be needed if we perform the factorization on *gemmμ* storage with a custom panel factorization because we need to copy back the result to the original storage anyway. The basic steps of our *gemmμ*-based LU factorization are shown in Figure 3.3.

Note that to avoid repeated *Ccopy*, as we start working on a new panel, we copy it (omitting the topmost block) to the *gemmμ* required storage format in the first step. The topmost block is omitted because it does not require any GEMM update before pivoting and TRSM. Then as the other blocks become ready to be pivoted and solved (TRSM) we copy it back to the column-major storage. Note that the copies of L_0, L_1, etc. are reused throughout the entire algorithm while the *Ccopy* blocks are reused for all the GEMM updates on that panel.

3.1.2 Improving the Panel Factorization

As discussed in Section 3.1.1, LAPACK uses the unblocked LU factorization to perform the panel factorization steps. Recall that an unblocked LU factorization is performed using Level-1 and Level-2 BLAS which are memory-bound operations with low serial and parallel performance. To optimize the panel factorization, we can apply the blocking optimization again on the panel with a smaller block factor which allows us to use Level-3 operations with smaller operands. An alternative approach is to implement recursive blocking [31, 29] like ATLAS does for its full LU factorization which can provide better performance for wider panels. To understand the recursively blocked LU factorization, consider the example shown in Figure 3.4 for factorizing a panel A is of size $M \times B_N$ where M is the number of rows in the panel and B_N is the number of columns. In recursively blocked LU factorization, the input panel A is partitioned into two sub-panels: A_0 and A_1. The first step is to recursively factorize the sub-panel A_0. Next is to apply pivots and perform TRSM on the top $\frac{B_N}{2} \times \frac{B_N}{2}$ sized block of the second sub-panel. After the TRSM, this block becomes a part of U. The

- **foreach** *active panel* from left to right in matrix:
  1. copy all blocks except the topmost to *gemmμ* C-storage
  2. **foreach** panel to the left of active panel, update the active panel by:
     (a) if not first update, copy the top C-storage block to column-major storage
     (b) apply pivots from diagonal block
     (c) solve (TRSM) using the diagonal block, creating a block of U in the *active panel*
     (d) copy the new block of U to *gemmμ* B-storage
     (e) apply previously copied L (GEMM) below diagonal block to rest of panel using *gemmμ*
  3. copy-back the remaining non-U blocks to the original matrix and perform panel factorization on it
  4. copy the below-diagonal portion (L) of the factorized panel to *gemmμ* A-storage
- As last step, apply pivots (from panels to the right) to non-diagonal part of L in the original storage

Figure 3.3: Basic steps for *gemmμ*-based left-looking LU for square matrices

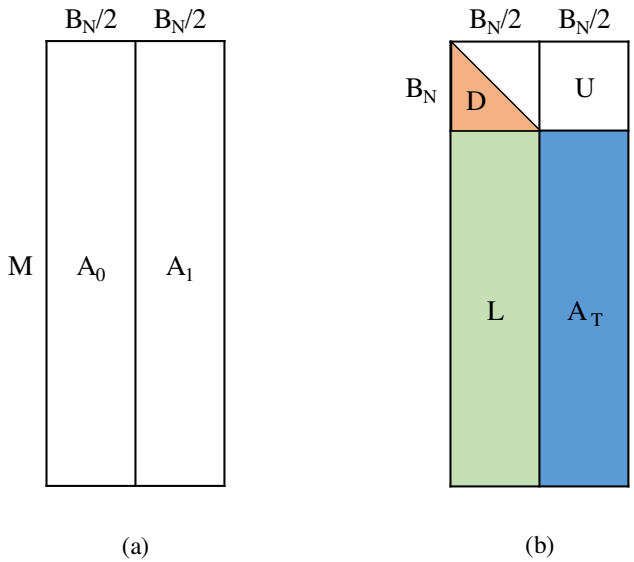


Figure 3.4: Basic steps for recursive LU factorization of a panel: (a) partitioning of the panel. (b) updates on the second half of the panel after first panel is factorized.

next step is to subtract the contribution of the first factorized panel from the trailing panel of (say A_{11}) with a GEMM update i.e. $A_T = A_{11}L \times U$. At this point, A_T is ready to be factorized which done recursively as before.

Note that we can implement our previously discussed *gemm*μ-based approach for panel factorization as well. However, with the wide variety of widths needed, and the amount of non-GEMM computation required, we have not yet found a way to approach this problem without increasing code complexity beyond what is reasonable given the limited performance improvement we have so far seen with this approach.

3.2 Cholesky Factorization

Cholesky factorization is a method to decompose a symmetric, positive-definite matrix. It is commonly used in Monte Carlo simulations and inverting Hermitian/symmetric, positive-definite matrices. An unblocked Cholesky factorization in LAPACK is done by a modified Gaussian elimination method. There are two variants of Cholesky depending of the parameter named $UPLO$. The factorization is defined as follows:

$A = LL^T$ if $UPLO = L$, i.e. the symmetric A is stored only in the lower-triangular part

or

$A = U^TU$ if $UPLO = U$, i.e. the symmetric A is stored only in the upper-triangular part

Note that depending on the order of the computations, both the above Lower and Upper versions of Cholesky have variants such as right-looking, left-looking, etc. For this research, we only explored the left-looking Lower Cholesky factorization using *gemm*μ-based approach but the Upper version can be implemented using the similar approach.

Like LU, columns of the matrix A are factorized one at a time from left to right using a modified Gaussian elimination method and the result is stored in the same storage as the input. Unlike LU, only the lower-triangular part of A is accessed or updated with the result

for Lower Cholesky. For the factorization, the following steps are repeated for each column from left to right:

1. Subtract the contribution of any previously factorized columns from the diagonal element of the current column, using the L1BLAS routine DOT.

2. Take the square-root of the diagonal element to make it a part of the result L.

3. Subtract the contribution of any previously factorized columns from the below-diagonal elements of the current column, using the L2BLAS routine GEMV.

4. Scale the rest of the column below the diagonal using the square rooted value computed in the step 2, using the L1BLAS routine SCAL.

To understand how a Cholesky factorization works, consider the 4×4 triangular matrix shown below where the first two columns are already factorized:

$$\begin{bmatrix} L_{11} & - & - & - \\ L_{21} & L_{22} & - & - \\ L_{31} & L_{32} & A_{33} & - \\ L_{41} & L_{42} & A_{43} & A_{44} \end{bmatrix} \tag{3.2}$$

Note that only the lower triangular part is shown since strictly upper triangular part is not accessed during the factorization. For factorizing the third column, the following steps are done:

1. Subtract the contribution of the left two factorized columns from A_{33} using DOT i.e.

$$\hat{A}_{33} = A_{33} - \begin{bmatrix} L_{31} & L_{32} \end{bmatrix} \times \begin{bmatrix} L_{31} \\ L_{32} \end{bmatrix}$$

2. Compute $L_{33} = sqrt(\hat{A}_{33})$.

3. Subtract the contribution of the left two columns from below diagonal elements (only A_{43} in this case) using GEMV i.e.

$$\begin{bmatrix} \hat{A}_{43} \end{bmatrix} = \begin{bmatrix} A_{43} \end{bmatrix} - \begin{bmatrix} L_{41} & L_{42} \end{bmatrix} \times \begin{bmatrix} L_{31} \\ L_{32} \end{bmatrix}$$

4. Scale the below-diagonal elements of current column, $L_{43} = \hat{A}_{43}/L_{33}$.

After above steps, the third column is completely factorized as shown below.

$$\begin{bmatrix} L_{11} & - & - & - \\ L_{21} & L_{22} & - & - \\ L_{31} & L_{32} & L_{33} & - \\ L_{41} & L_{42} & L_{43} & A_{44} \end{bmatrix}$$

We can repeat the steps for the fourth column to fully factorize the matrix. Since this approach factorizes one column at a time, it is commonly known as the *unblocked* Cholesky factorization. As with LU factorization, we can use blocking (both static and recursive) to improve the performance as discussed later.

3.2.1 Statically Blocked Cholesky Factorization

As in GEMM-based LU, cache blocking can be applied to improve the performance of the Cholesky factorization. To understand the GEMM-based Cholesky factorization, consider the example shown in Figure 3.5a where the first two panels are factorized. The first step is to subtract the contribution of all the factorized panels on left (L_{20}) from the diagonal block A_{2S} as shown in Figure 3.5b. At this point, the diagonal block is ready to be factorized (shown in Figure 3.5c). Note that both the SYRK and the factorization only access the lower-triangular part of the diagonal block. The matrix A after factorizing the diagonal block is shown in Figure 3.5d. The next step is to subtract the contributions of all the factorized panels on left from the rest of the current panel (A_2) i.e. $A_M = A_2 - L_{21}L_{20}^T$ (shown in Figure 3.5e). Finally, we need to perform the solve on A_M using the lower-triangular part of the diagonal block D_2.

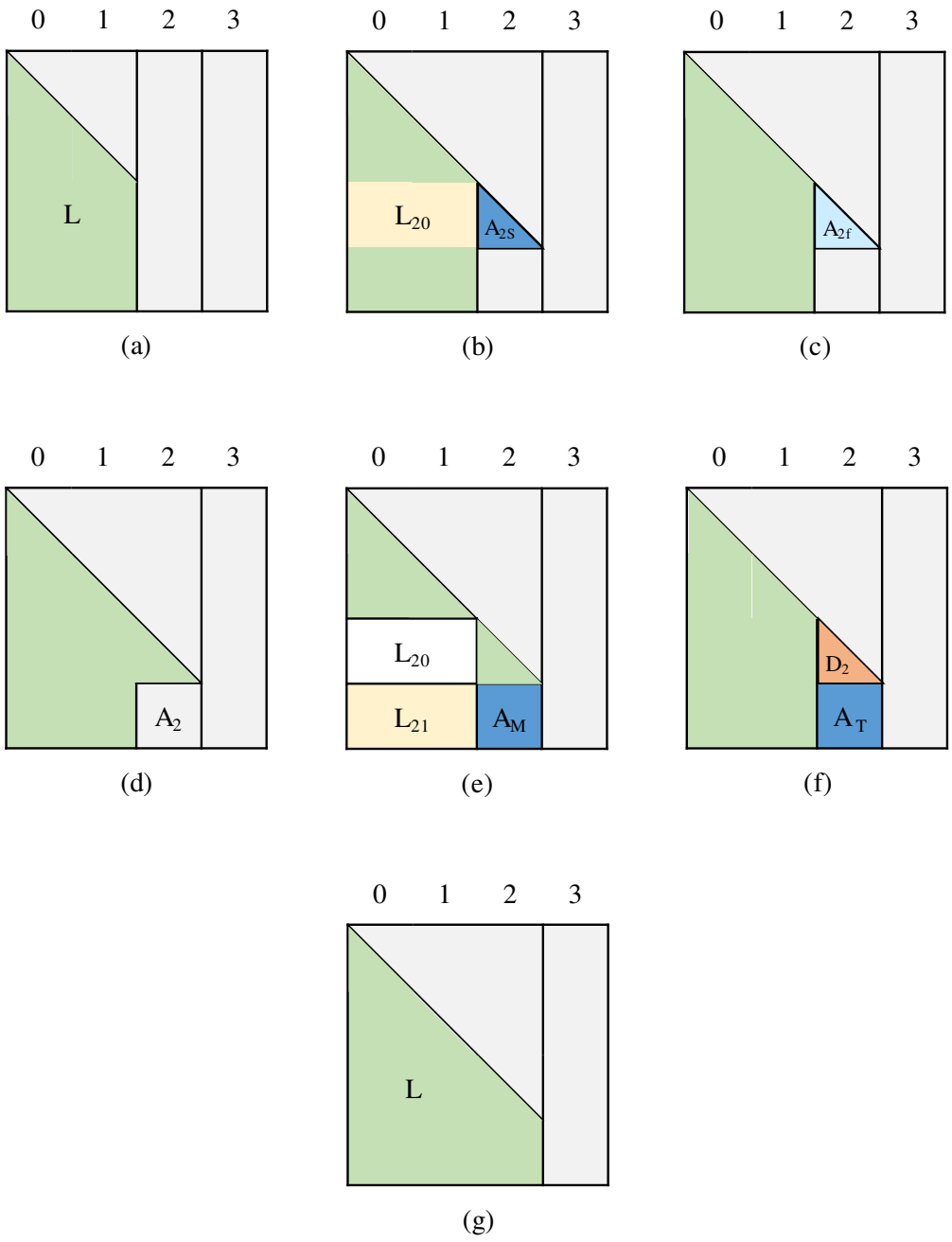


Figure 3.5: Basic steps for Lower Cholesky factorization of a panel: (a) Initial state of A after pan0 and pan1 are factorized. (b) Apply SYRK on A_{2S} using the corresponding row panel (L_{20}) on the left. (c) Factorize the updated diagonal block A_{2f}. (d) State of A after factorizing the diagonal block A_{2f}. (e) Apply GEMM updates on below-diagonal blocks (A_2) using the corresponding row panels on left (L_{21}) and the diagonal row panel (L_{20}). (f) Solve the updated below-diagonal blocks (A_T) using the lower-triangular part of the diagonal block (D_2). (g) Final state of A after pan2 is factorized.

The final state of A after pan2 is factorized is shown in Figure 3.5g. Note that the diagonal blocks are updated using SYRK (Level-3 BLAS operation) that performs a symmetric rank-K update on a triangular matrix. The basic steps for a GEMM-based left-looking Lower Cholesky factorization for is shown in Figure 3.6.

As before, our approach is to use *gemm*μ framework to improve the performance of GEMM-based Cholesky factorization for all sizes. The idea is to explicitly copy the factorized panels to *gemm*μ required storage and directly call the *gemm*μ as needed for all the GEMM updates. Note that the SYRK operation is similar to GEMM with the exception that it multiplies a symmetric matrix with its transpose (which leads to less floating point operations required than GEMM). For this research, we use *gemm*μ to perform the SYRK operation as well which leads to extra computations. At the time of this research, ATLAS did not provide any SYRK microkernel that could minimize these extra computations. In latest ATLAS release, such SYRK microkernels are included in its framework which, in future research, we can utilize to improve the performance of our *gemm*μ-based Cholesky factorization. The basic steps for *gemm*μ-based Cholesky factorization are shown in Figure 3.7.

3.3 Summary and Future Research

In this chapter, we presented the basic idea for improving LAPACK operations with two example routines: LU and Cholesky factorization. As mentioned before, our future research plan includes utilizing the SYRK microkernel in the latest ATLAS release to further optimize

- **foreach** *active panel* from left to right in matrix:
    1. Subtract the contribution from the diagonal block for all the factorized panels on left using SYRK
    2. Factorize the diagonal block
    3. Subtract contribution from all the below-diagonal blocks for all the factorized panels on left using GEMM
    4. Solve the below-diagonal blocks using the lower-triangular part of the diagonal block

Figure 3.6: Basic steps for left-looking Lower Cholesky factorization

- **foreach** *active panel* from left to right in matrix:
    1. Subtract contribution from the diagonal block for all the factorized panels on left using *gemmμ* and previously copied blocks
    2. Factorize the diagonal block
    3. Subtract contribution from all the below-diagonal blocks for all the factorized panels on left using *gemmμ* and previously copied blocks
    4. Solve the below-diagonal blocks using the lower-triangular part of the diagonal block
    5. Copy the solved below-diagonal blocks as both transpose and no-transpose *gemmμ* format for later use

Figure 3.7: Basic steps for *gemmμ*-based left-looking Lower Cholesky factorization

our *gemmμ*-based Cholesky factorization. Moreover, we want to improve the performance of QR factorization which is used for finding Eigenvalues and solving linear least squares problem.

CHAPTER 4

PERFORMANCE OPTIMIZATION OF PARALLEL LU FACTORIZATION

This chapter is an extended version of the previously published paper [37][1]. LU factorization is one of the most widely-used methods for solving linear equations, and thus its performance underlies a broad range of scientific computing. As architectural trends have replaced clock rate improvements with increases in parallel scale, library writers have responded by using tiled algorithms, where operand size is constrained in order to maximize parallelism, as seen in the well-known PLASMA library [1, 13, 14, 59, 48, 12, 26, 34, 26, 21, 78]. This approach has two main drawbacks: (1) asymptotic performance is reduced due to limited operand size; (2) performance of small to medium sized problems is reduced due to unnecessary data motion in the parallel caches. In this paper we introduce a new approach where asymptotic performance is maximized by using special low-overhead microkernel primitives that are auto-generated by the ATLAS framework, while unnecessary cache motion is minimized by using explicit cache management. We show that this technique can outperform all known libraries at all problem sizes on commodity parallel Intel and AMD platforms, with asymptotic LU performance of roughly 91% of hardware theoretical peak for a 12-core Intel Xeon, and 87% for a 32-core AMD Opteron.

4.1 Introduction

LU factorization (essentially Gaussian elimination with partial pivoting) is a critical component for scientific computing. This functionality is exposed to the user via the LAPACK [3, 50] (Linear Algebra PACKage) API. In this chapter we discuss the double precision real version of this routine, and will refer to it using its LAPACK name DGETRF (Double precision GEneral TRiangular Factorization). The main computational component of DGETRF is

[1]This chapter previously appeared as [Md Rakib Hasan and R. Clint Whaley, International Parallel & Distributed Processing Symposium, published by The Institute of Electrical and Electronics Engineers (IEEE)]. See the reuse permission letter in Appendix C.

matrix multiply, which is provided by the BLAS [22] API under the name DGEMM (Double precision GEneral rectangular Matrix-matrix Multiply). DGEMM is typically the most efficient (in both parallel and serial performance) of the provided BLAS operations. The second-most important BLAS call for DGETRF is DTRSM (Double precision TRiangular Solve to a Matrix of right-hand sides).

4.2 Experimental Details

Timings presented in this paper were run on Debian 6 (Linuxv2.6.32-5-amd64), gcc 4.7.0, with ACMLv5.3.1, ATLASv3.11.14, FLAMEr11400, LAPACKv3.4.2, MKLv10.3.9, PLASMAv2.5.1. All timings are for LU with LAPACK-equivalent partial pivoting, using the default block sizes provided by the libraries. PLASMA offers both static and dynamic scheduling options; we report static timings, since static scheduling gave dramatically better small-case performance with essentially the same asymptotic performance. Since parallel timings are volatile (particularly for smaller sizes), we always report the average across a number of timing runs. All our timings are on square matrices of order N. For $N \leq 2000$, we average fifty timings, for $2000 < N \leq 10,000$ six timings, for $10,000 < N \leq 20,000$ three timings, and for problems of greater size we average only two timings.

Timings were performed on two commodity shared-memory platforms: (1) **O32**: 32-core, 2.0 Ghz AMD Opteron 6128, organized on a 4-socket motherboard. Theoretical peak for entire machine is 256 GFLOPS; (2) **X12**: 12-core, 2.0 Ghz Intel Xeon E5-2620, organized on a two-socket motherboard wt theoretical machine peak of 192 GFLOPS.

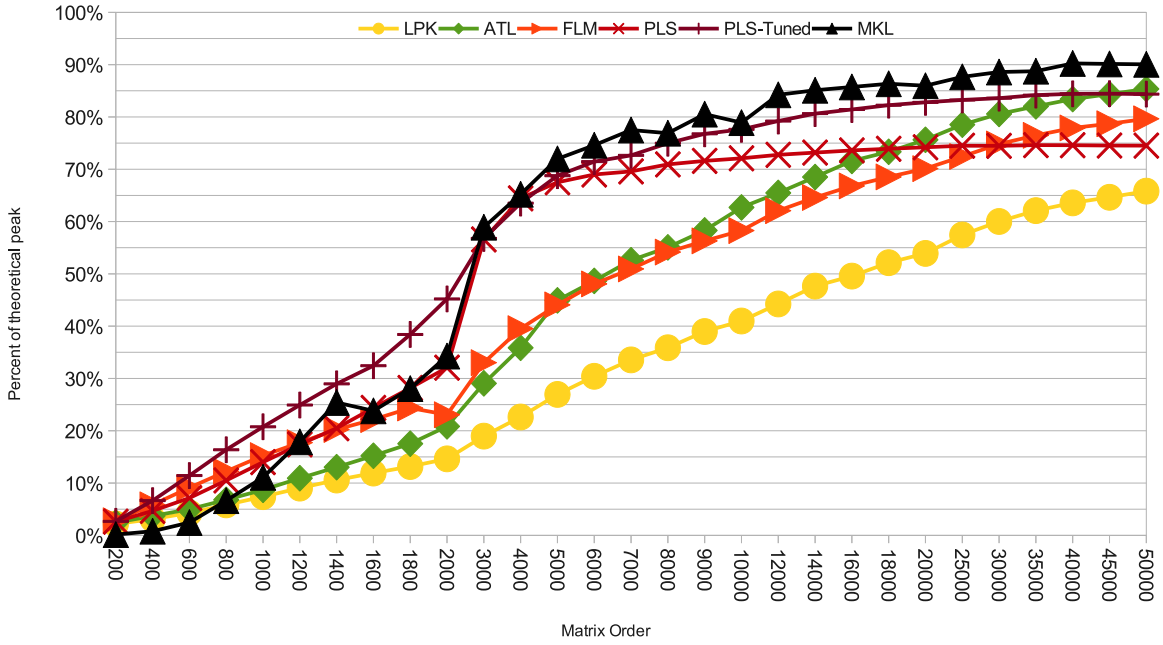
Our results are reported as a percentage of the given theoretical machine peak, using the FLOP count provided by LAPACK's `dopla.f` [49], which for LU is roughly $\frac{2}{3}N^3$. We use the first touch NUMA memory initialization technique discussed in Section 5.2 of [15] to initialize all matrices, so that page ownership is distributed amongst the parallel cores. This ensures that approaches that copy the input matrix (e.g. our approach and PLASMA) do not have a huge advantage over ones that do not (ATLAS, LAPACK, MKL) based merely

on input initialization. For each machine, we time problems as large as possible without thrashing virtual memory, which leads to a max size of 50,000 for X12 and 45,000 for O32.

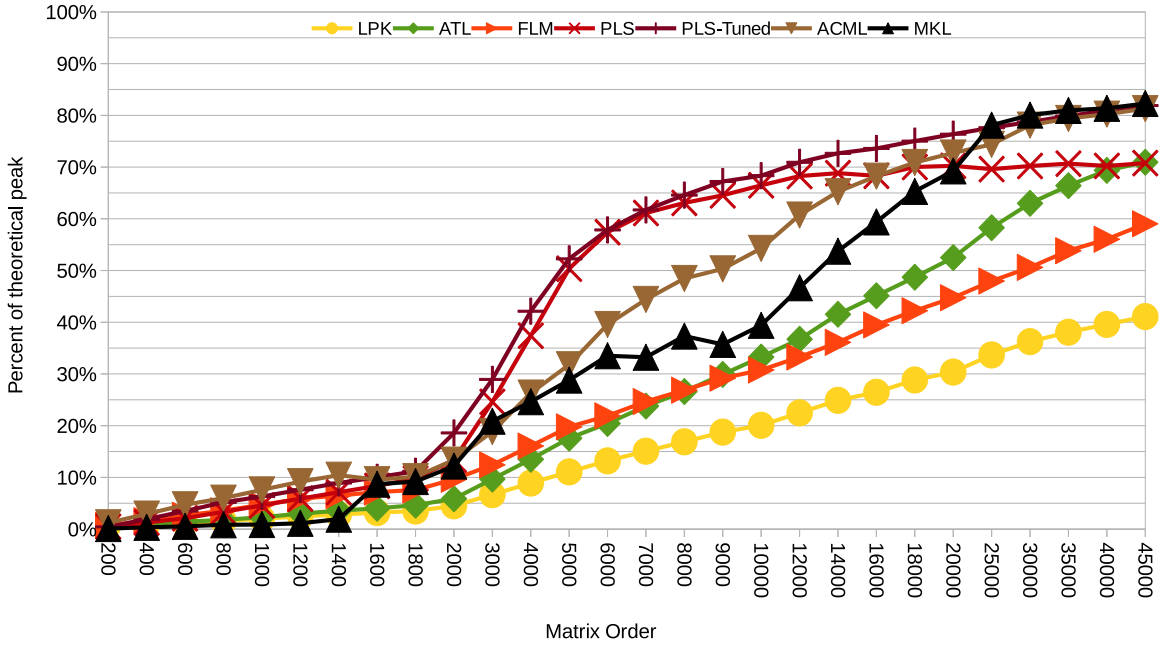
4.3 History, Motivation and Related Work

Historically, parallel DGETRF providers could focus almost exclusively on asymptotic performance. These are the only problem sizes that *require* parallel solutions, and users with small- to medium-sized problems could count on continuing performance gains from increasing clock rate. Since improving clock rate has been largely replaced by increasing parallel scale, it has become much more important to provide parallel speedup for modest sized problems as well. Therefore, Figure 4.1, which attempts to summarize the state-of-the-art in parallel DGETRF performance, has an X axis that has been manipulated to allow us to clearly see problems in this range ($N < 2000$). Note that the first nine data points have a stride of 200, while the larger sizes use a stride of 1000 between points. This leads to the illusion that there are huge increases in performance between points 2000 and 3000: this is due to the change in scale that occurs between these two points. The Y axis shows the percentage of theoretical peak achieved by various DGETRF implementations.

Netlib lapack using the ATLAS [76, 74, 75] parallel BLAS is the worst performer, as shown in yellow circles. Due to shortcomings in this original approach, the recursive formulation of LU was developed [63] and later implemented in ATLAS; the performance of this variant is shown as green diamonds in Figure 4.1. Due to its more effective exploitation of DGEMM, this algorithm is superior at all problem sizes compared to the original LAPACK. Both of these approaches have the strong advantage that they leave almost all of the optimization (both serial and parallel) to the BLAS, freeing the much larger LAPACK library to focus on the algorithmic level. The problem with this approach is its extremely slow rise in parallel performance with problem size; as parallel scale is increased, moderately sized problems will achieve an increasingly minuscule percentage of machine peak due to various inefficiencies and serializations.



(a)



(b)

Figure 4.1: Performance of LU factorization for netlib lapack (yellow circles), ATLAS (green diamonds), FLAME (orange point-right triangles), PLASMA (red x), empirically tuned PLASMA (dark red +), and MKL (black point-up triangles): (a) 12-core Intel Xeon E5-2620 (b) 32-core AMD Opteron 6128.

Therefore, the current wave of research has concentrated on breaking the problems up into sub-blocks (often called *tiles*) that can be explicitly scheduled. Examples of this general approach include both the FLAME [79, 17, 54, 80, 30, 39, 6, 56, 53] (though not for LU) and PLASMA [1, 59, 12, 26] libraries. In FLAME (orange, point-right triangles), their highest performing LU uses multi-level static blocking with blocking factors roughly tuned for this version MKL; this was shown to provide roughly the same performance as recursion in [69]. However, unlike recursion, this approach's optimality is strongly dependent on both the hardware and the BLAS implementation used, which is why it is generally less recommended than the recursive implementation. Since FLAME's LU is not meaningfully different from a hand-tuned LAPACK, we omit it from later charts for clarity.

The performance of PLASMA is shown as red "x" in Figure 4.1, and it demonstrates the fundamental problems experienced by these tiled algorithms. We see explicitly managing the parallelism at the LU level has paid off with an algorithm that scales much better for moderately sized problems than the prior approaches, but the performance hits a ceiling (in this case at just under 75% of peak), which leads to disappointing asymptotic performance despite scaling that is usually almost perfect in this range. The reason for this ceiling is straightforward: the BLAS contain optimizations such as copying input data to architecture-specific formats that must be amortized over the call to the operation. By breaking the problem into tiles of fixed sizes, these algorithms strongly constrain the problem sizes they call the BLAS with, which leads to these overheads remaining important at all problem sizes. Therefore this efficiency ceiling is in some sense a measure of the percentage of overhead when calling the BLAS with such small problem sizes. While this problem is unavoidable in the BLAS-based tiled approach, there is no reason for asymptotic performance to be this poor, since it should be possible to drastically increase the tile size for extremely large matrices while maintaining sufficient parallelism. Therefore, in order to get a best-case for the PLASMA approach, we wrote an autotuner that tried all relevant combinations of inner and outer blocking factors

for PLASMA, resulting in the much-improved empirically-tuned PLASMA curve shown as dark red "+". This leads to our first important observation: PLASMA's large-case (and to a lesser extent, small-case) performance could be drastically improved merely by taking problem size into account when choosing blocking factors; since this could be accomplished by instantiating some simple rules of thumb in a case statement, we are puzzled that PLASMA currently ignores problem size.

Note that on this machine, the recursive algorithm actually outperforms both PLASMA variants for extremely large problems: this is because recursion results in BLAS calls of dimensions that grow with problem size, with the largest call dominating execution time. For huge problems like this, low-order overheads are essentially free, and so recursion asymptotically approaches the speed of DGEMM, which is almost always very near machine peak.

Hardware vendors also provide optimized LAPACK and BLAS libraries, the most important of which are Intel's MKL [43] and AMD's ACML [2] (not shown in Figure 4.1a). We see that, on the Intel machine, MKL (black points-up triangles) and PLASMA scale roughly equally well for moderately sized problems, but MKL's performance continues to rise until it reaches a remarkable point at just over 90% of peak. Similar rise is seen for both MKL and ACML on the AMD machine as shown in Figure 4.1b. Since MKL and ACML are proprietary libraries tuned by the same companies that design the hardware, we cannot know all of the techniques MKL and ACML exploit to achieve this level of efficiency. Therefore, the question becomes: can we match or improve on this performance in a general library such as ATLAS? The answer, we will show, is a decided yes.

4.4 Our Approach

There are two problems seen in Figure 4.1 that we would like to address: (1) We would like to enable small-to-moderate sized problems to achieve a much greater percentage of machine peak than even the explicitly parallel approaches like PLASMA deliver; and (2) in explicitly parallelizing the operation, we must avoid the asymptotic ceiling problem that is typical of

the tiled/BLAS approach. We address these issues by employing several separate but related techniques.

In order to address moderate sized performance, it is critical to minimize the data movement, which is the dominant cost in this range (even though it is a low order term, and thus unimportant for asymptotic performance). Minimizing data movement at this level requires explicitly controlling (to the extent possible) how things are located in the cache, utilizing cache-optimized storage patterns, and using the owner-computes rule to ensure data loaded to a particular cache is not unnecessarily moved to another cache. We call this technique Parallel Cache Assignment (PCA), and in our previous work [15, 16] we showed that it can yield the most scalable and highest performing known algorithms for many bus-bound operations in this size range. In order to allow extremely fine-grained parallelization, we also exploit the x86's cache coherence mechanism to provide hardware-speed parallel synchronization and communication [15] (these overheads can be important in this range). In this earlier work, we concentrated on unblocked computations, but here we have adapted PCA for use in a multilevel statically-blocked left-looking LU. Our block handling directly descends from the distributed memory work of ScaLAPACK [44, 10]. Just as in ScaLAPACK, we lay out the cores in a 2-D process grid, and the data is then distributed amongst the threads (and thus amongst the caches) using a 2-D block cyclic distribution. We then explicitly manage and minimize movement in and out of caches using local copies and remote reads just like message passing is used in distributed memory parallelization to minimize communication (i.e. ScaLAPACK's off-node message becomes our out-of-cache operation); this allows us to reuse research on optimal communication patterns in this new context.

The main advantage of PCA is that when the problem is capable of being held in the collective cache of the machine, a PCA algorithm can achieve the theoretical minimum memory access allowed by the algorithm. Since bus bandwidth is the main constraint on performance in this range, this fact alone will greatly increase our performance anytime the

entire problem can be held in the collective cache. Since the collective cache has tended to grow at least weakly with parallel scale, the sizes of problems that can be cache-contained is now fairly impressive. The O32 machine has roughly 40MB of *usable* collective cache (some of the cache is reserved for use by AMD's cache coherence mechanism), while X12 has roughly 30MB of cache. Because caches don't use LRU replacement, they tend to have less effective space even when managed by PCA, and so we can expect our algorithms to start losing some of their cache advantage around $N = 1800$. The idea is to use PCA to increase performance until data movement becomes a dominated low-order term, and thus provide an algorithm that maximizes performance across all problem sizes.

Even if PCA can deliver good small and moderate-sized performance, the fact that we are using fixed-sized blocks has the possibility of constraining our asymptotic performance just as happens with the tiled/BLAS approach. In ScaLAPACK this was handled by aggregating the cyclically distributed blocks into normal column-major matrices within the local memories, which allowed for extremely large DGEMM calls to be made. While this is better asymptotically than storing data as tiles, it introduces a large copy cost into the algorithm which exerts a negative effect on parallel scaling of non-asymptotic problems. Furthermore, this storage pattern makes poor use of the memory hierarchy. In order to avoid these copy overheads, we rewrote our algorithm to directly call the microkernels that ATLAS uses to create its optimized DGEMM, rather than calling DGEMM. We also extended the ATLAS framework so that a user can provide ATLAS with a list of specific problem sizes to be auto-tuned during installation. This allows ATLAS to serve as an auto-tuner for blocked microkernel operations, in addition to doing its normal tuning of the BLAS.

To clarify, when ATLAS tunes a complex operation like DGEMM, ATLAS breaks the problem into simpler microkernels that can be effectively optimized on a particular machine. These GEMM microkernels are simplified matrix multiplications where the problem dimensions have been reduced so that the operands are known to fit into some level of cache. Thus,

the operands to these microkernels can be thought of as blocks or tiles. The problem then is that these blocks are not stored in normal array or block formats, but rather the formatting is arranged in a fashion that is tuned for both the operation and architecture [38]. ATLAS originally used one format for all machines, but has just been rewritten to allow the block storage to vary depending on architectural and microkernel details (even on the same machine, ATLAS may use a variety of storage patterns; for instance in order to handle different block shapes). Since the storage pattern is not fixed, it would seem impossible to write an LU factorization that uses it, but this can be handled by using a data copy. Our extended ATLAS framework not only generates the DGEMM microkernels themselves, it also automatically generates routines that copy normal row- or column-major storage to and from its internal block storage formats. Data copies are used to manage the cache anyway, and so these can be incorporated into the design in order to allow an application to automatically use variable storage patterns. Since the copy of blocks is now explicitly performed by the application, it can be managed and minimized along with all other communication costs, rather than occurring implicitly with each BLAS call.

4.4.1 Our Contribution

These two broad ideas: using blocked PCA for small-case performance, and exploiting overhead minimizing high-performance microkernels for block operations are the essential ideas needed to improve performance across the entire range of problem sizes. In Section 4.5 we will provide an outline of our actual algorithm, which is essentially a blocked left-looking owner-computes LU factorization with infinite lookahead capabilities and multilevel blocking to improve panel factorization performance. However, these implementation details are less important than the two main ideas we described in this overview. For instance, the observed data access pattern of MKL is that of a right-looking algorithm (we employ left-looking), and yet MKL achieves almost as good performance asymptotically as our approach. Therefore, our main contribution is in highlighting these two key concepts, and in extending the

ATLAS auto-tuning framework to auto-tune these block computational microkernels for our own and other researchers' use.

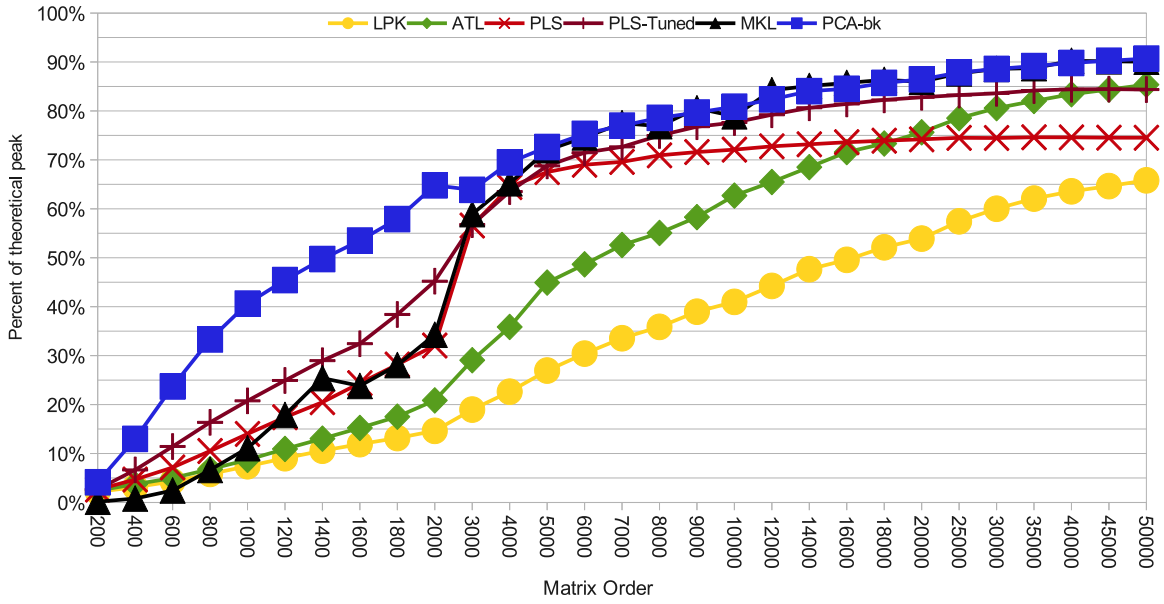
Figure 4.2 shows the performance achieved by our algorithm (labeled as PCA-bk for blocked PCA) against the state of the art on both machines. On the Intel (Figure 4.2a), our implementation (blue squares) dominates all libraries for moderately sized problems, achieving over 50% of peak for $N = 1600$ spread over 12 cores, and essentially ties MKL asymptotically (MKL achieves 90.06% of peak, while we get 90.7%). Figure 4.2b shows the results on the 32-core AMD system, where our algorithm is clearly superior for the entire curve. Note that we plotted Intel's MKL on the AMD machine, but not the reverse; this is because MKL provides the best asymptotic performance (excluding our own) on AMD, but ACML does not provide good performance on Intel.

4.4.1.1 Outline of Remaining Contributions

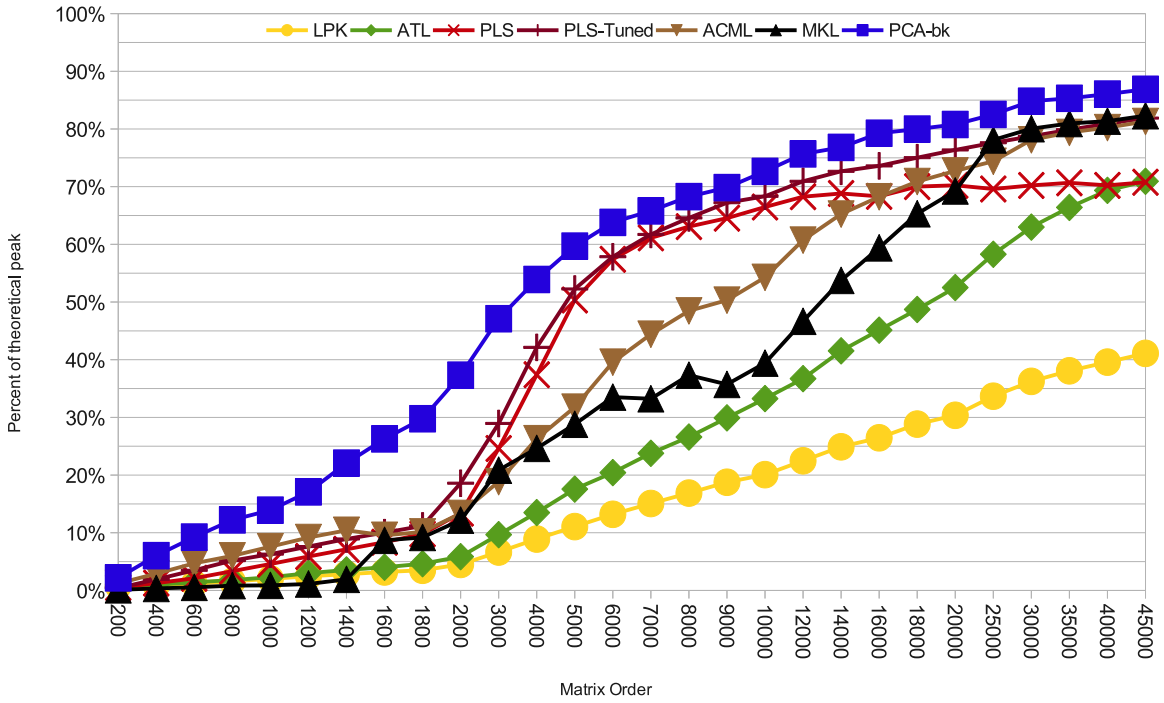
The rest of this paper is organized as follows: Section 4.4.2 presents some implementation details that could effect the sustainability of these results, while Section 4.4.3 outlines a surprising and straightforward method of improving LU performance that we previously (and erroneously) considered unimportant. Section 4.5 then explains the details of our algorithm, while Section 4.6 gives some prioritization advice on the optimizations we implemented. Finally Section 4.7 will describe some future work, with Section 4.8 providing summary and conclusions.

4.4.2 Drawbacks in Our Current Implementation

Since our LU distributes blocks across a 2-D process grid, for every problem size and architecture we must choose values for the number of rows in the process grid (r) and the number of columns (c), under the constraint $r \times c \leq p$, where p is the number of cores. We must also choose a blocking factor (N_b); typically small problems must use a small block factor in order to increase parallelism at the expense of serial performance, while large problems will allow us to expand the block factors in order to saturate serial microkernel performance.



(a)



(b)

Figure 4.2: LU factorization efficiency for netlib lapack (yellow circles), ATLAS (green diamonds), PLASMA (red x), empirically tuned PLASMA (dark red +), ACML (brown, point-down triangles) MKL (black point-up triangles), and our approach (blue squares): (a) For 12-core Intel Xeon E5-2620 and (b) For 32-core AMD Opteron 6128.

N_b is further constrained by essentially two factors: (1) the size of the cache, and (2) the architecture-dependent microkernel details discovered during the ATLAS microkernel tuning step. This still leaves us with a large degree of flexibility in choosing the N_b For this research, we empirically tuned N_b, r, and c using a brute-force search that is probably too expensive to be used routinely. In the future we will need to investigate to what degree we can replace this brute-force search with a model, heuristic, or combination of these with a smarter search. If this is not done well, the smooth climb in performance we achieved here can become more stair-step like, as we transition from a smaller to larger blocking factors at inappropriate times or choose less efficient process grids. There is little doubt that a more sustainable empirical search can be constructed that keeps the overall picture roughly the same, but future research is needed to demonstrate how close we can get to these curves where we have essentially searched the entire optimization space.

In order to improve small-case performance, we do all computation via the owner-computes rule. Since block cyclic distributes the data roughly evenly across cores, if one or more cores experiences sustained unrelated load, the entire algorithm can be slowed down (see Section 4.5 for details; our algorithm is not statically scheduled, but due to the owner-computes rule a heavily loaded core's tasks will eventually get into the critical path of the algorithm). Therefore, if the library is aimed primarily at usage where multiple jobs share cores, it may make sense to enable work stealing, which will fix this issue at the cost of reducing cache reuse.

4.4.3 The Surprising Importance of Parallelizing Row-swap

In the course of this research we measured the impact of various optimizations. The most surprising result of this profiling was how important parallelizing the LAPACK routine DLASWP turned out to be. DLASWP is an LAPACK routine used mainly to swap the rows dictated by LU's pivoting strategy. As such, it is a very simple function, and its par-

allelization requires almost no effort (it took us less than an hour, including debugging and timing).

Since this is an $O(N^2)$ cost that is completely memory hierarchy bound, we did not anticipate noticeable speedup in parallelizing this operation. However, in ATLAS's recursive LU, we got as much as 16% speedup on X12, and almost 18% speedup for O32, *for the entire LU factorization* (i.e. not an 18% speedup in swap, which would not have surprised us as much). This maximum speedup happens in the middle of the data range (for small problems, the row isn't long enough for parallel operation to make much difference, and for very large problems this $O(N^2)$ cost is dominated). However, it appears that as parallel scale is increased, and all other operations are at least partially parallelized by the BLAS, serial DLASWP (DLASWP does not call the BLAS, and so cannot be parallelized at that level) becomes a bottleneck due to some combination of memory hierarchy effects and Amdahl's law. On X12 we saw only a 5% asymptotic speedup at $N = 45,000$, but on the O32 it was still providing 13% speedup over performing the swaps serially. Therefore, we recommend that even those libraries wishing to allow the BLAS to handle the bulk of the optimization investigate parallelizing DLASWP, due to its simplicity and surprising payoff.

4.5 Our Approach in Detail

4.5.1 Understanding Left-looking LU

Our algorithm is based on the left looking variant of LU[2]. The algorithm breaks the matrix up into N_b-wide column panels, as shown in Figure 4.3. These panels are factorized one at a time from left to right, as outlined in Figure 3.1; the algorithm is "left-looking" because to factor column panel i, we read ("look") at only the panels to the "left" ($0 \leq j < i$). The main advantage of left-looking comes when we can contain the majority of the active panel in the (collective) cache; in this case, the repeated writes to the column panel cause little or no bus traffic, and will be smoothly ejected as needed during later steps (unless the entire

[2]Note that all algorithms discussed in this paper are drop-in replacements for LAPACK's DGETRF, and thus do partial pivoting considering all elements within a column.

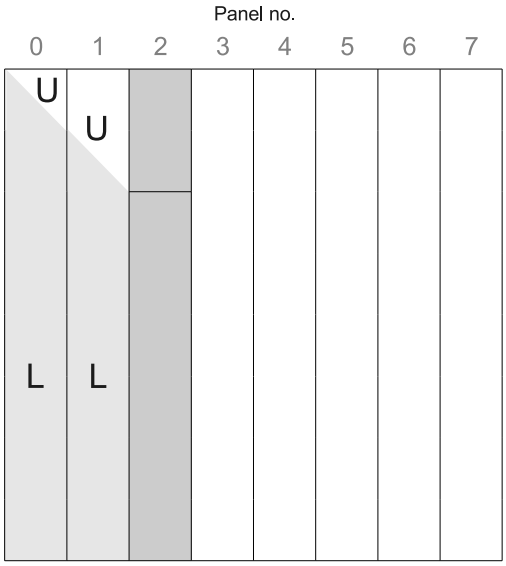


Figure 4.3: Serial blocked LU factorization

matrix fits in the cache, in which case they will only cause bus traffic when the final matrix is written out).

The *panel factorization* (step 2 of Figure 3.1) is just an LU factorization specialized for tall-skinny panel shapes, rather than square matrices. It can be implemented in a variety of ways, including recursion, (multi-level) static blocking, and using an unblocked algorithm. For now, it is enough to know that it will produce an LU factorization for a tall and skinny shaped input matrix.

In Figure 4.3, we assume we have already factored the first two column panels, and panel2 (third panel, dark grey) has become the active panel, which means we will apply updates arising from the first two panels in turn, before factorizing that portion of panel2 that hasn't become U (in this case, the upper 2 blocks become part of U). After the panel factorization is called on the remaining blocks of panel2, the third block from the top will become this panel's diagonal block, with the blocks beneath holding the final (but not fully pivoted) L. Therefore, the first three panels will be completely factorized, and panel3 will become the active panel. This process is outlined more formally in Figure 3.1.

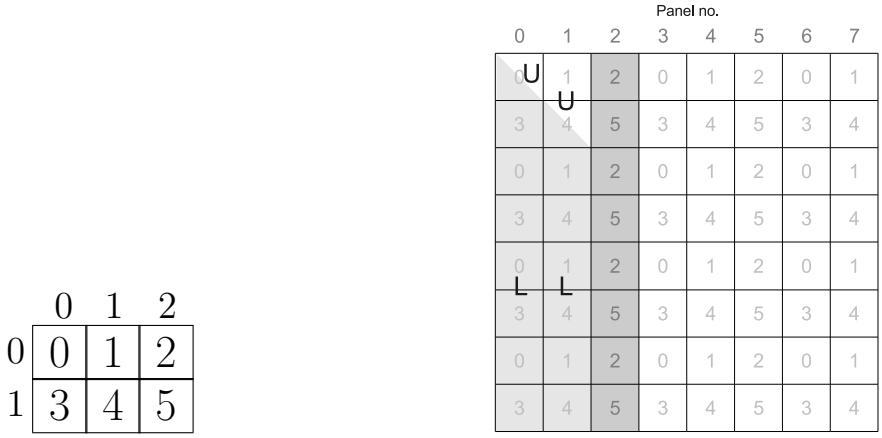
4.5.2 Our Overall Parallel LU Factorization

In order to describe the parallel algorithm, we will need to define some terminology. Anytime we use the word *panel* with no modifier, assume it is one of the column panels shown in Figure 4.3. We distribute the input matrix on a $r \times c$ process grid (here we use process, thread and core interchangeably). Figure 4.4a shows a process grid for $r = 2$ and $c = 3$. Only r processes will work on a panel at a time and we will refer to these r processes as a *pcol*. So, for the 2×3 grid, thread 0 and 3 will be referred as pcol0, thread 1 and 4 as pcol1 and so on. Just as a matrix column or column-panel is owned by a pcol, a given matrix row or row-panel will also be owned by a *prow*. A thread will now have two different views of the matrix: a *global* view that includes the entire input matrix, and a *local* view, that contains only the blocks that belong to this thread due to the block cyclic distribution. In the serial algorithm, the leftmost unfactored column panel was called the *active panel*, but locally, all c pcols will have a leftmost panel, which we will call the *llpan* (locally-leftmost panel). However, only one pcol's *llpan* will be the one that is currently being factorized, and we will refer to this panel as *gcpan* (globally-critical panel).

Figure 4.4b shows the distribution of a matrix for 2×3 process grid. As you can see, all the computations on the *gcpan* (panel 2) would be applied by pcol2; since panels 0 and 1 have been completely factored, pcol0 and pcol1 could be applying these updates to their *llpan* (panel 3 and panel 4, respectively).

The scheduling of parallel work is simple in abstract, but can be complex to understand once the algorithm is heavily optimized. Therefore, Figure 4.4c shows our basic algorithm before optimization, which is much more straightforward. We will now discuss how our optimizations improve this implementation, before summing up our full parallel algorithm.

From Figure 4.4c we see that each pcol is always working on one of their local column panels (when work is available), and so computation is normally occurring in all c column panels simultaneously. This effectively translates to the parallel algorithm providing $c - 1$



(a) (b)

1. *Copy-in*: Copy local part of global matrix to ATLAS storage.
2. **foreach** *llpan*:
   (a) **foreach** panel on the left of *llpan* globally:
      i. Wait until that panel is factorized.
      ii. If in same prow as that panel's diagonal block, apply pivots and solve (TRSM) using that diagonal block to create block of U; if not in prow, wait until U solve complete.
      iii. Using L from factorized panel and U block computed in step ii, update thread's local blocks of *llpan* (GEMM)
   (b) Participate in the panel factorization of *llpan*, which is now the *gcpan*
3. Copy the final result from local storage back to the original matrix
4. Wait for last panel to be factorized
5. Apply pivots to non-diagonal part of L in original matrix

(c)

Figure 4.4: Understanding parallel left-looking LU factorization: (a) 2×3 process grid (pgrid) (b) Block cyclic LU factorization on a 2×3 process grid (c) Straightforward parallel algorithm

lookahead for free. We will discuss the steps of Figure 4.4c in detail and discuss the scheduling optimizations we applied to improve them.

4.5.2.1 Copy-In

This step does not appear in the serial algorithm; its purpose is to bring blocks of the input matrix into the cache and store them in one of ATLAS's internal storage formats. Figure 4.4c shows the simplest approach where each thread starts the algorithm by copying all its local blocks into this storage. The problem is that this will put an initial massive load on the bus, resulting in strongly reduced parallel performance during this stage. Therefore, at the start of our algorithm, each thread will only copy their share of the first panel they will work on. So in Figure 4.4b, pcol0, pcol1 and pcol2 will only copy their share of panel 0, 1 and 2 respectively. Each thread will copy their share of subsequent panels only when they first need to access them (either because the panel has become the *llpan* or is examined due to lookahead). Performing the copy only when the data is needed for computation has two advantages: (1) it will produce less bus contention by spreading bus access throughout lifetime of the algorithm, and (2) since the computations on the copied data will be done right after the copy, it is much more likely we will operate on in-cache data.

4.5.2.2 Updates

When performing the actions of steps 2a(ii) and 2a(iii) of Figure 4.4c, it is necessary to await the factorization of at least the first *gcpan*. When a *gcpan* is factorized, every thread will get a signal indicating that an additional update opportunity is available. Step 2a(ii) is performed by only one thread within the pcol owning the *llpan*, and the other threads need the computed U block for the next step, which means the remaining $r-1$ threads are idle until step 2a(ii) is done (we will discuss removing this idle time later). Note that when U is formed in step 2a(ii), it is the final U of the algorithm. Therefore, in order to spread out the bus utilization of step 3, this block will be copied back to original storage as soon as the thread owning it completes its portion of step 2. Note that one pcol's *llpan* is actually the

gcpan, which is why it is critical to finish the panel (that other pcol's are waiting on) before performing the copy-back of U.

4.5.2.3 Panel Factorization

Once all the updates are applied on a panel, it is by definition the *gcpan*, and threads working on this panel will start working on the panel factorization, which can be done in several ways. In Section 4.5.3.1, we will discuss in detail on how we do the panel factorization, but here we mention a small but important (particularly for small and medium sized problems) optimization that effects the first *gcpan* only. The first panel is special in that all other pcols are idle, and this panel proceeds directly to the panel factorization. Since everyone must wait on this panel to be factorized, we prioritize its operations by having all other threads delay their normal copy-in step until the first panel has been copied to the local storage of pcol0. At that point, pcol0 signals that other threads may begin the copy-in of their *llpan* while pcol0 performs the panel factorization on in-cache data (at least for moderately sized problems). This prioritization allows the critical path to utilize the entire bus bandwidth during the copy-in stage, by artificially idling the rest of the pcols, which will then do their copy-in using shared bandwidth while the critical-path panel factorization is performed by pcol0.

After each panel factorization is performed (i.e. this discussion applies to all panels, not just the first), the pcol will copy back the factorized panel (the U blocks have already been copied back, as previously discussed). The diagonal block, like U, is in its final form, and so when copy-back is complete it is done. The non-diagonal portion of L, however, will be subject to later pivoting, and is therefore not in its final form even though it has been copied back to the input matrix.

4.5.2.4 Copy-back

In our optimized algorithm, this has been merged into prior steps, and no longer exists as a distinct step.

4.5.2.5 Pivoting to the Left

As described in Figure 4.4c, the pivots coming from later panels (to the right of the panel of L under consideration) are not applied until all panels have been factored. This is normally a hard requirement of left-looking algorithms due to data dependencies. However, since we actually do our updates using local copies of L, we are free to pivot those columns that have been copied back to the original storage at any time. Note that as the algorithm factorizes the last c panels, each time a panel is factored one pcol becomes idle. As an easy optimization, these idle pcols instead begin applying the later (to the right) pivots to the portions of L that have been written back to the original storage. For simplicity, we chose to just divide the swap work amongst all cores in this initial implementation, which means that even the last pcol to complete the algorithm will still need to do some L swaps (though the lion's share will have been done while the factorization was still proceeding). Since swap is very demanding on the bus, this will tend to spread out the bus access better than doing all swaps at the same time. In the future, we should introduce completely dynamically scheduled swapping in this step so that it is possible that all swaps of L complete at the same time as the overall factorization.

4.5.3 More Complicated Optimizations for Improving Scaling

Up until now, we have discussed only easily implemented improvements over the straightforward algorithm given in Figure 4.4c. However, further optimization is needed to make this algorithm competitive with the state of the art. Therefore, in this section we explore more complex optimizations that collectively improved our performance by roughly 40% for moderate sized problems, and almost 9% asymptotically on the O32 (the impact was less on the X12 due primarily to its lesser scale). Note that these improvements are sometimes complementary, so applying all of them gives a greater benefit than applying them individually.

4.5.3.1 Panel Factorization

For problems of reasonable size, the panel factorization is dominated by the higher-order computations coming from DGEMM and DTRSM. Therefore, for simplicity our first implementation performed the panel factorization serially. More specifically, all cores in the pcol would copy their local blocks back to the original matrix, where we could call ATLAS's DGETRF serially, after which all cores would copy their local blocks back to local storage. When we profiled our code using the serial factorization, this appeared adequate: the panel factorization time was negligible, and the main idle time it induced that we could see came from having $r-1$ cores in the pcol awaiting the results of the panel factorization (the other pcols could typically remain busy by performing lookahead, as discussed later).

However, we knew that this panel factorization was always in the critical path, and so we later parallelized it even though it seemed the impact might be small. What we found was that even when the panel factorization time was negligible, we could see large overall performance impacts. The reasons for this disproportionate impact are a mixture of critical path optimizations reducing idle time, and improved cache effects. To understand this better, imagine in the middle of the algorithm pcol0 is working on the *gcpan*; when the factorization is serial, the serial core tends to flush its cache doing the panel factorization; meantime other pcols have run out of updates for their *llpan* and begin applying earlier updates to panels to the right (lookahead) to avoid idling. If they go through too many such panels (or one very large panel), they will also flush their cache so that *llpan* will need to reloaded, and *one of the reloads will be in the critical path*, and perhaps fighting for bandwidth with other pcols' reloads.

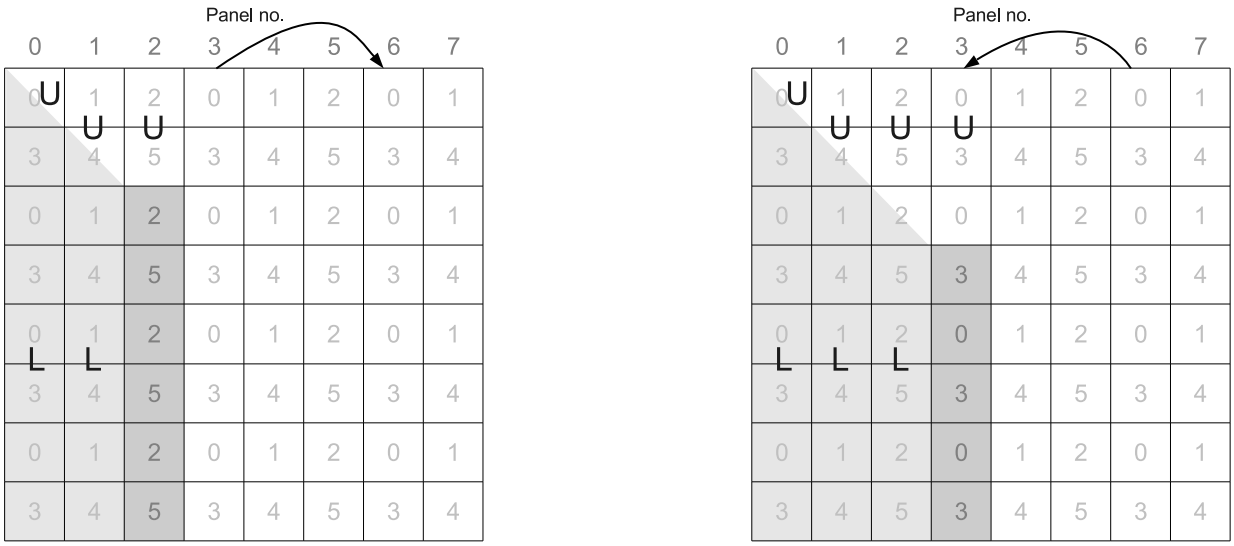
One lesson is that when scheduling is not static, critical path optimizations can provide performance benefits that are not correlated with their raw compute times or even the easily measured idle times of other cores. Commenting out the operation in question can give you a rough idea of the impact of parallelization, but even this will not capture all cache effects.

Therefore, it is highly recommended to at least prototype and time potential optimizations that effect the critical path and therefore the implied scheduling.

For simplicity, our first parallel panel factorization was an unblocked implementation. We later extended this to blocking at two levels. For this work we fixed the inner blocking at 4 (this is the smallest dimension DGEMM that our framework can meaningfully tune), with an outer blocking of 12 (this was dictated by the architectural preferences of the two machines). In general, the performance improvement in going from unblocked to blocked depends on many factors (whether data is in or out of collective cache, the speed of the small-sized GEMM microkernels, etc.), but its advantage should typically grow strongly with the full factorization's blocking factor. If the panel overflows the pcol's collective cache, its advantage should also grow with parallel scale (due to reduced bus contention).

4.5.3.2 Infinite Lookahead

Panel updates always must wait on the factorization of *gcpan*, and so non-*gcpan* pcols will tend to finish updating their *llpan* (using already factored panels of L), and become idle (particularly if the panel factorization is performed serially, as in our initial implementation). To reduce this idle time, we introduced to a form of dynamic scheduling that essentially provides infinite lookahead. The idea is threads will always prioritize their *llpan*, but when they run out of updates for it, they will begin to apply prior updates (already applied to their *llpan*) to panels further to the right. The longer the *gcpan* computations take, the more lookahead will be performed, and the more our "left-looking" algorithm will begin to resemble a "right-looking" variant. Lookahead is illustrated in Figure 4.5a, assuming panels 0 and 1 have already been factored, and panel2 (owned by pcol2) is the *gcpan*. If at this point pcol0 has already applied the first two panel updates to its *llpan* (panel3), it will move to it's next panel (panel 6) to apply panel0's (and possibly panel1's) updates. Having made the decision to update panel6, pcol0 will apply panel0's updates, *and then it will check if the factorization of panel2 is complete*. If so, it leaves panel6 updated only by the first panel,



(a) (b)

Figure 4.5: Infinite lookahead: (a) pcol0 (cores 0 & 3) apply updates to panel6 while panel2 is being factored by pcol2; (b) pcol0 moving back to panel3 after panel factorization complete on panel2

and returns to updating its *llpan*, which is always the panel most in the critical path of the algorithm. Therefore, the transition back to working on panel3 shown in Figure 4.5b can occur after any number of updates are applied to a panel on the right, which requires the algorithm to keep track of the number of updates applied to each panel independently. In scheduling, we never move to a panel to the right until all available updates are done to all panels to the left of that panel (therefore, if we returned to panel3 after applying only one panel to panel6, and we later became idle we would next apply panel1's updates to panel6 *before* applying panel0's updates to a notional panel9 (not shown in figure)).

4.5.3.3 Increased Parallelism of LASWP and TRSM

Note that step 2a(ii) of Figure 4.4c is wholly serial, due to the owner-computes rule. Only one thread within the pcol owns the block that is changed into a block of U using the previous panel's diagonal block, and so all $r-1$ other threads within the pcol are idle while the thread owning this block computes the new portion of U using DLASWP and DTRSM. The swap is an $O(N_b^2)$ memory-bound operation, while the DTRSM is $O(N_b^3)$ compute-bound

operation. Therefore, as long as N_b is small, these costs are trivial, but as N_b rises, this serial operation can become problematic. Recall that we use the owner-computes rule primarily to avoid unnecessary cache pollution, but that is not as important in this case. To understand why, recall that once we finish this step, we will proceed to step 2(iii), which will require every core on the pcol to load all of the produced block of U into its cache anyway, and so the only cache pollution will come from having all cores in the pcol read the diagonal block, rather than just one core.

Therefore, we violate the owner-computes rule and divide the columns of this prospective U block amongst all r cores in the pcol, and each one performs the swaps and updates on its reduced-length rows in parallel. Because we needed a parallel sync after the serial operation was performed anyway, this parallelization requires no additional parallel overhead. However, for small block factors it is nonetheless still the case that overall performance can be slightly lower for this parallel operation than the serial; the reason is that DTRSM is called with fewer right-hand sides, which means the operation gets less cache reuse, and may wind up calling a cleanup case within the microkernel itself. If the reduced serial performance is not made up by the increased scale, this parallelization will result in a slight slowdown. In our implementation, we always parallelize this operation regardless of N_b, judging that the performance loss on very small problems is too minor to justify empirically tuning the crossover between serial and parallel operations (this crossover point is almost wholly determined by architecture and microkernel implementation details that could not reasonably be captured in any a priori model).

4.6 Prioritizing the Complex Optimizations

Of the more complex optimizations surveyed in Section 4.5.3, the most difficult to implement is probably infinite lookahead. This technique is conceptually simple, but in practice it tends to introduce race conditions that are difficult to debug. This optimization is mostly important in the $400 \leq N \leq 2000$ range, where lack of lookahead can drop achieved performance by

as much as 17% for X12 and 9% for O32. Asymptotically, its tendency to occasionally flush the cache makes it very slightly more hindrance than help.

Parallel panel factorization is crucial for asymptotic performance at scale: on O32, the largest problem loses almost 3% performance when the panel factorization is done in serial, though it only loses around 1% if the parallel factorization is unblocked. The blocked panel factorization is not meaningfully more complex to parallelize than the unblocked, but since it is conceptually more complicated, we found it easier to first implement the unblocked, and with the parallelization debugged, write the blocked version. Since not having a blocked panel factorization could conceivably prevent you from using a larger N_b, it is probably worth implementing the blocked parallel panel factorization if you are concerned with asymptotic performance.

Parallelizing the DLASWP and DTRSM by violating the owner-computes rule has its greatest impact for mid-range (say around $N = 8000$, where not implementing it can cost you as much as 5% on both machines); around this size, its low-order term is not yet dominated by DGEMM, but the N_b has gotten large enough for pcol idle-time to be a problem. Asymptotically it gives a performance boost in the 1-1.5% range. This optimization is worth performing due to its straightforward implementation.

4.7 Future Work

See Section 4.4.2 for discussion of the need for developing some combination of model, heuristic and empirical tuning for choosing grid and blocking parameters, and of possibly adding work stealing. In order to apply our approach to many more applications, we need to investigate encapsulating our synchronization and communication in library calls, so that architecture-specific code is not embedded in the application. We also need to measure the impact of using standard mutexes on small-case performance, since the cache-based communication we used here will not work on systems with weakly-ordered cache coherence (ARM and PowerPC both have weakly ordered cache systems). Finally, we must document

our extensions to the ATLAS framework for others' use, and generalize it so any number of applications can ask for particular microkernels without causing namespace collisions. When these investigations are complete, we should apply these techniques to all data precisions of all the factorizations (LU, Cholesky, QR variants).

Longer term, there is limited room for improvement of asymptotic performance, and most appreciable performance gains will likely come from increasing our serial DGEMM efficiency. On the other hand, moderate-sized problems still have room for improvement; most of our current ideas in this area depend on increasingly platform-dependent bus management strategies, which are probably not worth implementing unless we have a compelling use case (i.e. a critical real-life application that requires solving a series of dependent, fixed-sized problems).

4.8 Summary and Conclusion

We have introduced a new approach to LU factorization, and shown that it handles both small and asymptotic problems better than the tiled/BLAS approaches used in PLASMA and FLAME. Further, we have shown that it produces the best known performance on large-scale representative Intel and AMD shared memory architectures across all problem sizes, including vendor-supplied libraries such as MKL and ACML[3]. This approach is widely applicable, and the tuning framework we developed in ATLAS can be used to achieve extremely high parallel performance for any DGEMM-based operation. More broadly, if an empirical microkernel framework like that of ATLAS is constructed, this parallelization approach should be effective for any blockable HPC application.

Along with the critical importance of using the approach advocated in this paper for parallelization, we draw three subsidiary conclusions: (1) In the classic parallelization approach, parallelizing DLASWP is surprising important. This routine's parallelization is extremely

[3]After our paper was published, MKL was updated to improve their small case performance which is now competitive with ours. On newer hardware (Xeon 2670v3) MKL can provide slightly better performance than our approach, which in part is due to our $gemm\mu$ being slower than theirs on this new platform.

straightforward, and so it should be done by all libraries, (2) For tiled libraries like PLASMA it is necessary to vary the blocking factors with problem size, and (3) Profiling critical path operations can massively under-predict the performance impact of optimizing them, due to scheduling differences and cache effects, and we therefore recommend that all possible critical-path optimization should be prototyped and timed, even when their potential impact appears small.

CHAPTER 5

PERFORMANCE OPTIMIZATIONS FOR ARM ARCHITECTURES

In recent years, mobile devices have become an interesting platform for the HPC community, partially due to the use of machine learning in smartphone applications. Hardware manufacturers are also promoting mobile architectures to build large clusters for HPC applications because of their low power usage. ARM is currently the most popular mobile architecture being used for HPC clusters. To continue this trend, hardware manufacturers have released heterogeneous systems (e.g. big.LITTLE), where some cores of the machine are for HPC applications, while the other cores are for simple tasks. The purpose of such heterogeneous architectures is that it can provide a balance between high performance and low power usage depending on the application. In 2015, as a part of an internship, we collaborated with ARM Research Inc. to improve the performance of ATLAS on their newly released 64-bit architecture. We worked on three key areas to improve ATLAS performance on ARM 64-bit architectures: a) optimizing *gemm*μ b) supporting heterogeneous systems and c) providing a reliable performance metric in presence of CPU frequency scaling. Note that the latter two are generic improvements that can be applied to other architectures as well. Section 5.1 provides detail on how we built *gemm*μ for two different ARM 64-bit architectures. In Section 5.2, we will discuss how we can adapt ATLAS to recognize heterogeneous systems and properly tune itself for best performance. In Section 5.3, we present a new approach for measuring performance (vital for ATLAS's auto-tuning step) in presence of CPU frequency scaling.

5.1 Developing *gemm*μ for ARM 64-bit Architectures

In previous chapters, we discussed how ATLAS's *gemm*μ framework works and how we can use that to achieve low-overhead BLAS and LAPACK routines. The microkernels that are exposed by the framework can be either hand-tuned or generated. Usually, hand-tuned microkernels provide better performance than a generated one because the author of a hand-

tuned microkernel needs to know the architectural features and limitations to tune the code for best performance. To understand how a *gemmμ* works, consider the example in Figure 5.1. We have a $B_M \times B_K$ sized block A, $B_K \times B_N$ sized block B and the result C is a block of size $B_M \times B_N$. To minimize the memory access, the data are register blocked: i.e. load some data in the available registers and use them for computations for as long as possible. As shown in Figure 5.1, to compute a $u_m \times u_n$ unroll-block of C (e.g. C_{11}), we stream through an u_m-sized row panel of A (i.e. A_1) and an u_n-sized column panel of B (i.e. B_1). Note that this $u_m \times u_n$ sized unroll-block of C is only accessed once for read and once for write (the theoretical minimum). The values of u_m, u_n, and u_k (the unroll factors in the dimensions M, N, and K, respectively) depend on the number of registers available and pipeline depth of the processor. In this section, we discuss the *gemmμ* we developed using ARMv8 assembly language that are tuned for two ARM 64-bit architectures: Cortex-A57 and Cortex-A53. We used a Juno development board [40] to test and time our microkernels.

5.1.1 Developing *gemmμ* for the Cortex-A57

The Cortex-A57 is a 64-bit ARM architecture, tailored for high performance. It has the following characteristics that we will exploit to develop our *gemmμ*:

1. 32 128-bit sized Advanced SIMD vector floating point registers (i.e. each vector register can hold 2 double-precision or 4 single-precision floating point values).

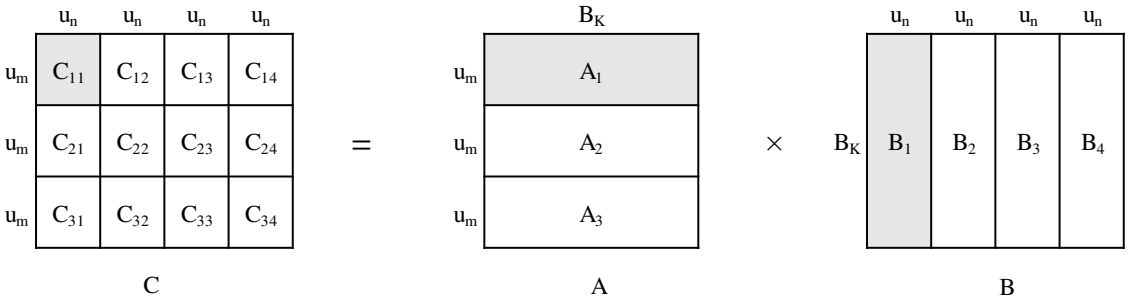


Figure 5.1: An example of *gemmμ* with unroll factors u_m and u_n in M and N dimensions, respectively

2. Peak of 4 double-precision floating point operations per cycle (1 Fused-Multiply-Accumulate also known as FMLA instruction on a vector register).

3. One FMLA and one Vector load (*ldr* or *ldp*) can be issued in the same cycle.

4. Each element of a vector register can be directly accessed in instructions.

5. Supports out-of-order execution of instructions.

Since we want to maximize the reuse of any data that we can load in the vector registers (register-blocking), we want to maximize the unroll factors u_m and u_n. For both ARM architectures discussed in this section, we chose to vectorize the operations in M-dimension. Therefore, for M-vectorized microkernels, u_m is a multiple of $vlen$, the number of elements a vector register can hold. We need $\frac{u_m}{vlen}$ registers to load and reuse elements of A, 1 register for elements of B which we can reuse, and $\frac{u_m \times u_n}{vlen}$ registers to store the result elements of C. Since Cortex-A57 has 32 vector registers and $vlen = 2$ for double precision, the number of registers needed: $\frac{u_m}{2} + 1 + \frac{u_m \times u_n}{2} \leq 32$. For our microkernel, we chose $u_m = 12$, $u_n = 4$ which causes at least 31 registers to be used. To reduce the dependence distance between the B load and the following FMLA (using the load) instructions, we used the spare register instead of reusing the same register for B. For $u_m = 12$ and $u_n = 4$, in each K-loop, we are calculating 48 elements of C that are stored in 24 vector registers. Therefore, in one K-loop iteration, we need 24 FMLA instructions to multiply A and B elements and accumulate to the C registers. Recall that on Cortex-A57, we can issue 1 FMLA instruction per cycle, so each iteration of K-loop needs at least 24 cycles to complete. Since we can issue vector load instructions along with FMLA on the same cycle, we will reuse these 24 cycles to schedule the data load into the vector registers. Note that since this is an M-vectorized microkernel, loads of A elements and stores of C elements can be done using vector load instructions. For B elements, since we can access each element of a vector register directly, we can also use the vector load for B and then access one element at a time. To summarize, we have 24

FMLA instructions, 6 vector loads (12 elements) of A, 2 vector loads (4 elements) of B and some integer arithmetic instructions (loop counter and pointer updates) inside the K-loop. At the end of the K-loop, we store the 24 vector registers i.e. 48 elements of C. Due to the Cortex-A57 supporting out-of-order execution, instruction scheduling did not provide any noticeable performance improvement and was omitted from the final version. For unrolling the K-loop (i.e. u_k), we tried several values. Larger u_k, while reducing the loop overhead of the K-loop and to helping with out-of-order execution, can overburden the instruction cache. From our experiments, $u_k = 6$ provided the best performance on this architecture. For single-precision *gemmμ*, we applied the same techniques with the only change to u_m to be 24 instead of 12 since $vlen = 4$ for single-precision. The final implementations can be found in any ATLAS release after 3.11.37.

5.1.2 Developing *gemmμ* for the Cortex-A53

The Cortex-A53 is another 64-bit ARM architecture which is very popular for its low power usage. This low power usage is the result of several hardware limitations that we need to address to optimize *gemmμ*. The following are the characteristics of Cortex-A53 that we can exploit or must work around to achieve high performance:

1. Like Cortex-A57, it has 32 128-bit Advanced SIMD vector registers.

2. In theory, it can do 4 double-precision or 8 single-precision FLOPs (i.e. 1 FMLA on a vector register) per cycle, but in practice, due to limited number of floating point register ports, FMLA and floating point loads (FP load) cannot be issued on the same cycle. Also, the FP loads cannot be issued on the 4^{th} cycle of each FMLA instruction when it enters the accumulator pipeline of the processor.

3. Vector FP loads takes 2 cycles to issue and no other floating point instruction can be issued during these 2 cycles.

4. Like the Cortex-A57, each element of a vector register can be accessed directly. This facilitates vector load of B-elements.

5. Cortex-A53 does not support out-of-order execution, and so careful scheduling of the instructions is required.

Since we have 32 128-bit vector registers, we can reuse our selection for the unroll factors u_m and u_n to be 12 and 4, respectively, for double-precision microkernel. As before, for unroll factors $u_m = 12$ and $u_n = 4$, we need to load 12 elements of A and 4 elements of B and we have 24 FMLA instructions inside K-loop that need to be issued in 24 cycles. Recall that we cannot issue any FP load instruction in the same cycle with an FMLA instruction. Therefore we need extra cycles to issue the load instructions for the required data. Also recall that we cannot issue a FP load instruction in the 4^{th} cycle of any previously issued FMLA instruction. As a result, the only place we can issue FP load instructions is after every 3 FMLA instructions and for 24 FMLA instructions, we can issue 8 FP load instructions. We will refer to these load instructions as interleaved loads. To issue these 24 FMLA instructions and 8 FP loads, each K-loop iteration requires at least 32 cycles. Note that these floating point loads are only 64-bit loads (1 double-precision element), since 128-bit loads take 2 cycles to issue which causes scheduling problem with the FMLA instructions. With the 64-bit interleaved loads, we can only load 8 of the required 16 elements. For the other 8 elements, as suggested by David Mansell (an ARM Researcher), we can load them with integer instructions which we can issue along with FMLA, and then move the values from integer registers to floating point registers. These moves can be done during the 8 64-bit interleaved load instructions. Using this technique, each iteration of K-loop still requires 32 cycles. Note that out of these 32 cycles, only 24 cycles are performing floating point operations (FLOPs) thus limiting the achievable peak performance to $\frac{24}{32}$ or 75% of the theoretical peak. This instruction scheduling of K-loop is shown in Figure 5.2. Like the *gemm*μ for Cortex-A57, we chose $u_m = 24$ for

time ↓

Cycles	FP Instruction Slot-1	FP Instruction Slot-2	INT Instruction Slot-1
1	FP 64-bit load B0	INT-to-FP convert A0	
2	FMLA		INT 64-bit load A1
3	FMLA		INT 64-bit load A2
4	FMLA		INT 64-bit load A3
5	FP 64-bit load B1	INT-to-FP convert A1	
6	FMLA		INT 64-bit load A4
7	FMLA		INT 64-bit load A5
8	FMLA		INT 64-bit load A6
9	FP 64-bit load B2	INT-to-FP convert A2	
10	FMLA		INT 64-bit load A7
11	FMLA		INT 64-bit load A0
12	FMLA		
13	FP 64-bit load B3	INT-to-FP convert A3	
14	FMLA		
15	FMLA		
16	FMLA		
17	FP 64-bit load A8	INT-to-FP convert A4	
18	FMLA		
19	FMLA		
20	FMLA		
21	FP 64-bit load A9	INT-to-FP convert A5	
22	FMLA		
23	FMLA		
24	FMLA		
25	FP 64-bit load A10	INT-to-FP convert A6	
26	FMLA		
27	FMLA		
28	FMLA		A-pointer update
29	FP 64-bit load A11	INT-to-FP convert A7	
30	FMLA		K-loop Counter
31	FMLA		B-pointer update
32	FMLA		K-loop Condition

Figure 5.2: Instruction scheduling of *gemm*μ K-loop for Cortex-A53 architecture

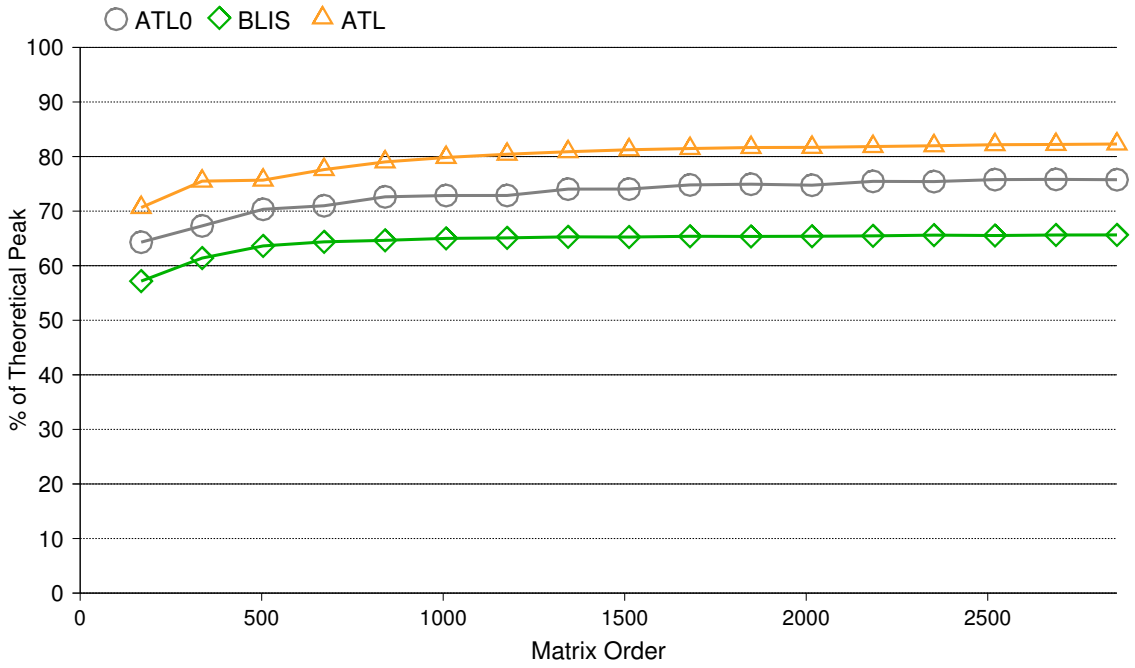
single-precision *gemmμ* for Cortex-A53. As before, the final implementation can be found in any ATLAS release after 3.11.37.

5.1.3 Performance Results

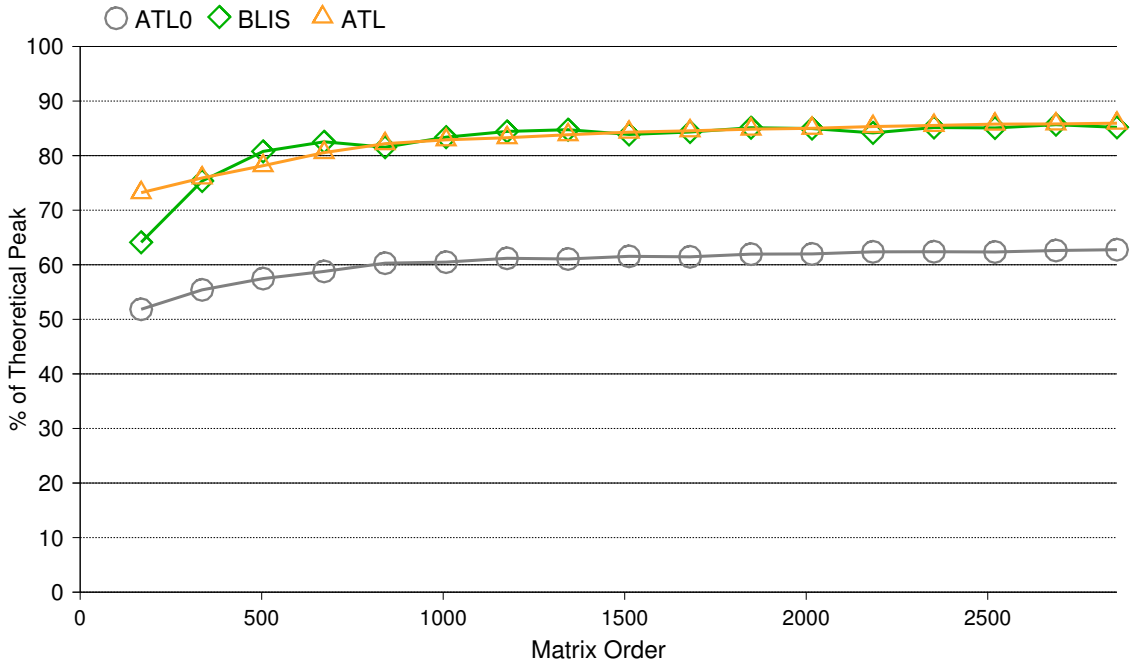
To realize the impact of a *gemmμ*, we need to measure the performance of full GEMM using the newly developed *gemmμ*. For comparison, we used full GEMM performance of the original ATLAS (we refer to this as ATL0) and another library called BLIS [67, 66, 58, 52, 65, 64], developed at the University of Texas at Austin. The details of the experimental methodology is given in Table 5.1. Figure 5.3 shows the performance comparison on Cortex-A57. All the performances are shown as a percentage of the core's theoretical peak performance of 4.4 GFLOPS for double-precision and 8.8 GFLOPS for single-precision. Note that at the time of this research, BLIS was hosted on github and the repository revision used for our timings is given in Table 5.1. As we can see, for the double-precision GEMM (i.e. DGEMM), the performance using our newly developed *gemmμ* (orange upward-triangle) outperforms both the original ATLAS and BLIS for the entire problem range. For single-precision GEMM (i.e. SGEMM), new ATLAS is essentially tied with BLIS but outperforms the original ATLAS by 25%. Figure 5.4 shows the performance comparison on Cortex-A53. Recall that due to architectural limitations of Cortex-A53, the theoretical peak of 3.4 GFLOPS cannot be achieved for GEMM. However, in the results shown in Figure 5.4, we still used 3.4 GFLOPS to normalize the performance. The maximum achievable performance (i.e. 75%) for GEMM is marked

Table 5.1: Experimental methodology for research on ARM 64-bit architectures

Machine	Juno Development Board
CPU	4 Cortex-A53 & 2 Cortex-A57
Cortex-A53 Frequencies (MHz)	450, 575, 700, 775, 850
Cortex-A57 Frequencies (MHz)	450, 625, 800, 950, 1100
Theoretical Peak Performance of A53	3.4 GFLOPS (DP), 6.8 GFLOPS (SP)
Theoretical Peak Performance of A57	4.4 GFLOPS (DP), 8.8 GFLOPS (SP)
OS	Debian 8.1, Kernel 3.15.0-rc8
Compiler	gcc4.9.2
Libraries	ATLAS3.11.34, BLISr483e4d6
Power Measurements	On-board sensors

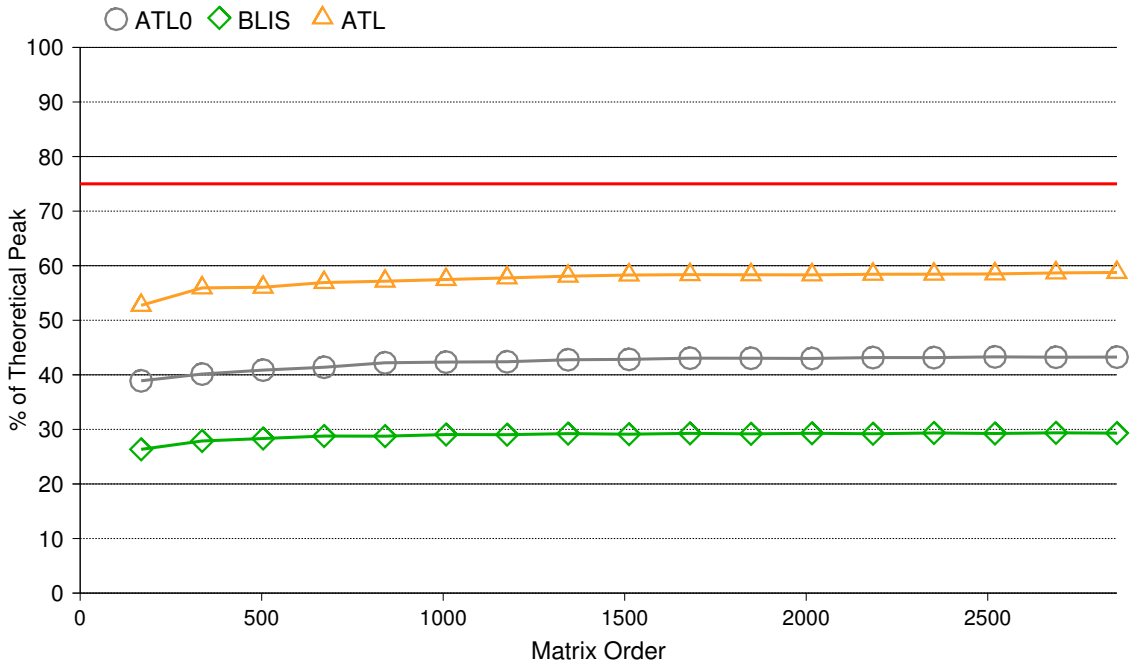


(a)

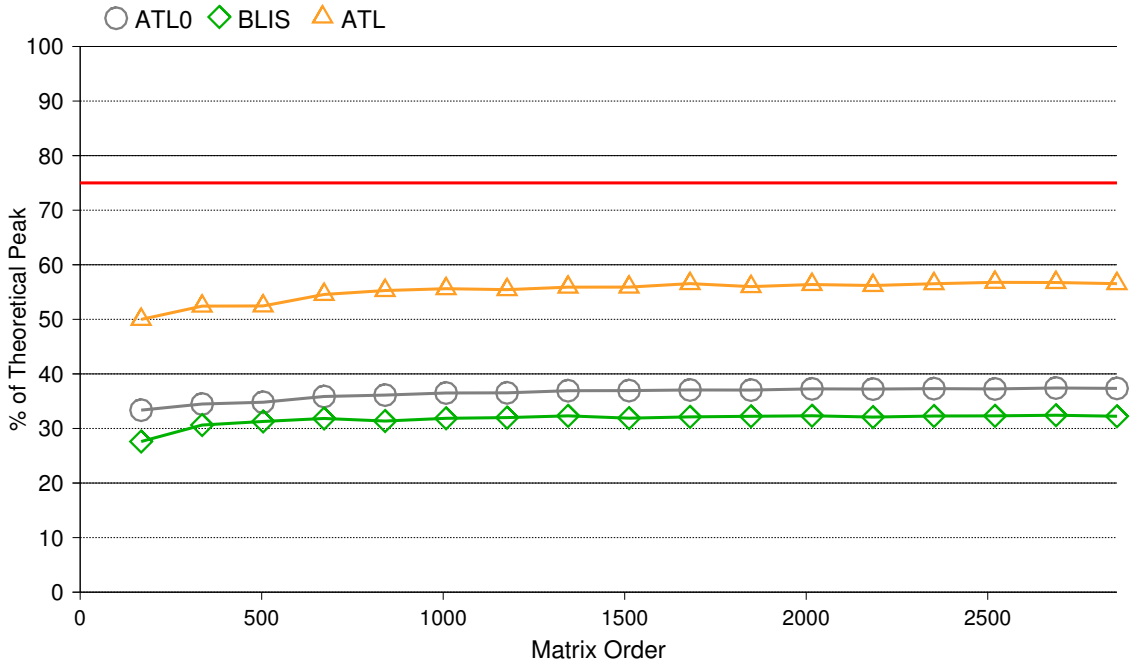


(b)

Figure 5.3: Performance comparison of GEMM for original ATLAS (gray circle), BLIS (green diamond) and new ATLAS (orange upward-triangle) on Cortex-A57 architecture (a) DGEMM and (b) SGEMM



(a)



(b)

Figure 5.4: Performance comparison of GEMM for original ATLAS (gray circle), BLIS (green diamond) and new ATLAS (orange upward-triangle) on Cortex-A53 architecture (achievable peak is shown with the solid red line) (a) DGEMM and (b) SGEMM

with a solid red line. Due to the specific optimizations that can overcome the architectural limitations of Cortex-A53, the performance of GEMM (both DGEMM and SGEMM) using our *gemmμ* outperforms both the original ATLAS and BLIS by up to 35%.

5.2 Adapting ATLAS for Heterogeneous Architectures

As mentioned earlier, ARM architectures like Cortex-A53 and Cortex-A57 are used to build systems with heterogeneous architecture like the Juno board mentioned in Section 5.1. The purpose of such systems is to facilitate applications to utilize the proper resources based on their requirements. For example, a video game that requires high performance can use Cortex-A57, where background jobs like checking for new emails can use Cortex-A53 to minimize energy consumption. In a Juno board, there are 4 Cortex-A53 cores and 2 Cortex-A57 cores. If some applications require all the cores to get the best performance possible, they need to run different versions of codes on different architectures that are specifically tuned for that architecture. At the time of this research, ATLAS did not support tuning for heterogeneous systems. In this section, we discuss how the lack of explicit support for heterogeneous systems can cause $12 - 18\%$ loss of performance for double-precision parallel GEMM and how we can adapt ATLAS's auto-tuning process to achieve the full parallel performance out of a heterogeneous system.

5.2.1 Motivation

In Section 5.1, we implemented highly optimized *gemmμ* for Cortex-A53 and Cortex-A57. The performance results in Figure 5.3 and 5.4 showed that the double-precision *gemmμ* for Cortex-A53 achieves about 2.0 GFLOPS and Cortex-A57 achieves about 3.6 GFLOPS for asymptotic problem sizes. A perfectly-scaled parallel GEMM on a Juno board, where we have 4 Cortex-A53 cores and 2 Cortex-A57 cores, should achieve performance of roughly $4 \times 2.0 + 2 \times 3.6 = 15.2$ GFLOPS. The actual performance of parallel DGEMM is shown in Figure 5.5. The performance shown is normalized to this perfectly-scaled performance of 15.2 GFLOPS. As we can see, even for an asymptotic sized problem, the parallel DGEMM

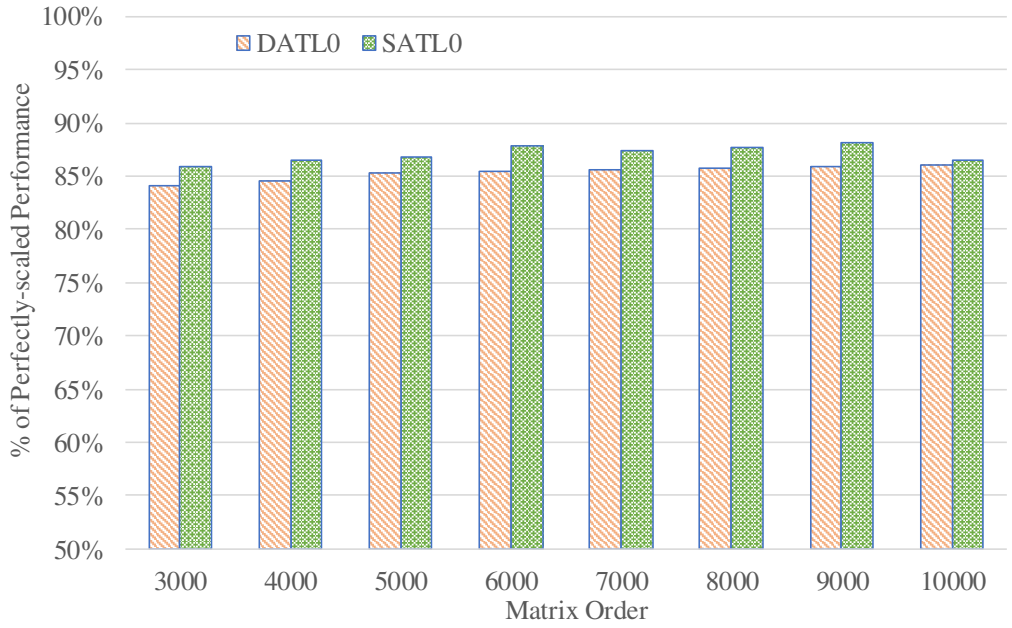


Figure 5.5: Performance of parallel GEMM Using our *gemmμ* on Juno board with unmodified ATLAS: (a) DGEMM (orange right-downward diagonal patterned bars i.e. DATL0) (b) SGEMM (green dot-patterned bars i.e. SATL0)

is losing about 12% of the achievable performance. Note that parallel GEMM in ATLAS is dynamically scheduled, hence this performance loss is not primarily caused by workload imbalance, but is rather a result of running a significantly bad *gemmμ* on the Cortex-A57 cores. Due to the lack of support for heterogeneous systems, ATLAS times *gemmμ* on all available cores at once and chooses the one providing the best average performance. In this case, the chosen *gemmμ* was the one for the Cortex-A53 which achieves significantly lower performance on the Cortex-A57 than the one specialized for the Cortex-A57. As we can see from Figure 5.5, similar performance loss occurs for parallel SGEMM as well.

5.2.2 Modifications in ATLAS

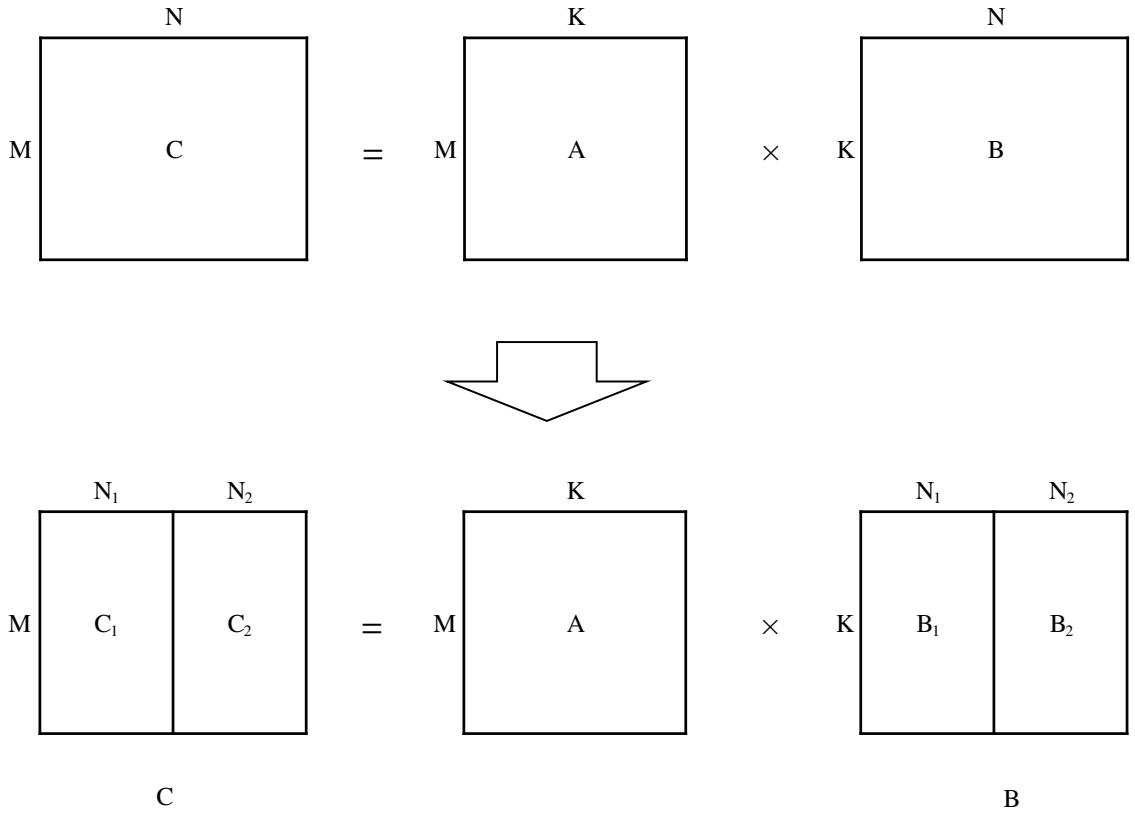
ATLAS installation consists of 3 major steps: a) Configuration b) Auto-tuning and c) Compilation. In the configuration step, ATLAS tries to detect architectural features that it can exploit to achieve high performance. In the auto-tuning step, it compares performance of various microkernels that are potentially high performing on the system and finds the best one for each BLAS routine. In the compilation step, as the name suggests, it compiles the

best microkernels it found for each BLAS routine and builds a complete library for users. To adapt ATLAS to recognize heterogeneous architectures, we need to modify all these steps accordingly.

In the configuration step, the only change we need is to have a mechanism for detecting heterogeneous architectures and save the architecture information (if detected). The ideal way would be to use CPU identification number that are usually provided by the ISA (Instruction Set Architecture). However for the ARM architecture, the CPU identification information is not available for user applications (at the time of this research). As a result, we used the maximum available clock speed of a core to identify different architectures. On the Juno board, the maximum clock speed for Cortex-A53 cores and for Cortex-A57 cores are 850 MHz and 1100 MHz, respectively. Using this information, we categorize the core ranks into different clusters. For Juno, we have two clusters: one for Cortex-A53 cores (core ranks: 0, 1, 2 and 3) and one for Cortex-A57 cores (core ranks: 4 and 5).

In the auto-tuning step, instead of tuning on all cores at once, we use the cluster information from the configuration step and tune the microkernels on one cluster at a time. Note that with this approach, we have tuning steps that are repeated for each cluster. For this research, we only changed the tuning phase of *gemm*μ so that ATLAS can use both of the optimized *gemm*μ that we developed in Section 5.1.

In the compilation step, we compile the best *gemm*μ and the related files (e.g. copy microkernels) for each cluster using cluster-specific generated names. Note that to utilize the cluster-specific microkernels and other tuning parameters (e.g. block size), we also need to adapt the BLAS routines. As before, for this research, we only adapted the parallel GEMM implementation. For simplicity, we added one layer above the current implementation of parallel GEMM: trivially partitioning the original problem among the clusters based on their relative performance and then calling the current dynamically scheduled implementation on the partitioned sub-problems. Figure 5.6a shows an example of partitioning the input prob-



(a)

```
Hetero_GEMM(M, N, K, A, B, C)
{
   N1 = perf_ratio[cluster1] * N;
   N2 = N - N1;
   ATL_PGEMM(M, N1, K, A, B1, C1, cluster1);
   ATL_PGEMM(M, N2, K, A, B2, C2, cluster2);
}
```

(b)

Figure 5.6: Basic idea of parallel GEMM for heterogeneous systems with two Clusters: (a) partitioning the input (b) basic steps for the implementation

lem for two clusters. Figure 5.6b shows the basic steps for our new parallel GEMM for the given example. The value of N_1 (as shown in the figure) depends on the ratio of a cluster's performance and the system's overall performance. For Juno, performance ratio of cluster 0 (Cortex-A53) would be $\frac{4\times2.0}{(4\times2.0)+(2\times3.6)} = 52.63\%$. Using this ratio, the input problem is partitioned into two sub-problems. Each sub-problem is given to the corresponding cluster of the machine. ATLAS's parallel GEMM needs to be modified so that it runs only on a specified cluster instead of the whole system. Note that invoking this cluster-specific parallel GEMM needs to be done in parallel so that all clusters become active at the same time. Also note that we divided the N-dimension of the input so that no communication (result accumulation) is needed among different clusters. This approach may not be the most efficient due to static partitioning of the problem among clusters. A more efficient approach would be to adapt the dynamically scheduled implementation so that each core uses its own optimal code for its share of the computations. In future research, we can investigate whether the performance benefit of the dynamically scheduled approach is worth the added complexity in the implementation.

5.2.3 Performance Results

As before, we used the Juno development board to install, test and time our modified ATLAS. Since Juno has two clusters (Cortex-A53 and Cortex-A57), the input of a parallel GEMM is divided into two sub-problems as shown in Figure 5.6a using the performance ratio of these two clusters. Then each sub-problem is given to ATLAS's cluster-specific dynamically scheduled parallel GEMM. The performance of the modified parallel GEMM implementation is measured using ATLAS's own unmodified timing framework. Figure 5.7 compares the performance of the modified parallel DGEMM of our adapted ATLAS with the unmodified ATLAS. We can see that even with our simple and less-efficient approach we can achieve up to 98% of the perfectly-scaled achievable performance for the parallel DGEMM. Similar improvement is shown in Figure 5.8 for our modified parallel SGEMM.

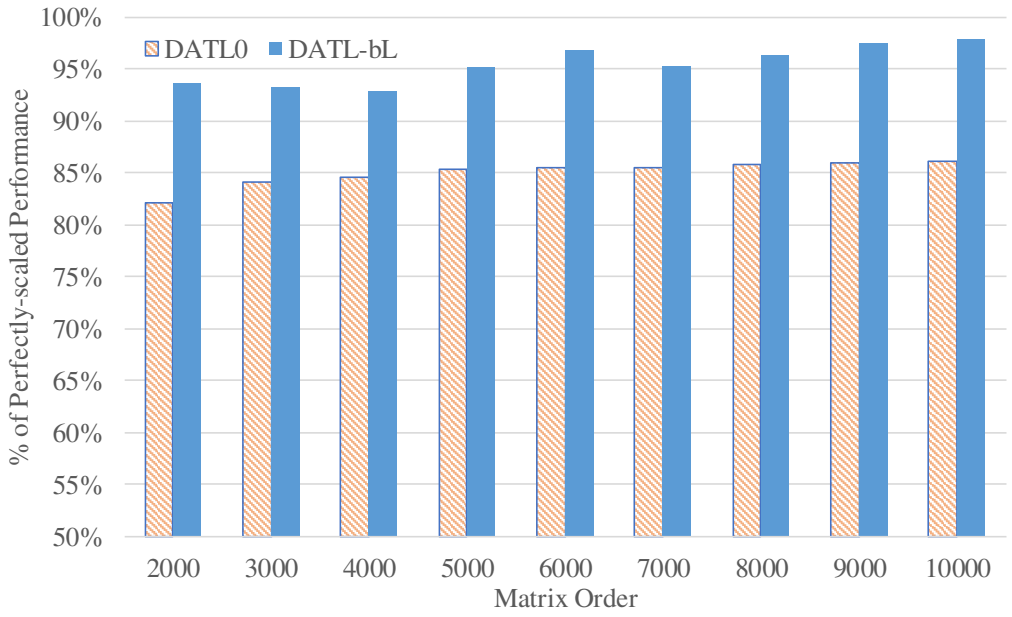


Figure 5.7: Performance comparison of parallel DGEMM using our *gemmμ* on Juno board: unadapted ATLAS (DATL0) and adapted ATLAS (DATL-bL)

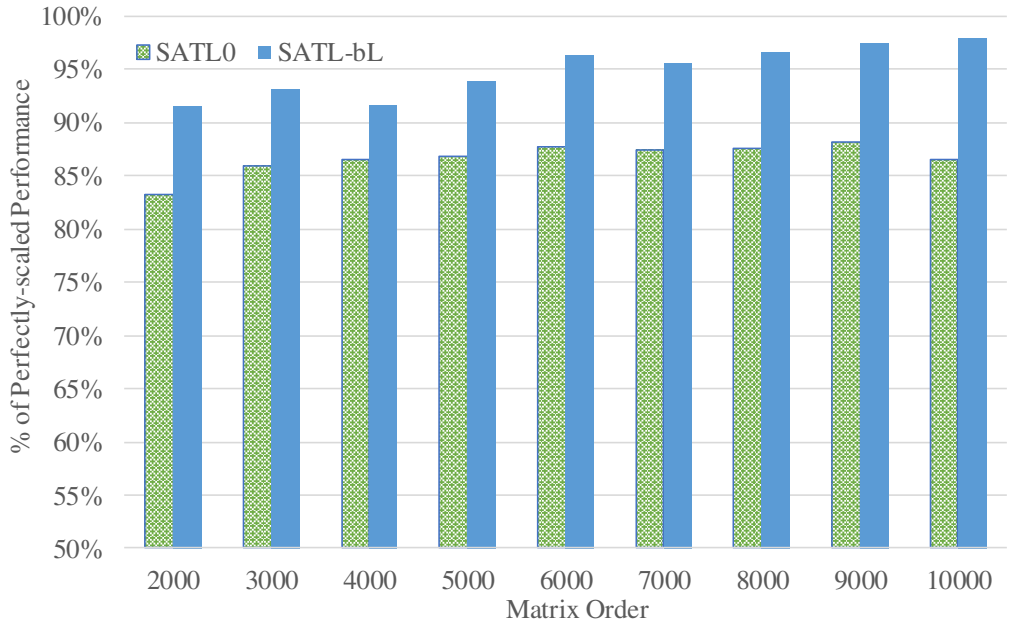


Figure 5.8: Performance comparison of parallel SGEMM using our *gemmμ* on Juno board: unadapted ATLAS (SATL0) and adapted ATLAS (SATL-bL)

5.3 Reliable Performance Auto-tuning in Presence of DVFS

This section is an extended version of a poster published at SC15 [36][1]. In an era where exascale systems are imminent, maintaining a power budget for such systems is one of the most critical problem to overcome. Along with much research on balancing performance and power, Dynamic Voltage and Frequency Scaling (DVFS) is being used extensively to save idle-time CPU power consumption. The idea of DVFS is to run the CPUs at the lowest speed at idle times (i.e. low CPU utilization) and jump to a higher speed when CPU utilization becomes high. The drawback is that the inherent random behavior of DVFS makes walltime unreliable to be used as a performance metric, i.e. using walltime in presence of DVFS leads to random performance from libraries (e.g. ATLAS) that rely on machine-specific auto-tuning of several characteristics for the best performance. In this work:

1. We showed that a sub-optimal selection (not the worst case) of $gemm\mu$ and block size during auto-tuning can cause ATLAS to lose up to 40% of DGEMM performance.

2. We presented a more reliable performance metric in the presence of DVFS that can estimate similar performance as a no-DVFS install and thus facilitating proper auto-tuning.

5.3.1 Introduction and Motivation

To get the best performance from a single library (e.g. ATLAS [75]) or the best performing code from an iterative compiler (e.g. iFKO [77, 68, 61]) for a vast range of different architectures, machine-specific auto-tuning is a state-of-the-art technique [73]. The auto-tuning phase of ATLAS (Automatically Tuned Linear Algebra Software) tries to find the best $gemm\mu$, block size, etc. for that specific machine to build its optimized GEMM. For such tuning, ATLAS requires a reliable performance metric (i.e. walltime) to compare and

[1]This section previously appeared as [Md Rakib Hasan, Eric Van Hensbergen and Wade Walker, The International Conference for High Performance Computing, Networking, Storage and Analysis, published by Association for Computing Machinery (ACM). See the copyright form in Appendix C.

make the best decision. In presence of DVFS, where a *gemmμ* could run at any frequency or a combination of any number of different frequencies, walltime doesn't necessarily reflect the actual performance. As a result, the final GEMM can perform anywhere from the worst case to the best case. In this research, we are trying to enable proper auto-tuning in presence of DVFS so that we can still get the best performance. In this section, we show how DVFS can impact the effectiveness of auto-tuning and in Section 5.3.2, we present our proposed approach to properly auto-tune in presence of DVFS. The behavior of walltime under DVFS is seemingly random for two main reason:

1. Due to short-time executions during auto-tuning, the warm-up time from low to high frequency may be significant.

2. Thermal throttling can cause the frequency to go from high to low and then low to high again depending on the thermal state of the CPU.

In order to see how poor the performance of an auto-tuned library can get due to a bad auto-tuning step in the presence of DVFS and to make a repeatable comparison with our proposed solution, we purposefully simulated two scenarios for ATLAS auto-tuning where:

1. A sub-optimal (second-best) *gemmμ* is chosen (T_K): when the known best *gemmμ* is being timed, the CPU frequency is set to the lowest possible value. This ensures that the best *gemmμ* is measured as poor performing and thus never gets selected.

2. a sub-optimal (inefficient use of cache) block size is chosen (T_B): during timing for all block sizes larger than some pre-selected inefficient block size, the CPU frequency is set to the lowest possible value so that the larger block factors are measured as poor performing and thus do not get selected.

We separately built ATLAS under these two scenarios while using only walltime as the performance metric on a Juno development board (for Cortex-A57 cluster). Our experimental

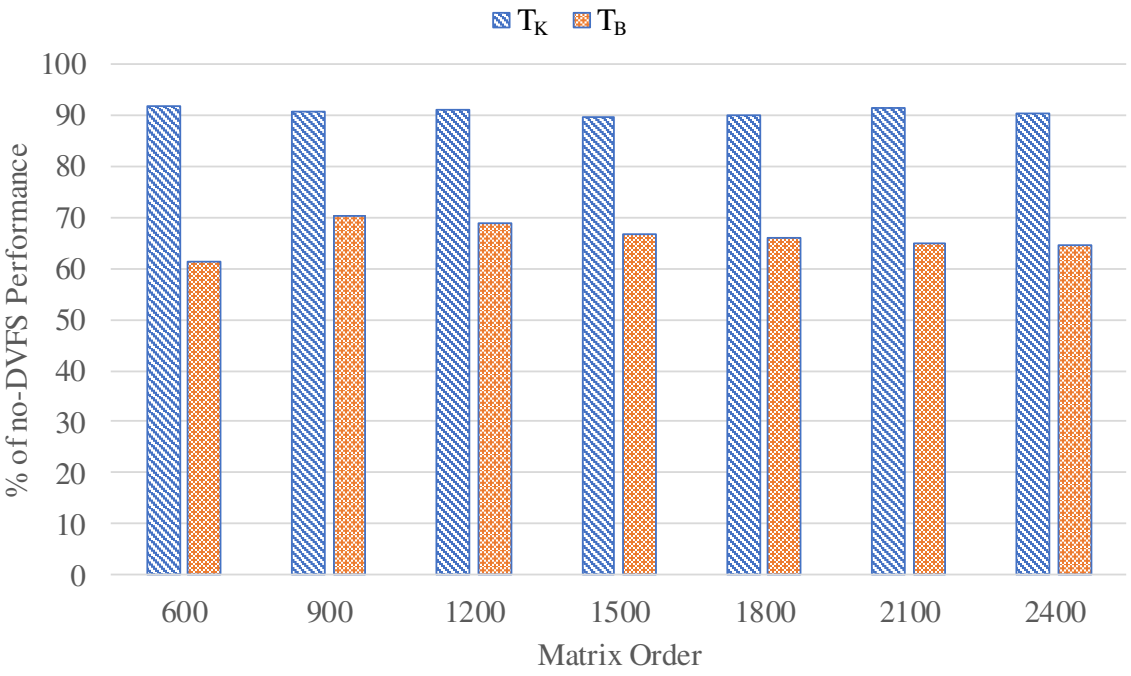


Figure 5.9: Performance of parallel DGEMM on A57 cluster after ATLAS installations with simulated DVFS and using walltime as the performance metric: sub-optimal *gemm*μ (T_K) and sub-optimal block size (T_B).

methodology was shown before in Table 5.1. Figure 5.9 shows the performance of parallel DGEMM (normalized to no-DVFS performance) from the two different ATLAS installations described above. Note that this parallel DGEMM is only running on Cortex-A57 cluster. For repeatable comparison, all these DGEMM timings were done with DVFS turned-off after installation. For sub-optimal *gemm*μ selection (T_K), there is about 10% performance loss and for a fairly inefficient block size selection (T_B), the performance loss is about 40%. Note that with true DVFS, the performance loss can be far worse or none at all. Our goal, ideally, is to ensure no performance loss.

5.3.2 Our Approach

The ideal solution for the performance metric in presence of DVFS would be for the hardware vendors to provide some way for the users to determine the average frequency between two certain points in time. At the time of this research, this information was not available on any machines that we knew of. Therefore, our research was focused on finding a solution, even an approximate one, with the information we could collect at the time. With the

growing concern for power consumption by a machine, hardware manufacturers are including sensors/mechanisms to measure energy consumption by an application. Our primary goal for this research, was to use energy consumption in some way so that we can reliably measure performance. In Section 5.3.2.1, we discuss our initial approach that tries to utilize the overall energy consumption, with and without combining with walltime. In Section 5.3.2.2, we discuss our final approach (still requires some prior knowledge of the system) which we can reliably use to measure performance.

5.3.2.1 Using Energy Consumption

Our initial approach involved using the overall energy consumption as the performance metric with the idea that a slower code running longer will consume more energy. However this idea only works when the DVFS is turned off. In presence of DVFS, if a slow *gemm*μ runs at low frequency and a fast *gemm*μ runs at a higher frequency, the slower *gemm*μ is deemed as the faster one. This behavior is dictated by the non-linear relationship between the CPU frequency and the energy consumption at that frequency. To better understand this, consider the chart shown on Figure 5.10. It shows the amount of execution time (walltime) and the total energy consumption of a parallel DGEMM on Cortex-A57 cluster of Juno at different frequencies. For all frequencies, a fixed size problem (i.e. $M = N = K = 1200$) is analyzed. Note that at the lowest frequency (i.e. 450 MHz), though the execution time is almost twice than the execution time at the highest frequency (i.e. 1100 MHz), the total energy consumption at 450 MHz is still less than the energy consumption at 1100 MHz. As a result, if the total energy consumption is used as the performance metric, a *gemm*μ running at the lowest frequency will always tend to be selected. A revised idea is to combine energy consumption with walltime to penalize slower *gemm*μ with their long execution time. In this case, we use a simple combination of the product of time and the total energy consumption as the performance metric. Figure 5.11 shows the performance of parallel DGEMM for our two simulated DVFS installations. For both of the installations, we used the metric $M = T \times E$

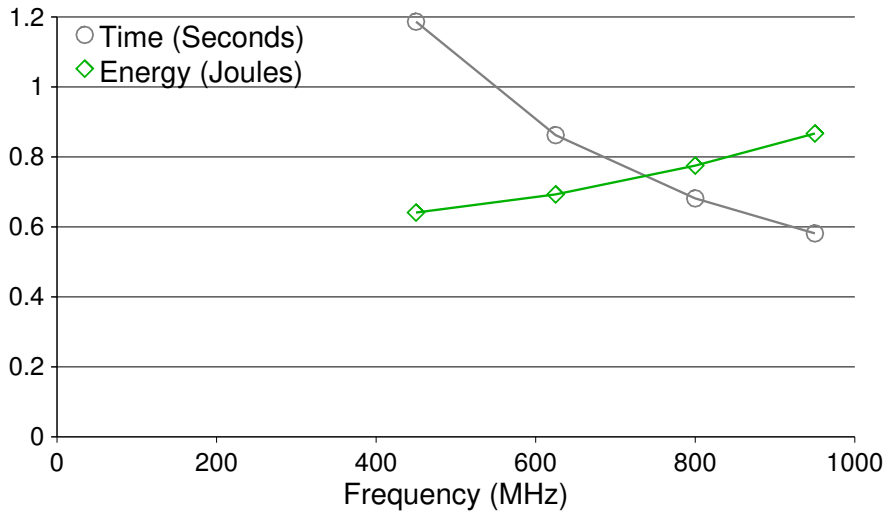


Figure 5.10: Time (gray circle) and total energy consumption (green diamond) of a parallel DGEMM at different frequencies on Cortex-A57 cluster for a square input with $M = N = K = 1200$.

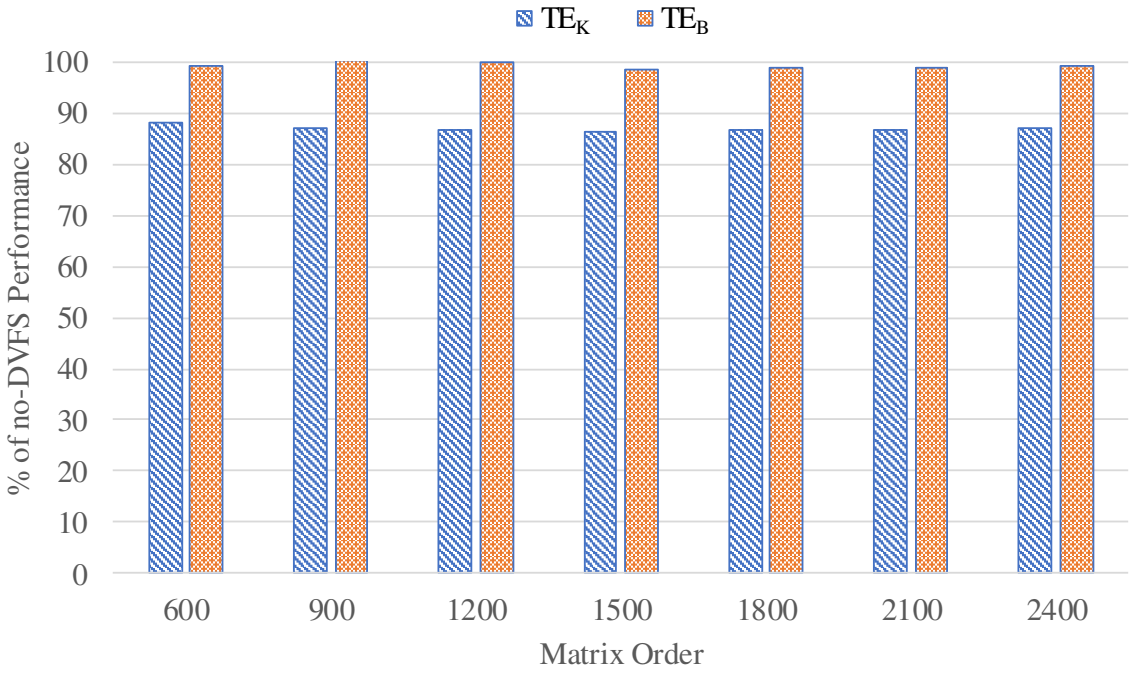


Figure 5.11: Performance of parallel DGEMM on A57 cluster after ATLAS installations with simulated DVFS and using the product of walltime and total energy consumption as the performance metric: sub-optimal *gemm*μ (T_K) and sub-optimal block size (T_B).

to compare performance, where T is the measured walltime and E is the measured total energy consumption. As before, the GEMM performance was measured with DVFS turned off. As we can see, for the inefficient block size selection case (TE_B), there is almost no performance loss, which is what we want. However, for the suboptimal *gemm*μ selection case (TE_K), we still have a performance loss of around 12%. These results indicate that the performance metric $M = T \times E$ provides low accuracy: if the performance difference is significant between two *gemm*μ, it can detect the better *gemm*μ but fails otherwise. In Section 5.3.2.2, we propose our final approach that can provide no performance loss for both of our simulated DVFS cases.

5.3.2.2 Using the Average Power Consumption

As we explored different energy characteristics of the system, we realized that at a certain frequency, the average power consumption is constant. The average power consumption is defined as the energy consumption (joules) per unit time (seconds). Figure 5.12 shows the power consumption of parallel DGEMM at different frequencies on Cortex-A57 cluster with DVFS turned off for a problem of size $M = N = K = 1200$. Figure 5.13 shows the same but for a problem of size $M = N = K = 2400$. Though the execution time for this two problems vary by factor of 10 at a certain frequency, we can see the average power consumption is almost constant for both cases. This is also true for Cortex-A53 cluster as shown in Figure 5.14 and 5.15. Since the average power consumption seems to be consistent at a certain frequency (no matter how long the execution time is), our idea is to estimate the average frequency (f_a) for the entire execution time from the average power consumption (P).

$$\text{Estimated average frequency, } f_a = F(P) \tag{5.1}$$

For this research, the estimating function (F) is formed using the relationship between frequency and average power consumption as shown in Figure 5.12, 5.13, 5.14 and 5.15. Since some hardware vendors already provide similar information (on a system level, not

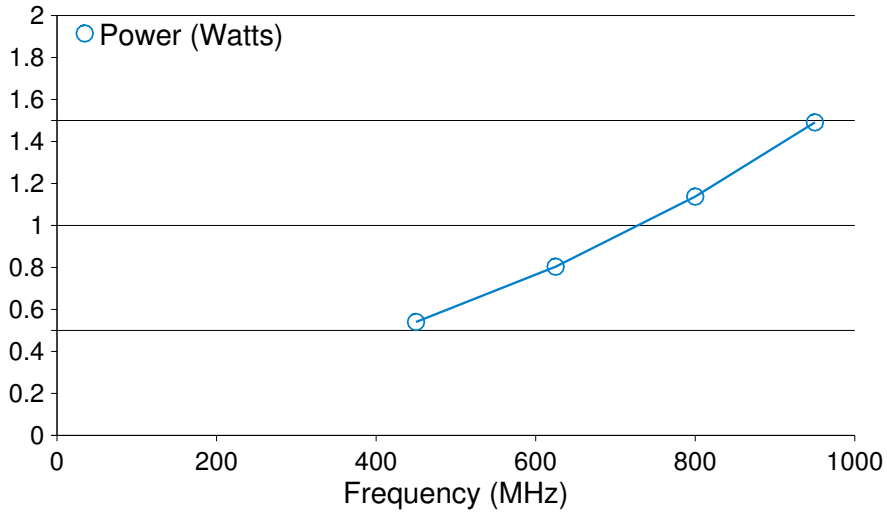


Figure 5.12: Average power consumption of a parallel DGEMM at different frequencies on Cortex-A57 cluster for a square input with $M = N = K = 1200$.

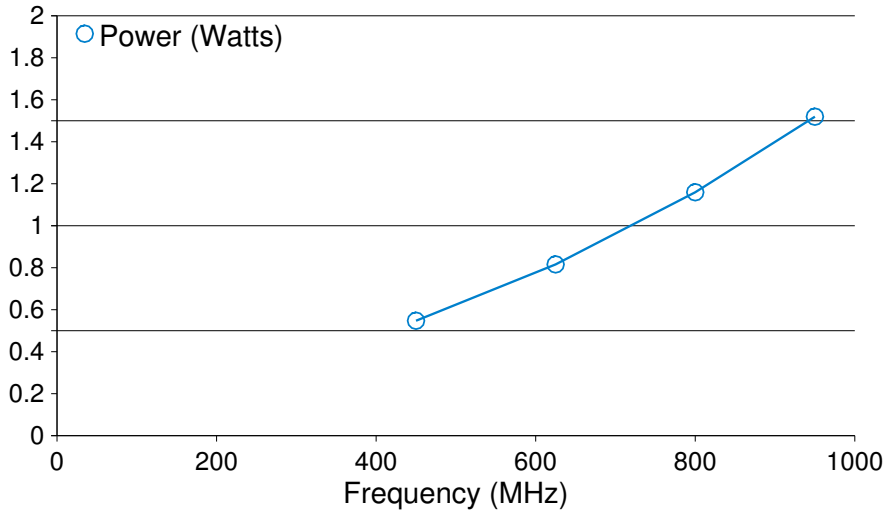


Figure 5.13: Average power consumption of a parallel DGEMM at different frequencies on Cortex-A57 cluster for a square input with $M = N = K = 2400$.

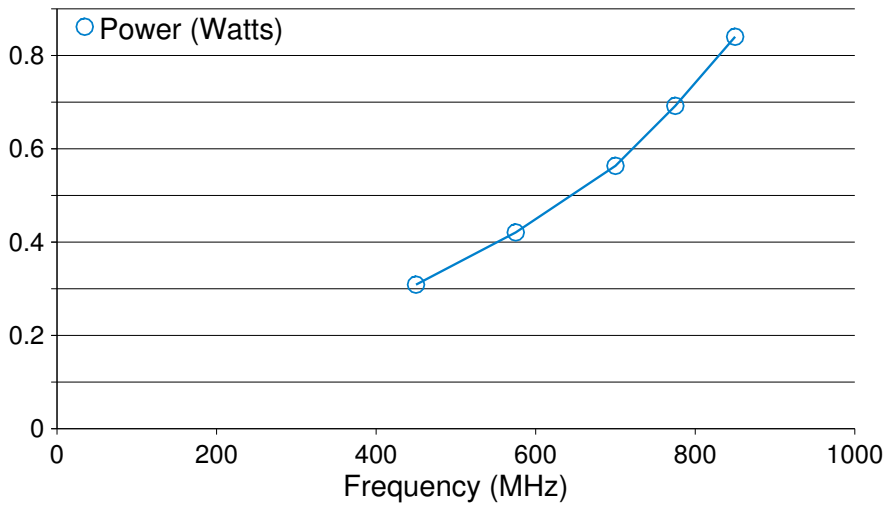


Figure 5.14: Average power consumption of a parallel DGEMM at different frequencies on Cortex-A53 cluster for a square input with $M = N = K = 1200$.

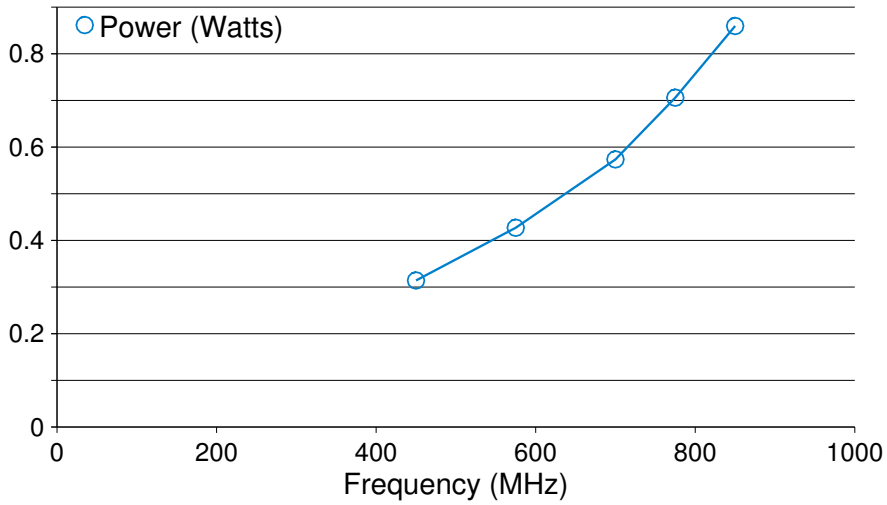


Figure 5.15: Average power consumption of a parallel DGEMM at different frequencies on Cortex-A53 cluster for a square input with $M = N = K = 2400$.

the processor level), we suggest that hardware vendors to provide such specification so that the relationship between a processor's frequency and power consumption is known without any empirical time or energy measurements. This relationship, as we saw in Figure 5.12, 5.13, 5.14 and 5.15, is non-linear but we also explored a linear approximation to compare the results. Using this computed average frequency (i.e. $f_a = F(P)$), we scale the measured walltime (T) and use the scaled walltime (T_s) as the performance metric.

$$\text{Performance metric}, M = T_s = \frac{f_a \times T}{f_b} \tag{5.2}$$

where T_s represents the estimated walltime at frequency f_b.

Figure 5.16 shows the performance of parallel DGEMM on the A57 cluster after simultaneously enforcing both simulated DVFS scenarios of sub-optimal $gemm\mu$ and sub-optimal block size selection during the auto-tuning step of ATLAS. For S_P, we empirically computed the power consumption at each available frequency and formed a polynomial estimator of the average frequency. For S_L, we used the power consumption at the lowest and the highest frequency to form a linear estimator. On Juno, Figure 5.16 shows that the performance of both solutions using non-linear (S_P) and linear (S_L) estimator are essentially the same as no-DVFS, which may not be true for other architectures.

5.3.3 Conclusions

DVFS is becoming very popular for saving idle-time power consumption but due to making walltime useless as a performance metric, it is still not being adopted by researchers focusing entirely on performance. In this work, we proposed two techniques: (1) one that combines measured walltime and energy consumption to provide a low-accuracy performance metric without needing any prior knowledge of the system behavior and (2) one that uses average power consumption to estimate the true walltime at a certain frequency (requires prior knowledge on the relationship between frequency and power consumption possibly from hardware vendors) that can be used as a reliable performance metric in presence of DVFS. The relationship between frequency and power consumption was empirically determined for

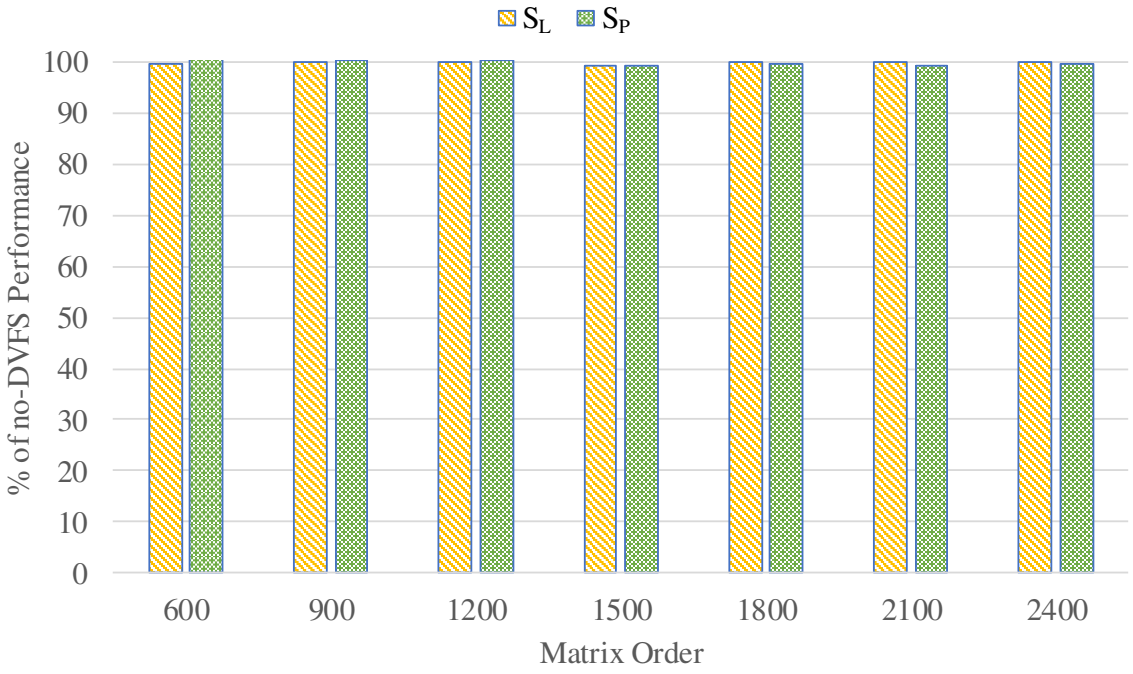


Figure 5.16: Performance of parallel DGEMM on A57 cluster after ATLAS installation with simulated DVFS and using the scaled walltime as the performance metric: non-linear estimator (S_P) and linear estimator (S_L)

this research but it can be manually specified by the system vendors in future (for a linear approximation). The next steps for this research would be to verify the approach on other architectures that do not have on-board power sensors (e.g. using RAPL [42, 41, 55, 57] on Intel architectures). Also, simultaneous CPU and memory DVFS is also being considered as an energy saving technique while maintaining high performance [19, 62]. We need to explore the effect of such combinations (not yet supported by OSes) during auto-tuning and therefore its impact on overall performance.

5.4 Summary and Future Research

In this chapter, we have improved the performance of ATLAS in three different areas:

1. We developed two *gemm*μ for two ARM-64bit architectures which improves the performance ATLAS's DGEMM and SGEMM by up to 12% and 42%, respectively on the Cortex-A57 and by up to 39% and 53%, respectively on the Cortex-A53.

2. We adapted ATLAS to recognize and autotune for heterogeneous architecture systems that improved ATLAS parallel DGEMM and SGEMM performance by up to 13%.

3. We presented a new performance metric that can be used to properly measure performance in ATLAS's autotuning step in the presence of DVFS.

For future research, we plan to investigate other unroll factors and the use of software prefetching to further optimize the *gemm*μ performance. For heterogeneous systems, we demonstrated that adapting ATLAS's parallel GEMM for such systems can provide improved performance. We can extend the idea for all routines in ATLAS to complete the support for heterogeneous systems. Finally, as discussed in Section 5.3.3, we need to investigate the accuracy of our proposed performance metrics on other architectures (e.g. x86).

CHAPTER 6
SUMMARY, FUTURE WORK AND CONCLUSIONS

In Chapter 2 we outlined a new approach for developing and maintaining a microkernel-based BLAS. We did this study using the most challenging BLAS to adequately tune, TRSM, and then showed TRMM as an example of a more typical BLAS case. We reported initial results that, even prior to our full empirical tuning, are extremely impressive, with across-the-board speedups over ATLAS's prior BLAS support, and results competitive with Intel's MKL library. We believe this demonstrates the huge promise of this approach, and that it should lead to even greater speedups for the parallel BLAS, where the low-order terms like the TRSM component of a GEMM-based TRSM are likely to be more visible due to increasing scale.

For future work, we must first get the new tuning design working enough to autotune the new *trsm*μ and *trmm*μ as outlined. We can then use these timings to validate and improve the block predicting model outlined in Sections B.3 and B.4. Next, we can extend both the code and model to support the entire serial BLAS, at which point this work should be ready for journal publication, possibly in TOMS (Transactions on Mathematical Software). We may also wish to provide and support microkernel-based serial factorizations as outlined in Chapter 3, though this is probably not necessary for publication.

In Chapter 4 we developed a *gemm*μ-based parallel LU factorization which, at the time of the research, was the best performing on both Intel and AMD machines. We conclude that the microkernel-based approach is the best we know of for small- and medium sized problems, and we have already reoriented ATLAS's tuning framework due to these results. Once the preceding serial work is published, we need to extend our microkernel-based BLAS and associated models to the parallel BLAS. We first need prototype parallel implementations, and then as the tuning and timing of parallel overheads come online, to predict the best blocking factors in parallel using models. If this work is successful, we we will then be

ready to resume our original work in predicting the parallel block factor and grid size more accurately, as described in Appendix B.1.

A promising approach for arbitrarily sized factorizations is to develop hybrid algorithms that use PCA for aggregate-cache sized problems, while using right-looking or other factorization variants for very large problems. We will need to first complete the parallel BLAS work discussed above before such a hybrid algorithm will be competitive, however. If this work is successful, it should be extended to the Cholesky and QR factorizations.

This ongoing parallel BLAS and/or modeling work should be publishable in conferences like IPDPS (International Parallel and Distributed Processing Symposium), PPoPP (Principles and Practice of Parallel Programming) and ICPP (International Conference on Parallel Processing), and the full research should fit well in a journal article for venues such as *Parallel Computing* and *Concurrency: Practice and Experience*.

In Chapter 5, we developed two *gemm*μ for ARM 64-bit architectures that improved ATLAS's GEMM performance by up to 39% for double-precision and 53% for single-precision. We know of other optimizations that we believe could improve this even further, so this may not be the ceiling for our performance.

The proof-of-concept for heterogeneous support was extremely compelling. Even the simple approach we outlined requires significant tuning and configure framework support, but if the heterogeneous trends continue it will become necessary. This work would be of interest to all the same publication venues as discussed above for the parallel BLAS and factorization work.

Finally, Chapter 5 presented intriguing experiments on doing empirical tuning in the face of DVFS (Dynamic Voltage and Frequency Scheduling). This work will need to be revisited as vendors supply us with more DVFS-related profiling.

REFERENCES

[1] Emmanuel Agullo, Bilel Hadri, Hatem Ltaief, and Jack Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 20:1–20:12, New York, NY, USA, 2009. ACM.

[2] AMD. Acml homepage. `http://www.amd.com/acml/`, 2013.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.

[4] E. Anderson, J. Dongarra, and S. Ostrouchov. Lapack working note 41: Installation guide for lapack. Technical report, University of Tennessee, Knoxville, TN, USA, 1992.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[6] 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. *ACM Trans. Math. Softw.*, 31(1):1–26, March 2005.

[7] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the ACM SIGARC International Conference on SuperComputing*, Vienna, Austria, July 1997.

[8] C. Bischof and C. van Loan. The *WY* representation for products of householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1 1987.

[9] C. Bishof. Adaptive Blocking in the QR Factorization. *The Journal of Supercomputing*, 3(3):193–208, 1989.

[10] L. 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*. SIAM, Philadelphia, PA, 1997.

[11] BLAS-group. Blas homepage. `http://www.netlib.org/blas/`, 2011.

[12] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov. The impact of multicore on math software. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 1–10, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled qr factorization for multicore architectures. *Concurr. Comput. : Pract. Exper.*, 20(13):1573–1590, September 2008.

[14] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, January 2009.

[15] Anthony M Castaldo, Clint Whaley, and Siju Samuel. Scaling lapack panel operations using parallel cache assignment. *ACM Transactions on Mathematical Software (TOMS)*, 39(4):22:1–22:30, July 2013.

[16] Anthony M. Castaldo and R. Clint Whaley. Achieving Scalable Parallelization For The Hessenberg Factorization. In *Proceedings of IEEE Cluster*, pages 65–73, Austin, TX, September 2011. IEEE.

[17] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 123–132, New York, NY, USA, 2008. ACM.

[18] M. Dayde, I. Duff, and A. Petitet. A Parallel Block Implementation of Level 3 BLAS for MIMD Vector Processors. *ACM Transactions on Mathematical Software*, 20(2):178–193, 1994.

[19] Qingyuan Deng, D. Meisner, A. Bhattacharjee, T.F. Wenisch, and R. Bianchini. Coscale: Coordinating cpu and memory system dvfs in server systems. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 143–154, Dec 2012.

[20] F. Desprez, S. Domas, and B. Tourancheau. *Optimization of the ScaLAPACK LU factorization routine using communication/computation overlap*, pages 1–10. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[21] Simplice Donfack, Jack Dongarra, Mathieu Faverge, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Ichitaro Yamazaki. A survey of recent developments in parallel implementations of gaussian elimination. *Concurr. Comput. : Pract. Exper.*, 27(5):1292–1309, April 2015.

[22] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[23] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[24] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[25] J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable dense linear algebra libraries. In *Proceedings Scalable High Performance Computing Conference SHPCC-92.*, pages 372–379, Apr 1992.

[26] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving Numerical Accuracy and High Performance Using Recursive Tile LU Factorization with Partial Pivoting. *Concurrency and Computation: Practice and Experience*, 26(7):1408–1431, May 2014.

[27] Jack Dongarra, Emmanuel Jeannot, and Julien Langou. Modeling the LU factorization for SMP clusters. Research report, University of Tennessee, 2006.

[28] E. Elmroth and F. Gustavson. Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance. *IBM Journal of Research and Development*, 44(4):605–624, 2000.

[29] Krassimir Georgiev and Jerzy Wasniewski. Recursive version of lu decomposition. In *Revised Papers from the Second International Conference on Numerical Analysis and Its Applications*, NAA '00, pages 325–332, London, UK, UK, 2001. Springer-Verlag.

[30] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, December 2001.

[31] F. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.

[32] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive blocked data formats and blas's for dense linear algebra algorithms. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 195–206, 1998.

[33] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Superscalar gemm-based level 3 blas – the on-going evolution of a portable and high-performance library. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 207–215, 1998.

[34] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 90:1–90:12, New York, NY, USA, 2013. ACM.

[35] R. Hanson, F. Krogh, and C. Lawson. A Proposal for Standard Linear Algebra Subprograms. *ACM SIGNUM Newsl.*, 8(16), 1973.

[36] Md Rakib Hasan, Eric Van Hensbergen, and Wade Walker. Reliable performance autotuning in presence of dvfs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, 2015.

[37] Md Rakib Hasan and R. Clint Whaley. Effectively exploiting parallel scale for all problem sizes in LU factorization. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS2014)*, pages 1039–1048, Phoenix, AZ, May 2014. IEEE press.

[38] Greg Henry. BLAS Based on Block Data Structures. Technical Report 89, Cornell University, January 1992. `http://ecommons.library.cornell.edu/bitstream/1813/5471/1/92-089.pdf`.

[39] Francisco D. Igual, Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Robert A. Van De Geijn, and Field G. Van Zee. The flame approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *J. Parallel Distrib. Comput.*, 72(9):1134–1143, September 2012.

[40] ARM Inc. Juno arm development platform soc technical reference manual. `https://static.docs.arm.com/ddi0515/d/DDI0515.pdf`, 2015.

[41] Intel. Measuring processor power. `http://www.intel.com/content/dam/doc/white-paper/resources-xeon-measuring-processor-power-paper.pdf`, 2011.

[42] Intel. Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide, part 2. `https://software.intel.com/sites/default/files/managed/7c/f1/253669-sdm-vol-3b.pdf`, 2016.

[43] Intel. Mkl homepage. `http://software.intel.com/en-us/mkl/`, 2016.

[44] J. Choi and J. Dongarra and S. Ostrouchov and A Petitet and D. Walker and R. C. Whaley. The Design and Implementation of ScaLAPACK LU, QR, and Cholesky. *Scientific Programming*, 5:173–184, 1996.

[45] B. Kågström, P. Ling, and C. van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. Technical Report UMINF 95-18, Department of Computing Science, Umeå University, 1995. Submitted to ACM TOMS.

[46] B. Kågström, P. Ling, and C. van Loan. Gemm-based level 3 blas: High performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software*, 24(3):268–302, 1998.

[47] Bo Kågström and Charles van Loan. Algorithm 784: Gemm-based level 3 blas: Portability and optimization issues. *ACM Trans. Math. Softw.*, 24(3):303–316, September 1998.

[48] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurr. Comput. : Pract. Exper.*, 22(1):15–44, January 2010.

[49] LAPACK. `LAPACK/TIMING/LIN/dopla.f`. `http://www.netlib.org/lapack/lapack-3.0.tgz`, 2013.

[50] LAPACK-group. Lapack homepage. `http://www.netlib.org/lapack/`, 2011.

[51] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[52] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software*, 43(2):12:1–12:18, August 2016.

[53] M. Marqués, G. Quintana-Ortí, E. S. Quintana-Ortí, and R. vande Geijn. Using desktop computers to solve large-scale dense linear algebra problems. *The Journal of Supercomputing*, 58(2):145–150, 2011.

[54] Mercedes Marques, Gregorio Quintana-Orti, Enrique S. Quintana-Orti, and Robert A. van de Geijn. Out-of-Core Computation of the QR Factorization on Multi-Core Processors. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 809–820, Heidelberger Platz 3, Berlin, 2009. Springer Berlin / Heidelberg.

[55] Jacob Pan. Rapl (running average power limit) driver. `https://lwn.net/Articles/545745/`, 2014.

[56] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3):14:1–14:26, July 2009.

[57] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, March 2012.

[58] Tyler M. Smith, Robert A. van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)*, 2014.

[59] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[60] G.W. Stewart. *Introductions to Matrix Computations*, chapter 2, 3 & Appendix 3. Academic Press, 1973.

[61] Majedul Haque Sujon, R. Clint Whaley, and Qing Yi. Vectorization past dependent branches through speculation. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 353–362, Piscataway, NJ, USA, 2013. IEEE Press.

[62] Ananta Tiwari, Anthony Gamst, MichaelA. Laurenzano, Martin Schulz, and Laura Carrington. Modeling the impact of reduced memory bandwidth on hpc applications. In Fernando Silva, Ins Dutra, and Vtor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 63–74. Springer International Publishing, 2014.

[63] S. Toledo. Locality of Reference in LU Decomposition with Partial Pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[64] Field G. Van Zee. Implementing high-performance complex matrix multiplication via the 1m method. submitted.

[65] Field G. Van Zee and Tyler Smith. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Transactions on Mathematical Software*, 2017. accepted.

[66] Field G. Van Zee, Tyler Smith, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, Tze Meng Low, Bryan Marker, Lee Killough, and Robert A. van de Geijn. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software*, 42(2):12:1–12:19, June 2016.

[67] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, June 2015.

[68] R. Clint Whaley. *Automated Empirical Optimization of High Performance Floating Point Kernels*. PhD thesis, The Florida State University, November 2004.

[69] R. Clint Whaley. Empirically Tuning LAPACK's Blocking Factor for Increased Performance. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, volume 3, pages 303–310, Wisla, Poland, October 2008. IEEE press.

[70] R. Clint Whaley and Anthony M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. Technical Report CS-TR-2008-001, University of Texas at San Antonio, January 2008.

[71] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. `http://www.netlib.org/lapack/lawns/lawn131.ps`.

[72] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, San Antonio,

TX, USA, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.**
`http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`.

[73] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[74] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[75] R. Clint Whaley and Antoine Petitet. Atlas homepage. `http://math-atlas.sourceforge.net/`, 2011.

[76] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[77] R. Clint Whaley and David B. Whalley. Tuning high performance kernels through empirical compilation. In *The 2005 International Conference on Parallel Processing*, pages 89–98, Oslo, Norway, June 2005.

[78] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the plasma numerical library to the openmp standard. *International Journal of Parallel Programming*, 45(3):612–633, 2017.

[79] Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. Scalable Parallelization of FLAME Code via the Workqueuing Model. *ACM Transactions on Mathematical Software*, 34:10:1–10:29, March 2008.

[80] Field G. Van Zee, Ernie Chan, Robert A.van de Geijn, Enrique S. Quintana-Ort, and Gregorio Quintana-Ort. The libflame library for dense matrix computations. *Computing in Science Engineering*, 11(6):56–63, 2009.

APPENDIX A
HANDLING TRAPEZOIDAL UNROLL BLOCKS IN *trmmμ*

In Section 2.2.2, we discussed how the four variants LLN, LUN, RLN, and RUN of *trmmμ* are implemented for $u_m = i \times u_k$ or $u_n = i \times u_k$ for the left- or right-variants respectively where $i \geq 1$. In this chapter, we will discuss the same variants of *trmmμ* when $u_k = i \times u_m$ or $u_k = i \times u_n$ case where $i > 1$ which leads to trapezoidal unroll blocks. Note that this only changes the copy microkernels for the triangular A and the *trmmμ* but not the copy microkernel for the B matrix.

A.1 *trmmμ* for LLN-variant

Recall that LLN denotes the operation: $B = alpha \times A \times B$, where A is the lower triangular matrix and B is a general matrix.

Triangular copy microkernel changes: The triangular A is partitioned into u_m-sized row panels and each of these row panels are partitioned into $(u_m \times u_k)$-sized unroll blocks. Each of these row panels contains a triangular or trapezoidal block at the end. These blocks are zero-padded to make them $(u_m \times u_k)$-sized rectangular unroll blocks. Figure A.1 shows the partitioning and the zero-padded row panels of A for an example of LLN-variant where $u_k = 2u_m$.

***trmmμ* changes**: The required changes from a *gemmμ* are shown in Algorithm A.1.

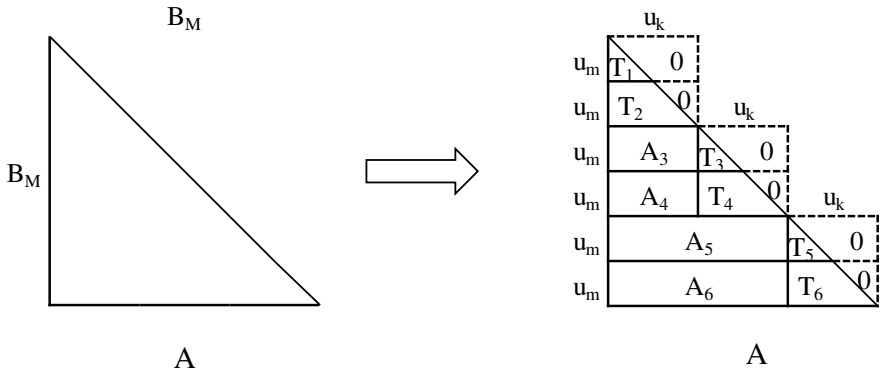


Figure A.1: LLN-variant: partitioning and zero-padding the lower-triangular A for $u_k = 2\ u_m$

.

Algorithm A.1: Pseudocode of looping and pointer updates for LLN-variant of *trmmμ* for $u_k = i \times u_m$ (changes from a *gemmμ* are highlighted in bold)

```
1  Function trmmμ-LLN(nmu, nnu, K, pA, pB, pC):
2  {
       arguments: nmu : no. of mu-sized row-panels in A matrix
                  nnu : no. of nu-sized column-panels in B matrix
                  K : size of the dimension common to A and B
                  pA : pointer to the triangular matrix
                  pB : pointer to the general matrix
                  pC : pointer to the result matrix
3      pA0 ← pA
4      pB0 ← pB
5      Askip ← mu × ku                    /* Initialize the skips needed */
6      Bskip ← nu × (K − ku)
7      off ← 0
8      for i ← 1 to nmu do
9       {
10        for j ← 1 to nnu do
11         {
12           Kbound ← (i × mu) − off + ku
13           for k ← 1 to Kbound do
14            {
15              ⟹ perform one unroll-block multiplication
16              pA ← pA + mu
17              pB ← pB + nu
18            }
19           pB ← pB + Bskip            /* skip to correct column-panel */
20           pA ← pA0
21           pC ← pC + (mu × nu)
22         }
23        pA0 ← pA0 + Askip                /* skip to correct row-panel */
24        pA ← pA0
25        pB ← pB0
26        off ← off + mu
27        if (off ≥ ku) then
28         {
29           off ← off − ku
30           Askip ← Askip + (mu × ku)       /* update skips for next */
31           Bskip ← Bskip − (ku × nu)         /* row-panel if needed */
32         }
33      }
34 }
```

A.2 *trmmμ* for LUN-variant

LUN denotes the operation: $B = alpha \times A \times B$, where A is the upper triangular matrix and B is a general matrix.

Triangular copy microkernel changes: The triangular A is partitioned into u_m-sized row panels and each of these row panels are partitioned into $(u_m \times u_k)$-sized unroll blocks. Each of these row panels contains a triangular or trapezoidal block at the beginning. These blocks are zero-padded to make them $(u_m \times u_k)$-sized rectangular unroll blocks. Figure A.2 shows the partitioning and the zero-padded row panels of A for an example of LUN-variant where $u_k = 2u_m$.

***trmmμ* changes**: The required changes from a *gemmμ* are shown in Algorithm A.2.

A.3 *trmmμ* for RLN-variant

RLN denotes the operation: $B = alpha \times B \times A$, where A is the lower triangular matrix and B is a general matrix.

Triangular copy microkernel changes: The triangular A is partitioned into u_n-sized column panels and each of these column panels are partitioned into $(u_k \times u_n)$-sized unroll blocks. Each of these column panels contains a triangular or trapezoidal block at the beginning. These blocks are zero-padded to make them $(u_k \times u_n)$-sized rectangular unroll blocks. Figure A.3 shows the partitioning and the zero-padded column panels of A for an example of RLN-variant where $u_k = 2u_n$.

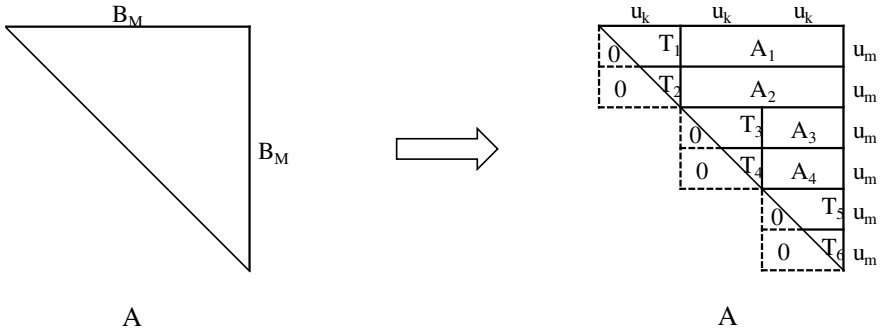


Figure A.2: LUN-variant: partitioning and zero-padding the lower-triangular A for $u_k = 2\ u_m$

.

Algorithm A.2: Pseudocode of looping and pointer updates for LUN-variant of *trmmμ* for $u_k = i \times u_m$ (changes from a *gemmμ* are highlighted in bold)

```
1  Function trmmμ-LUN(nmu, nnu, K, pA, pB, pC):
2  {
     arguments: nmu : no. of mu-sized row-panels in A matrix
                nnu : no. of nu-sized column-panels in B matrix
                K : size of the dimension common to A and B
                pA : pointer to the triangular matrix
                pB : pointer to the general matrix
                pC : pointer to the result matrix
3    pA0 ← pA
4    pB0 ← pB
5    Askip ← mu × K                    /* Initialize the skips needed */
6    Bskip ← 0
7    off ← 0
8    for i ← 1 to nmu do
9     {
10      for j ← 1 to nnu do
11       {
12         Kstart ← (i × mu) − off
13         for k ← Kstart to K do
14          {
15             ⟹ perform one unroll-block multiplication
16             pA ← pA + mu
17             pB ← pB + nu
18          }
19         pB ← pB + Bskip           /* skip to correct column-panel */
20         pC ← pC + (mu × nu)
21         pA ← pA0
22       }
23      pA0 ← pA0 + Askip             /* skip to correct row-panel */
24      pA ← pA0
25      pB ← pB0
26      off ← off + mu                /* update the skips if needed */
27      if (off ≥ ku) then
28       {
29         off ← off − ku
30         Askip ← Askip − (mu × ku)
31         Bskip ← Bskip + (ku × nu)
32       }
33      pB ← pB + Bskip              /* advance pB to correct block */
34    }
35 }
```

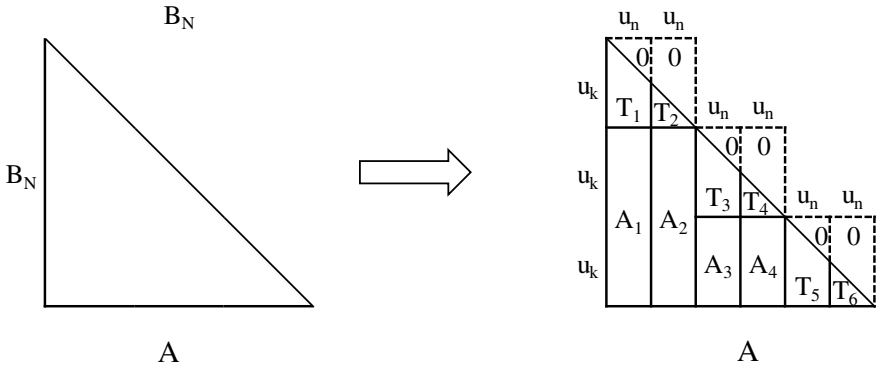


Figure A.3: RLN-variant: partitioning and zero-padding the lower-triangular A for $u_k = 2\ u_n$

.

***trmmμ* changes**: The required changes from a *gemmμ* are shown in Algorithm A.3.

A.4 *trmmμ* for RUN-variant

RUN denotes the operation: $B = alpha \times B \times A$, where A is the upper triangular matrix and B is a general matrix.

Triangular copy microkernel changes: The triangular A is partitioned into u_n-sized column panels and each of these column panels are partitioned into $(u_k \times u_n)$-sized unroll blocks. Each of these column panels contains a triangular or trapezoidal block at the end. These blocks are zero-padded to make them $(u_k \times u_n)$-sized rectangular unroll blocks. Figure A.4 shows the partitioning and the zero-padded column panels of A for an example of RUN-variant where $u_k = 2u_n$.

***trmmμ* changes**: The required changes from a *gemmμ* are shown in Algorithm A.4.

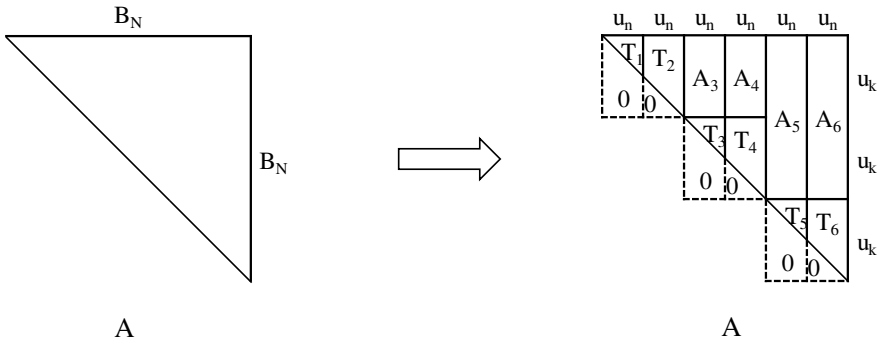


Figure A.4: RUN-variant: partitioning and zero-padding the lower-triangular A for $u_k = 2\ u_n$

.

Algorithm A.3: Pseudocode of looping and pointer updates for RLN-variant of *trmmμ* for $u_k = i \times u_n$ (changes from a *gemmμ* are highlighted in bold)

```
1  Function trmmμ-RLN(nmu, nnu, K, pA, pB, pC):
2  {
     arguments: nmu : no. of mu-sized row-panels in A matrix
                nnu : no. of nu-sized column-panels in B matrix
                K : size of the dimension common to A and B
                pA : pointer to the general matrix
                pB : pointer to the triangular matrix
                pC : pointer to the result matrix
3    pA0 ← pA
4    pB0 ← pB
5    Apan ← mu × K                                  /* size of one row-panel */
6    for i ← 1 to nmu do
7     {
        /* Initialize the skips needed                                      */
8       Askip ← 0
9       off ← 0
10      for j ← 1 to nnu do
11       {
12         Kstart ← (j × nu) − off
13         for k ← Kstart to K do
14          {
15             ⟹ one unroll-block multiplication
16             pA ← pA + mu
17             pB ← pB + nu
18          }
           /* update skip for next column-panel                             */
19         off ← off + nu if (off ≥ ku) then
20          {
21             off ← off − ku Askip ← Askip + (mu × ku)
22          }
23         pA ← pA0 + Askip                  /* skip to correct block */
24         pC ← pC + (mu × nu)
25       }
26      pB ← pB0
27      pA0 ← pA0 + Apan                    /* move to next row-panel */
28      pA ← pA0
29    }
30 }
```

Algorithm A.4: Pseudocode of looping and pointer updates for RUN-variant of *trmmμ* for $u_k = i \times u_n$ (changes from a *gemmμ* are highlighted in bold)

```
1  Function trmmμ-RUN(nmu, nnu, K, pA, pB, pC):
2  {
     arguments: nmu : no. of mu-sized row-panels in A matrix
                nnu : no. of nu-sized column-panels in B matrix
                K : size of the dimension common to A and B
                pA : pointer to the general matrix
                pB : pointer to the triangular matrix
                pC : pointer to the result matrix
3    pA0 ← pA
4    pB0 ← pB
5    Apan ← mu × K                                  /* size of one row-panel */
6    for i ← 1 to nmu do
7     {
        /* offset needed to keep track of K-loop                              */
8       off ← 0
9       for j ← 1 to nnu do
10       {
11         Kbound ← (j × nu) − nu − off + ku
12         Kbound ← min(K, Kbound)
13         for k ← 1 to Kbound do
14          {
15            ⟹ perform one unroll-block multiplication
16            pA ← pA + mu
17            pB ← pB + nu
18          }
           /* update offset for next K-loop                                   */
19         off ← off + nu if (off ≥ ku) then
20          {
21            off ← off − ku
22          }
23         pC ← pC + (mu × nu)
24         pA ← pA0
25       }
26      pB ← pB0
27      pA0 ← pA0 + Apan                          /* move to next row-panel */
28      pA ← pA0
29    }
30 }
```

APPENDIX B

COMPUTATIONAL MODEL FOR MICROKERNEL-BASED BLAS AND LAPACK OPERATIONS

In Chapters 2-4, we discussed how ATLAS's *gemm*μ framework can be used to achieve high performance for all sized problems for BLAS and LAPACK routines. One problem for such an approach is the need to select the best block factors during execution based on the problem size. We will try to address this issue in this chapter by building computational models for our *gemm*μ-based routines. Since our primary goal is to model our *gemm*μ-based parallel LU factorization, we will start our discussion with building a computational model for it, and later in the chapter, we discuss building computational models for serial routines that are less complex.

B.1 Computational Model for Parallel LU Factorization

In our parallel LU factorization, we divide the input into blocks and the blocks are then distributed in a cyclic fashion for a $r \times c$ process grid. For best performance, we need to not only find the best blocking factor B but also the grid size r and c. Note that for a parallel routine, smaller B provides more parallelism but may impact *gemm*μ performance. Also, using smaller r may reduce the parallel overhead but it leads to increased idle times due to less parallelism within a column panel. In this section, we will build a computational model for our LU factorization to predict B, r and c for a given problem size (N).

Building a computational model for a parallel routine is much harder (compared to computational model for serial routines discussed in later in the chapter) due to unpredictable thread scheduling and inconsistent cache effects. There are some prior research [10, 20, 25, 27] that tried to build computational model for LU factorization for SMP clusters but they only focused on asymptotic sized problems for which thread scheduling and cache effects do not tend to affect the model. For our research, we want to try to incorporate some such effects into the model to improve its small-sized accuracy.

To incorporate the scheduling effects, we built a simulator that imitates the steps of our parallel LU factorization and accumulates the estimated time taken by each step. For incorporating the caching effects, we use different timings for in-cache and out-of-cache problems but as we will later see, this adaptation still cannot predict the best parameters for problems that are at the cache-size boundary. Note that in an ideal case, the simulator is going to estimate the total execution time for each thread and use the maximum as the execution time for the whole LU factorization. However, with our 2-D grid approach, each thread working on a column panel are synchronized after almost every step of the computation. To simplify our simulator using this characteristic of our approach, we only estimated the execution time for one thread for each column of the process grid (`pcol`). Note that for each column panel of the input, we estimate the maximum time taken among the threads of the pcol to process that panel and we accumulate that to the execution time of that pcol that owns the column panel. Consider the example shown in Figure 4.4b. For the first column panel, the execution time would only be the panel factorization time where the panel factorization is done by the first column of the process grid. We have $N_p = N/B$ blocks to factorize, performed by r threads. Therefore, the execution time for the first panel would be roughly the time taken by one thread for LU factorization on an input of size $(\lceil \frac{N_p}{r} \rceil B) \times B$. Now consider the second panel where the first step is to apply B pivots on the top $(B \times B)$-sized block and then perform TRSM on it. Both operations are performed in parallel by r threads. So, the estimated pivoting time (t_a) would be the time taken to apply B pivots on $\lceil \frac{B}{r} \rceil$ columns and the estimated TRSM time (t_s) would be the time taken for a TRSM of size $B \times \lceil \frac{B}{r} \rceil$ i.e. $B \times B$ sized triangle and $\lceil \frac{B}{r} \rceil$ right hand sides. Both pivoting and TRSM time estimations can be done through prior empirical timing and interpolation/extrapolation if needed. For the GEMM updates, we have $Nb-1$ calls to *gemm*μ on the second panel that are performed by r threads. The estimated time for GEMM updates would be $t_m \times \lceil \frac{N_p - 1}{r} \rceil$ where t_m is the time taken by the *gemm*μ for performing a square GEMM on an $B \times B$ input. After the

GEMM is done, the panel (i.e. $(N_p - 1)$ blocks) needs to be factorized. As before, the estimated time for the second panel factorization would be the time for serial LU factorization on an input of size $(\lceil \frac{N_p - 1}{r} \rceil B) \times B$.

$$\text{Total panel factorization time}, T_f = \sum_{j=1}^{N_p} t_f \left(\left\lceil \frac{(N_p - j + 1)}{r} \right\rceil B, B \right) \tag{B.1}$$

$$\text{Total pivot time}, T_a = t_a \left(B, \left\lceil \frac{B}{r} \right\rceil \right) \sum_{j=1}^{N_p} \sum_{i=1}^{j-1} (i - 1) \tag{B.2}$$

$$\text{Total TRSM time}, T_s = t_s \left(B, \left\lceil \frac{B}{r} \right\rceil \right) \sum_{j=1}^{N_p} \sum_{i=1}^{j-1} (i - 1) \tag{B.3}$$

$$\text{Total GEMM time}, T_m = t_m(B, B, B) \times \sum_{j=1}^{N_p} \sum_{i=1}^{j-1} \left\lceil \frac{N_p - i}{r} \right\rceil \tag{B.4}$$

Note that summing Equation B.1, B.2, B.3 and B.4 would provide an estimate for the whole LU factorization but not considering any scheduling effects (idle times waiting for data to be ready). Instead, to incorporate such idle times, our simulator estimates each component time and simulates the parallel LU factorization. As before, we can have two variants (in-cache and out-of-cache) of estimated time for each component to reflect the cache effects for small sized problems. Later we added another variant of timings where a panel fits in the aggregate cache. For a given input problem, we can use the simulator for each available B, r and c combination to predict the best one. To limit the possible number of combinations, in our simulator, we imposed a restriction of $r < c$. This restriction is always a good idea in our experience: the reason is that the communication within a process column is extremely tightly coupled, and if any r threads gets out of sync, the pcol working on the critical path is unnecessarily delayed, while the communication along the prow is effectively loosely coupled, and can be overlapped via look-ahead. This results in $r \leq c$ for every non-degenerate case that we have investigated so far.

Note that in the above discussion, we ignored the cost of thread communications and of copying the input matrix. At the time of this research, these timings are an in-progress

improvement of the *gemm*μ framework in ATLAS. Once the framework is updated, we can improve our computational model (i.e. the simulator) to incorporate these times for better predictions. Also note that in our simulator, we ignored the *infinite lookahead* technique due to its unpredictable scheduling effects.

B.1.1 Performance Results

In Section 4.4, we presented the performance of our parallel LU factorization with the best combination of B, r and c (with an exhaustive search) for each problem sizes. In this section, we compare those results with the performance of our LU factorization using the predicted combination B, r and c from our simulator. As mentioned before, to measure the accuracy of the computational model for our parallel LU factorization, we measured its performance using the block factor and grid size from: (a) an exhaustive search and (b) the prediction of our simulator for all selected problem sizes. Figure B.1, B.2 and B.3 shows the percentage of performance loss when using the prediction from our model over the best configuration from an exhaustive search. As we can see, for mid-to-asymptotic sized problems, the accuracy of our model is within 3% but for small problems, the performance loss is up to 73% on O32. This high error is primarily caused by two issues: (1) the cache effects are extremely unpredictable at that range and (2) for such small problems, we also need to figure out how many available cores to use since using all available cores is not optimal at that range due to significant parallel overhead. As we can see, both these issues become worse with scale i.e. up to 14% loss on X12 vs. 73% on O32. This indicates that there might be some effects that we are unable to capture in our current model. To mitigate the performance loss for small sizes, one simple idea would be to have ATLAS try the top 3 predictions from our simulator and save the best combination for later use. Also note that on X12 and X24, the two regions for low accuracy of our model are: for very small problems (around 200-1,000) and for the problems of size at the cache size boundary (around 2,000-4,000). For very small problems, parallel overhead such as thread startup and communication cost are significant; we plan to

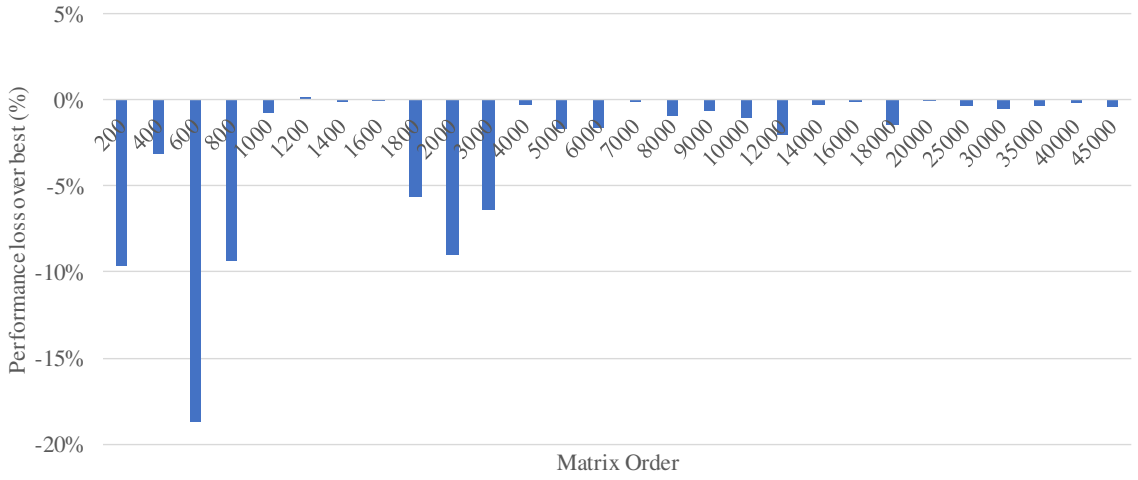


Figure B.1: Performance loss of our parallel *gemmμ*-based LU factorization on X12

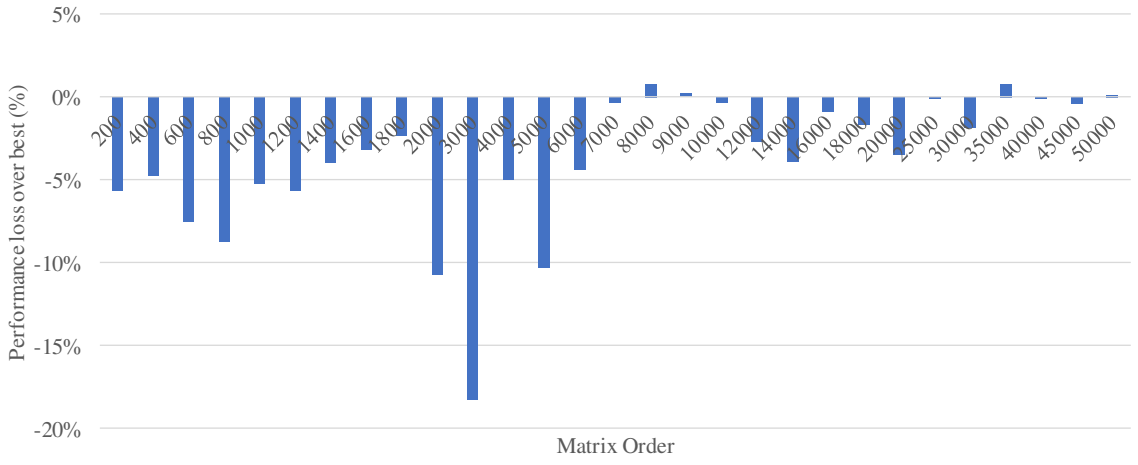


Figure B.2: Performance loss of our parallel *gemmμ*-based LU factorization on X24

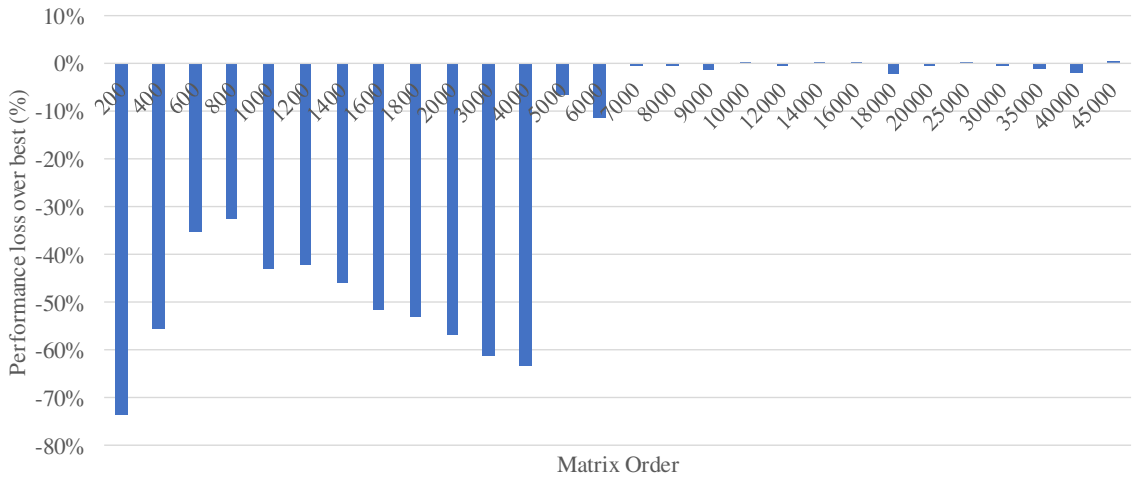


Figure B.3: Performance loss of our parallel *gemmμ*-based LU factorization on O32

extend our model to include them as future research. To isolate this issue from the cache boundary issue, we started building computational models for serial routines as discussed in the following sections.

B.2 Motivation for Modeling Serial Routines

Recall that for serial routines, we only need to know the block factor to use for the best performance for a given problem. A simple approach for selecting the block factors is to use the best performing ones for *gemm*μ. This approach is usually sufficient for asymptotic sized problems but it is not the best for small problems. To better understand this, recall that our *gemm*μ-based TRMM requires two searches:

1. Finding an optimized *gemm*μ with a restriction of $B_M = B_K$ for left variants or $B_N = B_K$ for right-variants, where B_M, B_N, and B_K are block factors in the M, N, and K dimensions, respectively.

2. Finding an optimized *trmm*μ with a restriction of $u_m = i \times u_k$ or $u_k = i \times u_m$ for left variants ($u_n = i \times u_k$ or $u_k = i \times u_n$ for right variants), where $i \geq 1$ and u_m, u_n, and u_k are unroll factors in the M, N, and K dimensions.

Due to *trmm*μ performing a small portion of the total number of required floating point operations (FLOPs), we can afford to select only one *trmm*μ and use it for any block-sized TRMM. Selecting only one combination of *gemm*μ and the block factors, however, is not always optimal. Consider the triangle shown in Figure B.4 for a LLN-variant of TRMM with $M = 240$ and $N = 240$. Let us assume that our *trmm*μ for block size of 240 achieves 50% of the machine's theoretical peak performance. If we use a block factor of $B_M = B_N = B_K = 240$ (shown in Figure B.4a), our full TRMM runs at *trmm*μ's efficiency (i.e. 50%). On the other hand, if we use the block factors $B_M = B_N = B_K = 120$ (shown in Figure B.4b), roughly half of the computation will be done by the *trmm*μ and the rest will be done by *gemm*μ (shown in gray). We will discuss in detail how we can calculate this in Section B.3.

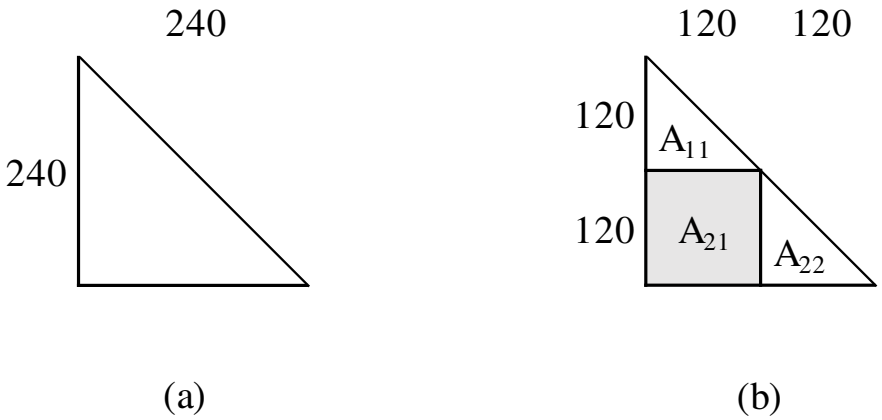


Figure B.4: Triangle partitioning for an example of LLN-variant of TRMM: (a) $B_M = 240$ (b) $B_M = 120$

Since *gemm*μ are extremely optimized, they can achieve between 25%-95% (depending on architecture, problem and block sizes) of the peak performance. On recent Intel machines (e.g. Haswell-EP), a *gemm*μ with $B_M = B_N = B_K = 120$ achieves about 85% and with $B_M = B_N = B_K = 240$ achieves about 90% of peak performance. Note that assuming the performance of 45% of peak for our *trmm*μ with $B_M = B_N = B_K = 120$, the overall performance of our TRMM for the above problem would be $(0.5 \times 45\%) + (0.5 \times 85\%) = 65\%$. As we can see, even though the performance of both *gemm*μ and *trmm*μ is higher for a block factor of 240, it is not the best block factor for our example problem. Therefore, the best block factors to use will vary depending on relative *gemm*μ and *trmm*μ performance and the input size. Therefore, during our *gemm*μ search we tune and time a set of block factor combinations and one of those combinations needs to be used during runtime based on the given problem size. The best way to do this would be to exhaustively tune each routine for all problem sizes possible and store the selections in a table to be used later for a user-given problem. However, tuning for all problem sizes is not feasible in practice. A compromise is to exhaustively tune for some representative problem ranges and interpolate/extrapolate for other problem sizes. Even with smart pruning of potentially non-optimal solutions in this exhaustive search, tuning can take a couple days even for one routine (e.g. our parallel LU factorization). To avoid this, our proposed approach is to build a computational model for

each routine that combines a few empirically tuned data points and a theoretical model of the operation based on its FLOP count. Note that the FLOP count analysis we are doing in this chapter are only extensions of prior research [4, 49] with a focus on our microkernel-based operations. In this chapter, we discuss how we can build such computational models for our *gemm*μ-based serial TRMM, TRSM, Cholesky and LU factorization. Note that for all the models discussed in this chapter, we assume that the problem sizes are multiple of the block factors (i.e. $M = i \times B_M$, $N = j \times B_N$, where $i, j \geq 1$). This simplifies the discussion by omitting the analysis for partial blocks or panels. In actual implementation of the models, proper analysis for partial blocks or panels are included.

B.3 Computational Model for *gemm*μ- and *trmm*μ-based TRMM

In this section, we will develop a computational model for TRMM using the FLOP count analysis of TRMM operation and the empirical timings of the basic components needed for TRMM.

B.3.1 Minimal FLOP Count for TRMM

In this section, we analyze the TRMM operation to find the minimum number of FLOPs required to perform the operation. Our analysis will assume $DIAG = N$ so that the diagonal elements are always used for multiplication as in our *trmm*μ discussed in Section 2.3.1. Consider the problem in equation B.5, where $M = 3$, $N = 2$ and $alpha = 1$.

$$\begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \\ Z_{31} & Z_{32} \end{bmatrix} = \begin{bmatrix} A_{11} & 0 & 0 \\ A_{21} & A_{22} & 0 \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix} \tag{B.5}$$

Recall that the computation of the first column of Z requires the steps shown in Figure B.5. Note that for Z_{11}, we only need one multiplication. For Z_{21}, we need two multiplications and one addition and for Z_{31}, we need three multiplications and two additions. In general, to compute i-th row element of Z, we need i multiplications and $(i - 1)$ additions. Therefore,

1. $Z_{31} = A_{33}B_{31}$ ..... 1 mult
2. $Z_{31} = Z_{31} + A_{31}B_{11} + A_{32}B_{21}$ ..... 2 mults, 2 adds
3. $Z_{21} = A_{22}B_{21}$ ..... 1 mult
4. $Z_{21} = Z_{21} + A_{21}B_{11}$ ..... 1 mult, 1 add
5. $Z_{11} = A_{11}B_{11}$ ..... 1 mult

Figure B.5: Computational steps of LLN-variant of TRMM for first column of Z

for an M-length column of Z, the total number of FLOPs required is given below:

$$\begin{aligned} \text{FLOPs for } M\text{-length column} &= \sum_{i=1}^{M}(i) + (i-1) \\ &= \sum_{i=1}^{M}(2i-1) \\ &= M\left(\frac{2M-1+1}{2}\right) \\ &= M^2 \end{aligned} \tag{B.6}$$

Since the same operations are repeated for each column of Z, the total number of FLOPs needed for an $(M \times N)$-sized LLN-TRMM is given below:

$$\text{Total FLOPs in LLN-TRMM} = NM^2 \tag{B.7}$$

In terms of computations needed, the LUN-variant is symmetric to the LLN-variant, thus requiring the same number of FLOPs. Recall that other left-variants (e.g. LLT, LUT, etc.) are mere reflections of LLN and LUN variants as described in Section 2.2.1.5. Therefore, Equation B.7 applies for all the left-variants of TRMM.

For right-variants, consider the example of RUN-variant shown in equation B.8, where $M = 2$, $N = 3$, and $alpha = 1$.

$$\begin{bmatrix} Z_{11} & Z_{12} & Z_{13} \\ Z_{21} & Z_{22} & Z_{23} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \end{bmatrix} \times \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & A_{33} \end{bmatrix} \tag{B.8}$$

Recall that for right-variants, we process one row at a time. For Z_{11}, we only need one multiplication. For Z_{12}, we need two multiplications and one addition. In general, for i-th column element, we need i multiplications and $(i-1)$ additions. Therefore, the total number of FLOPs needed for one N-sized row of Z can be derived as:

$$\begin{aligned} \text{FLOPs for } N\text{-sized row} &= \sum_{j=1}^{N}(j) + (j-1) \\ &= \sum_{j=1}^{N}(2j-1) \\ &= N\left(\frac{2N-1+1}{2}\right) \\ &= N^2 \end{aligned} \tag{B.9}$$

Since the same operations are repeated on each row for right-variants, the total number of FLOPs needed for an $(M \times N)$-sized RUN-variant of TRMM is given below:

$$\text{Total FLOPs in RUN-TRMM} = MN^2 \tag{B.10}$$

Similar to left-variants, Equation B.10 is valid for all the right-variants of TRMM.

For the GEMM-based approach, consider an example of the LLN-variant of TRMM where M=36 and N=12. Using Equation B.7, the total number of FLOPs required is 15,552. Assuming $B_M = 12$ and $B_N = 12$ for the GEMM-based approach, the blocked matrices are shown in Figure B.6. The steps needed to complete the operation are identical to the steps shown in Figure 2.8.

Step 1, 3, and 5 themselves are TRMM operations of size 12×12. We can use our previously developed Equation B.7 to compute the number of FLOPs needed for each of these steps (1,728 FLOPs each). For step 2, we need a GEMM update with $M = 12$, $N = 12$, and $K = 24$. A GEMM operation of size M, N, and K with $\alpha = 1$ and $\beta = 1$ requires $2MNK$ FLOPs [4]. Using this formula, step 2 requires $2 \times 12 \times 12 \times 24 = 6,912$ FLOPs. Similarly, step 4 requires $2 \times 12 \times 12 \times 12 = 3,456$ FLOPs. Therefore, the total number of FLOPs required $= (3 \times 1,728) + 6,912 + 3,456 = 15,552$, which is the exact number we previously

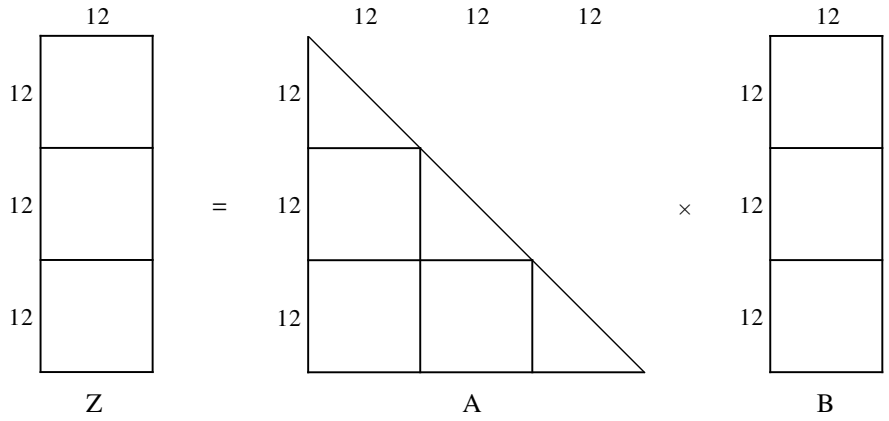


Figure B.6: An example of LLN-variant of GEMM-based TRMM: $M = 36$ and $N = 12$.

computed by using equation B.7 on the whole problem. This analysis shows that blocking doesn't require any extra FLOPs for GEMM-based TRMM.

B.3.2 FLOP Count for our *gemm*μ- and *trmm*μ-based TRMM

In Section B.3.1, we showed that GEMM-based TRMM doesn't require any extra FLOPs. However, this is not true for our *trmm*μ-based approach discussed in Section 2.2. As we have seen, our *trmm*μ zero-pads the diagonal blocks of the A matrix to make them full matrices, so that we can perform all computations as if it were a *gemm*μ routine (i.e. minimal changes are required to use the *gemm*μ framework). In Figure 2.16, let us assume $u_m = 4$ and $u_n = 12$ for the *trmm*μ. Note that for a TRMM operation of size 12×12, we have 3 u_m-sized row panels of A where each requires zero-padding in the diagonal unroll-block. As a result, each of these row panels are now rectangular during multiply with size 4×4, 4×8 and 4×12 from top to bottom row-panel respectively. Since these zero-padded row-panels are rectangular, they are treated just like GEMM row panels and are multiplied with the B column-panels of size 4×12, 8×12 and 12×12, respectively. Using GEMM's required FLOP formula for $\alpha = 1, \beta = 0$ for the diagonal block multiplication, the number of required FLOPs for Z_1, Z_2, and Z_3 are $4 \times 12 \times (2 \times 4 - 1) = 336$, $4 \times 12 \times (2 \times 8 - 1) = 720$, and $4 \times 12 \times (2 \times 12 - 1) = 1,104$, respectively. The total number of FLOPs done by our *trmm*μ in this case is 2,160, compared to the minimum 1,728 FLOPs required in the ideal case without zero-padding. Note that we increased the FLOPs required by our *trmm*μ by 25% for a 12×12 input with $u_m = 4$.

With the increased FLOPs done by our *trmmμ*, the total number of required FLOPs by our *gemmμ*- and *trmmμ*-based TRMM is $(3 \times 2,160) + 6912 + 3456 = 16,848$. Note that this is about 8.33% extra computations done by our *gemmμ*- and *trmmμ*–based approach. As M and N grows for the input of TRMM, this percentage of extra computations quickly becomes negligible. For example, for an LLN-TRMM of size $6,000 \times 6,000$, assuming $B_M = B_N = 240$ and $u_m = 12$, the amount of extra FLOPs is only about 0.2%. A general formula for the number of extra (i.e. useless) FLOPs for a $(B_M \times B_N)$-sized input in LLN-variant of *trmmμ* can be derived as follows[1]:

1. **Minimal TRMM FLOPs**: For a $u_m \times u_n$ unroll-block of Z, the minimal number of required FLOPs can be computed using the formula in Equation B.7.

$$\text{Minimal TRMM FLOPs} = u_n u_m^2 \tag{B.11}$$

2. **Unroll-block *trmmμ* FLOPs**: After the zero padding, *trmmμ* will perform the same number of FLOPs as a *gemmμ* on these blocks with $\alpha = 1$ and $\beta = 0$ assuming these zero-padded blocks are multiplied first[2]. Therefore, for a $u_m \times u_n$ block of Z, the FLOPs performed by the *trmmμ* is given below:

$$\begin{aligned} \text{Unroll-block } \textit{trmm}\mu \text{ FLOPs} &= u_n u_m (2u_m - 1) \\ &= 2u_n u_m^2 - u_n u_m \end{aligned} \tag{B.12}$$

3. **Unroll-block Useless FLOPs**: The number of useless FLOPs are the FLOPs due to zero-padding the diagonal $u_m \times u_n$ blocks. This can simply be computed by subtracting

[1]Note that we assume B_M is divisible by u_m and B_N is divisible by u_n for simplified discussion.

[2]For some variants, even though the zero-padded blocks are multiplied last with $\alpha = 1$ and $\beta = 1$, one non-diagonal *gemmμ* is performed with $\alpha = 1$ and $\beta = 0$ so the total number of FLOPs stays the same.

Equation B.11 from Equation B.12 as shown below:

$$\begin{aligned}\text{Unroll-block Useless FLOPs} &= 2u_n u_m^2 - u_n u_m - u_n u_m^2 \\ &= u_n u_m^2 - u_n u_m \\ &= u_n u_m (u_m - 1)\end{aligned} \tag{B.13}$$

4. **Panel Useless FLOPs**: For LLN-variant of *trmm*μ, a column-panel of size $B_M \times u_n$ is processed at a time. For each unroll-block of such panels, only one unroll-block computation requires useless FLOPs (the rest of the computation involves non-diagonal unroll-blocks). Since each column panel consists of $\frac{B_M}{u_m}$ unroll-blocks, we derive the number of useless FLOPs for such panel as shown below:

$$\begin{aligned}\text{Panel Useless FLOPs} &= \left(\frac{B_M}{u_m}\right) u_n u_m (u_m - 1) \\ &= B_M u_n (u_m - 1)\end{aligned} \tag{B.14}$$

5. **Block Useless FLOPs**: As discussed before, for LLN-variant of *trmm*μ, the same operation is repeated for each $(B_M \times u_n)$-sized column panel of Z. For a $(B_M \times B_N)$-sized block of Z, we have $\frac{B_N}{u_n}$ column-panels. Therefore, the total number of useless FLOPs by *trmm*μ for a block-sized input is given below:

$$\begin{aligned}\text{Block Useless FLOPs} &= \left(\frac{B_N}{u_n}\right) B_M u_n (u_m - 1) \\ &= B_N B_M (u_m - 1)\end{aligned} \tag{B.15}$$

Recall that *trmm*μ is called once for each $(B_M \times B_N)$-sized block of a Z. For an input of size $M \times N$, the total number of useless FLOPs by our *gemm*μ- and *trmm*μ-based TRMM can be computed as below:

$$\begin{aligned}\text{Total Useless FLOPs} &= \left(\frac{N}{B_N}\right)\left(\frac{M}{B_M}\right) B_N B_M (u_m - 1) \\ &= NM(u_m - 1)\end{aligned} \tag{B.16}$$

The total number of FLOPs required by our *gemmμ*- and *trmmμ*-based TRMM can be computed by simply adding Equation B.7 and B.16 as shown below:

$$\begin{aligned} \text{Total LLN-TRMM FLOPs} &= NM^2 + NM(u_m - 1) \\ &= NM(M + u_m - 1) \end{aligned} \tag{B.17}$$

As discussed before, the above equation can be applied for any left-variant of our *gemmμ*- and *trmmμ*-based TRMM. We can do similar analysis to show that the total number of FLOPs for any right-variant of our *gemmμ*- and *trmmμ*-based TRMM is $MN(N + u_n - 1)$.

B.3.3 Building the Computational Model

As mentioned before, *trmmμ* is called once for each block of Z. Let N_{rp} and N_{cp} be the number of row panels and column panels of Z, respectively. If t_T is the time taken by a single call to *trmmμ*, the total time spent (T_T) in *trmmμ* can be computed as shown below:

$$\begin{aligned} \text{Total } \textit{trmm}\mu \text{ Time, } T_T &= \sum_{j=1}^{N_{cp}} \sum_{i=1}^{N_{rp}} t_T \\ &= N_{cp} N_{rp} t_T \\ &= \left(\frac{N}{B_N}\right)\left(\frac{M}{B_M}\right) t_T \end{aligned} \tag{B.18}$$

Recall that for each i-th row-block of each column-panel, we need $(i-1)$ calls to the *gemmμ* for GEMM updates. If t_M is the time taken by one *gemmμ* call on one block sized input, the total time (T_M) spent for all calls to *gemmμ* is given below:

$$\begin{aligned} \text{Total } \textit{gemm}\mu \text{ Time, } T_M &= \sum_{j=1}^{N_{cp}} \sum_{i=1}^{N_{rp}} (i-1) t_M \\ &= \frac{1}{2} N_{cp} N_{rp} (N_{rp} - 1) t_M \\ &= \frac{1}{2} \left(\frac{N}{B_N}\right)\left(\frac{M}{B_M}\right)\left(\frac{M}{B_M} - 1\right) t_M \end{aligned} \tag{B.19}$$

To estimate the total time of our LLN-variant of TRMM, we can can use the following equation, derived by adding Equation B.18 and B.19:

$$\text{Total TRMM Time} = \left(\frac{N}{B_N}\right)\left(\frac{M}{B_M}\right) t_T + \frac{1}{2}\left(\frac{N}{B_N}\right)\left(\frac{M}{B_M}\right)\left(\frac{M}{B_M} - 1\right) t_M \tag{B.20}$$

We can use this equation for a given problem (i.e. M and N are known) to estimate the performance of using a certain combination of block factors (B_M and B_N) and choose the best one based on our estimated performance. We can do a similar analysis to build model for each variant of TRMM. The only unknown factors in our model (Equation B.20) are t_T and t_M that varies depending on the system. Section B.3.4 describes how we can estimate the values of t_T and t_M.

Note that in our computational model, we ignored the cost of copying the blocks. Although it may not be important for asymptotic problems, the cost for copy is significant for small problems. Therefore, we can improve the model by including this cost but that requires extra timing of the copy routines. At the time of this research, the timing of copy routines in the ATLAS *gemm*μ framework is in progress. In future research, we can include the copy cost to improve the decision of our model.

B.3.4 Empirical Timings

In the computational model for TRMM, we need to know the time (t_T) it takes for a *trmm*μ on a block sized TRMM and the time (t_M) it takes for a *gemm*μ on a block sized GEMM. Recall that during our search for *trmm*μ and the *gemm*μ for our *gemm*μ-based TRMM, we measure the performance of these all the microkernels anyway to find the best ones for the system. We can save those timing results during the search and later use them as t_T and t_M respectively for different block factors. Note that proper timing of a microkernel is vital to have a good estimate of performance [70]. ATLAS uses various timing techniques depending on whether the input problem can fit into some level of the system's cache or not. We can use the same approach to time the microkernels for two categories of problems: (1) the problem fits and (2) the problem does not fit in the system's cache. During the selection of block factors using our model, we will determine whether the problem is small enough to fit in the cache or not and use the appropriate timing data to estimate the performance.

B.4 Computational Model for *gemmμ*- and *trsmμ*-based TRSM

For our *gemmμ*- and *trsmμ*-based triangular solve (TRSM), we need two searches as well for the best performance:

1. Finding an optimized *gemmμ* with a restriction of $B_M = B_K$ for left variants or $B_N = B_K$ for right-variants, where B_M, B_N, and B_K are block factors for the M, N, and K dimensions respectively and

2. Finding a combination of u_m, u_n, and u_k so that our TRSM microkernel can utilize an optimized *gemmμ* for best performance. For *trsmμ*, we have a restriction of u_m to be a multiple of u_k for left variants, or u_n to be a multiple of u_k for right variants, where u_m, u_n, and u_k are unroll factors for M, N, and K dimensions.

Like TRMM, due to *trsmμ* performing a small portion of the total number of required computation, we can afford to select one combination of u_m, u_n, and u_k and then use it for any block-sized TRSM. However, as discussed before, selecting only one combination of *gemmμ* and the block factors is not always optimal. Therefore, we have the same problem as TRMM that we need to select one combination of the block factors B_M, B_N, and B_K for a given input problem for performing the solve. To select the combination for best performance, we will develop a computational model for our *gemmμ*-based TRSM.

B.4.1 FLOP Count Analysis

Consider the example of LLNN-TRSM in the equation below, where $M = 3$, $N = 2$ and $alpha = 1$.

$$\begin{bmatrix} A_{11} & 0 & 0 \\ A_{21} & A_{22} & 0 \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \\ X_{31} & X_{32} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix} \tag{B.21}$$

Recall that the solve of the first column of X requires the steps shown in Figure 2.24. Note that for X_{11}, we only need one division. For X_{21}, we need one multiplication, one subtraction

and one division and for X_{31}, we need two multiplications, two subtractions and one division. In general, for i-th row element, we need $(i-1)$ multiplications, $(i-1)$ subtractions and 1 division. The same number of computation is needed for each column.

$$\begin{aligned}\text{Number of FLOPs in LLN-TRSM of size } M \times N &= \sum_{j=1}^{N}\sum_{i=1}^{M}(i-1)+(i-1)+1 \\ &= NM^2\end{aligned} \tag{B.22}$$

Like TRMM, LUN-variant of TRSM is symmetric to the LLN-variant, thus requiring the same number of FLOPs.

Now consider the RUN-variant of TRSM shown in the equation below, where $M = 2$, $N = 3$ and $alpha = 1$. For the right-variants, since we process one row of X at a time, we start with X_{11}. For X_{11}, we only need one division, for X_{12}, we need one multiplication, one subtraction and one division. In general, for j-th column, we need 1 division, $(j-1)$ multiplications and $(j-1)$ subtractions. The same number of FLOPs is needed for each row.

$$\begin{aligned}\text{Number of FLOPs in RUN-TRSM of size } M \times N &= \sum_{i=1}^{M}\sum_{j=1}^{M}(j-1)+(j-1)+1 \\ &= MN^2\end{aligned} \tag{B.23}$$

Due to being symmetric, RLN-variant of TRSM requires the same number of FLOPs as the RUN-variant.

As shown in Section B.3 for GEMM-based TRMM, we can similarly show that GEMM-based TRSM requires the same number of FLOPs as the unblocked TRSM. Next, we will analyze our *gemmμ*-based approach. Note that unlike TRMM, we do not pad the triangular blocks for our *trsmμ*. As a result, the number of FLOPs required should be the same as the unblocked TRSM. However, recall that we store the inverted diagonal during copy and multiply with inverted diagonal elements to avoid repeated division operation. This incurs B_M or B_N extra FLOPs for left or right variants, respectively.

$$\begin{aligned}\text{Number of FLOPs in Left-}trsm\mu &= B_N B_M^2 + B_M \\ \text{Number of FLOPs in Right-}trsm\mu &= B_M B_N^2 + B_N\end{aligned} \tag{B.24}$$

If we assume N_{rp} to be M/B_M, i.e. the number of B_M-sized row-panels and $N_{cp} = N/B_N$ i.e. the number of B_N-sized column panels, the total number of FLOPs done by the *trsm*μ is given below:

$$\begin{aligned}\text{Total Left-}trsm\mu\text{ FLOPs} &= N_{cp} \times N_{rp} \times (B_N B_M^2 + B_M) \\ &= \left(\frac{N}{B_N}\right)\left(\frac{M}{B_M}\right) B_M(B_N B_M + 1) \\ &= \left(\frac{MN}{B_N}\right)(B_N B_M + 1)\end{aligned} \tag{B.25}$$

We can use a similar analysis to that of TRMM to see that for i-th B_M-sized row-panel, we need $(i-1)$ calls to the *gemm*μ of size B_M, B_N, B_M. The total number of calls to *gemm*μ is given below:

$$\begin{aligned}\text{Total number of calls to }gemm\mu\text{ for Left-variants} &= \sum_{j=1}^{N_{cp}} \sum_{i=1}^{N_{rp}} (i-1) \\ &= \frac{1}{2} N_{cp} N_{rp}(N_{rp} - 1) \\ &= \frac{1}{2}\left(\frac{N}{B_N}\right)\left(\frac{M}{B_M}\right)\left(\frac{M}{B_M} - 1\right)\end{aligned} \tag{B.26}$$

Each of these calls to *gemm*μ performs $2 \times B_M \times B_N \times B_M$ FLOPs. Therefore, the total number of FLOPs done by *gemm*μ is given below:

$$\begin{aligned}\text{Total }gemm\mu\text{ FLOPs for Left-variant} &= \frac{1}{2} N_{cp} N_{rp}(N_{rp} - 1) \times 2 \times B_M \times B_N \times B_M \\ &= \left(\frac{N}{B_N}\right)\left(\frac{M}{B_M}\right)\left(\frac{M}{B_M} - 1\right) B_M^2 B_N \\ &= \left(\frac{N}{B_N}\right)\left(\frac{M}{B_M}\right)\left(\frac{M - B_M}{B_M}\right) B_M^2 B_N \\ &= NM(M - B_M)\end{aligned} \tag{B.27}$$

B.4.2 Building the Computational Model

If we assume that t_S is the time taken by one *trsm*μ for an $B_M \times B_N$ sized problem and t_M is the time taken by one *gemm*μ for a GEMM of size B_M, B_N, and B_M. Using these two parameters, we can build a computational model to estimate the amount of time that will

be needed to perform a TRSM operation using our *gemmμ*-based approach, as shown below:

$$\begin{aligned}\text{Total Time} &= N_{cp} \times (N_{rp} \times t_S + \frac{1}{2} N_{rp}(N_{rp}-1) \times t_M) \\ &= \left(\frac{N}{B_N}\right)\left(\frac{M}{B_M} t_S + \frac{M(M-B_M)}{2B_M^2} t_M\right)\end{aligned} \tag{B.28}$$

We can use this equation for a given problem (i.e. M and N are known) to estimate the performance of using a certain combination of block factors (B_M and B_N) and choose the best one based on our estimated performance. We can do a similar analysis to build model for each variant of TRSM. The only unknown factors in our model (Equation B.28) are t_S and t_M that varies depending on the system.

As described in Section B.3.4 for the computational model of TRMM, we can save the timing results during *gemmμ* and *trsmμ* search for estimation of t_M and t_S respectively. Note that like TRMM, in our computational model for TRSM, we ignored the cost of copying the blocks which can later be included for improved predictions.

B.5 Computation Model for Serial Cholesky Factorization

For our *gemmμ*-based serial Cholesky factorization, we need one search to find the best *gemmμ* and a set of block factors with the restriction of square block factors i.e. $B_M = B_N = B_K$. Note that it may be possible to apply non-square blocking for Cholesky but in our research, we chose square block factors to simplify implementation. We will simply use B_N to denote this square block factor.

B.5.1 FLOP Count Analysis

The minimal FLOP count for Cholesky factorization on a matrix of size $N \times N$, as given in [4], is shown below:

$$\text{Total Cholesky FLOPs} = \frac{1}{3}N^3 + \frac{1}{2}N^2 + \frac{1}{6}N \tag{B.29}$$

Recall that a Cholesky factorization involves four major operations: a) block factorization b) TRSM c) SYRK and d) GEMM. However, in our *gemmμ*-based approach, we are using *gemmμ* to perform SYRK which requires extra/useless FLOPs. The total number of extra FLOPs done can be derived as follows:

1. Minimal SYRK FLOP count for an $(N \times K)$-sized input for $\alpha = -1, \beta = 1$ given in [4] as:

$$\text{Minimal SYRK FLOPs} = KN(N+1) \tag{B.30}$$

2. In Cholesky factorization, SYRK is performed with $N = B_N$ and $K = (j-1)B_N$ for column panel j. If we assume that SYRK operation is performed for $B_N \times B_N$-sized inputs, we need to call the SYRK operation $j-1$ times for column panel j. If $N_p = \frac{N}{B_N}$ is the number of column panels, then

$$\begin{aligned} \text{Block-sized SYRK calls} &= \sum_{j=1}^{N_p}(j-1) \\ &= \frac{N_p(N_p-1)}{2} \\ &= \left(\frac{N}{2B_N}\right)\left(\frac{N}{B_N}-1\right) \end{aligned} \tag{B.31}$$

3. For the number of FLOPs performed by each block-sized SYRK calls can be computed using Equation B.30 as follows:

$$\text{Block-sized SYRK FLOPs} = B_N^2(B_N+1) \tag{B.32}$$

4. Since we call *gemmμ* to perform SYRK and each *gemmμ* operates on block-sized inputs with $\alpha = -1, \beta = 1$:

$$\text{Block-sized } \textit{gemm}\mu\text{-based SYRK FLOPs} = 2B_N^3 \tag{B.33}$$

5. We can compute the number of useless FLOPs performed by each *gemmμ* calls by subtracting Equation B.32 from Equation B.33 as shown below:

$$\begin{aligned} \text{Block Useless FLOPs} &= 2B_N^3 - B_N^2(B_N+1) \\ &= B_N^3 - B_N^2 \\ &= B_N^2(B_N-1) \end{aligned} \tag{B.34}$$

6. We can multiply Equation B.34 and B.31 to compute the total number of useless FLOPs performed by our *gemmμ*-based Cholesky factorization as shown below:

$$\begin{aligned}\text{Total Useless FLOPs} &= B_N^2(B_N-1)\left(\frac{N}{2B_N}\right)\left(\frac{N}{B_N}-1\right)\\ &= B_N^2(B_N-1)\left(\frac{N}{2B_N}\right)\left(\frac{N-B_N}{B_N}\right)\\ &= \frac{1}{2}N(B_N-1)(N-B_N)\end{aligned} \tag{B.35}$$

Recall that the latest ATLAS release provides SYRK microkernels which may reduce the number of useless FLOPs but may require copying the same data blocks to different storage formats required by the *gemmμ* and the SYRK microkernel. Depending on the problem size, we may choose to either do the extra FLOPs or do the extra copy.

B.5.2 Building the Computational Model

If T_f is the total amount of time taken by all block factorization steps, T_s is the total amount of time taken for TRSM, and T_m is the total amount of time taken by GEMM and SYRK using *gemmμ*.

$$\text{Total Time for Cholesky Factorization}, T = T_f + T_s + T_m \tag{B.36}$$

Recall that the block factorization step is done once for every column-panel. If N_p is the number of row/column-panels (i.e. $N_p = \frac{N}{B_N}$), the total time for factorizing the blocks is given below:

$$\begin{aligned}\text{Total Factorization Time}, T_f &= N_p\ t_f\\ &= \left(\frac{N}{B_N}\right) t_f\end{aligned} \tag{B.37}$$

t_f is the time taken by an $(B_N \times B_N)$-sized Cholesky factorization. We can estimate the value of t_f by timing the block factorization used in our Cholesky factorization.

Recall that on the first column-panel, TRSM is called on block-sized inputs for $N_p - 1$ times. On the second column-panel, TRSM is called on block-sized inputs for $N_p - 2$ times

and so on.

$$\begin{aligned}\text{Total TRSM Time}, T_s &= \sum_{j=1}^{N_p} (N_p - j)\ t_s \\ &= \frac{1}{2} N_p (N_p - 1)\ t_s \\ &= \left(\frac{N}{2B_N}\right)\left(\frac{N}{B_N} - 1\right) t_s\end{aligned} \tag{B.38}$$

t_s is the time taken by an $(B_N \times B_N)$-sized TRSM. We can similarly estimate t_s by timing ATLAS's TRSM routine on $(B_N \times B_N)$-sized inputs.

For GEMM, on the first panel, we have zero *gemmμ* calls. On the second column-panel, one *gemmμ* is called for $N_p - 1$ blocks. On the third column-panel, two *gemmμ* is called for $N_p - 2$ blocks and so on.

$$\begin{aligned}\text{Total } \textit{gemm}\mu \text{ Time}, T_m &= \sum_{j=1}^{N_p} ((j-1)(N_p - j + 1))\ t_m \\ &= \frac{1}{6}(N_p^3 - N_p)\ t_m \\ &= \frac{1}{6}\left(\frac{N^3}{B_N^3} - \frac{N}{B_N}\right) t_m\end{aligned} \tag{B.39}$$

t_m is the time taken by an $(B_N \times B_N)$-sized *gemmμ*. t_m can be estimated using the timing data from the *gemmμ* search for Cholesky factorization. As before, we can have two variants for each of the estimated values t_f, t_s, and t_m: for in-cache and out-of-cache problems.

B.6 Computation Model for Serial LU Factorization

Recall that in an LU factorization, the major steps are panel factorization, applying row-swaps (pivoting), TRSM and GEMM updates using *gemmμ*. The total time taken by the complete LU factorization, $T = T_f + T_a + T_s + T_m$, where T_f is the total amount of time taken by all panel factorization steps, T_a is the total amount of time taken for applying row-swaps, T_s is the total amount of time taken for TRSM, and T_m is the total amount of time taken by GEMM. Recall that in our *gemmμ*-based serial LU factorization, all the operations except the panel factorization are done on block-sized data at a time.

Since a panel factorization is itself an LU factorization (unblocked) on an B-sized column-panel, we need to analyze the panel factorization in detail to build a model. An unblocked panel factorization is done with the steps: a) finding pivot with IAMAX b) row-swaps using SWAP and c) rank-1 update using GER. We will estimate the time taken by each of these components for estimating the panel factorization time for an $(M \times B)$-sized panel.

IAMAX: As discussed in Section 3.1, for an $(M \times B)$-sized panel, we need an IAMAX on each column with size M, $M-1$, $M-2$, and so on. An IAMAX operation with size M requires $M-1$ comparisons to find the pivot element. Therefore:

$$\begin{aligned} \text{Total IAMAX comparisons} &= \sum_{j=1}^{B}(M-j-1) \\ &= \frac{1}{2}(2MB - B^2 - B) \end{aligned} \tag{B.40}$$

$$\text{Time for IAMAX}, T_i(M,B) = \sum_{j=1}^{B} t_i(M-j+1) \tag{B.41}$$

where $t_i(x)$ is the time needed for IAMAX on an input of size x.

Note that to estimate these IAMAX times for all possible sizes is not feasible. Rather we can time IAMAX for some pre-selected sizes and use interpolation or extrapolation as needed to estimate the time for any other input sizes. Note that an alternative approach to estimate $T_i(M,B)$ would be to estimate the time for an average case, and use that for all columns (instead of estimating all possible ti i.e. $ti(M)$, $ti(M-1)$, etc.). In that case, $T_i(M,B) = B \times t_i(M - B/2)$

SWAP: In the worst case, we have B element swaps for each column. If the diagonal element of the current column is already the pivot element, no swap is necessary. Since this is rarely true for a uniformly random input, we will consider the worst case in our analysis.

$$\text{Total SWAPs} = \sum_{j=1}^{B} B = B^2 \tag{B.42}$$

$$\text{Time for SWAP}, T_{sw}(M,B) = B\ t_{sw}(B) \tag{B.43}$$

where $t_{sw}(B)$ is the time needed to swap B elements. We can time the SWAP operation for each available Bs to estimate $t_{sw}(B)$.

GER: After each swap, we need to apply GER on the rest of the matrix. The GER operations are of size $(M-j) \times (B-j)$ after performing SWAP for the j-th column where $i < B$ since no GER is needed after B-th swap.

$$\text{Total GER FLOPs} = \sum_{j=1}^{B} (M-j) \times (B-j) \tag{B.44}$$

$$\text{Total GER time}, T_g(M,B) = \sum_{j=1}^{B} t_g(M-j, B-j) \tag{B.45}$$

where $t_g(x,y)$ is the time needed to perform a GER on $(x \times y)$-sized input. As with IAMAX, timing GER for all possible inputs is not feasible. Therefore, we can time GER for some pre-selected input sizes and use interpolation or extrapolation if needed to estimate time for each of these GER operations or use an average estimation as with IAMAX. In that case, $T_g(M,B) = B\ t_g(M - B/2, B/2)$.

Using the above equations, we can estimate the time for panel factorization on $(M \times B)$-sized input.

$$\text{Time for one panel factorization}, t_f(M,B) = T_i(M,B) + T_{sw}(M,B) + T_g(M,B) \tag{B.46}$$

Recall that the height (M) of the panel factorization decreases as we progress for successive column-panels in the outer LU factorization. Therefore the total time for all panel factorization can be computed as:

$$\text{Total time for panel factorization}, T_f(N) = \sum_{j=0}^{N_p} t_f(N + B - jB, B) \tag{B.47}$$

where $N_p = N/B$ i.e. the number of column-panels in the input.

After each panel factorization, we move on to the next column-panel of the matrix and apply the necessary updates on it before it can be factorized using our selected panel factorization method. These updates include pivoting rows and applying TRSM on $(B \times B)$-sized

blocks. For column-panel j, we apply pivot and TRSM on $j-1$ blocks (reading data from previously factorized $j-1$ panels on the left) where $1 \leq j \leq N_p$.

$$\text{Total pivot time}, T_p = \sum_{j=1}^{N_p} (j-1) \times t_p(B,B) \tag{B.48}$$

$$\text{Total TRSM time}, T_s = \sum_{j=1}^{N_p} (j-1) \times t_s(B,B) \tag{B.49}$$

Both pivot time ($t_p(B,B)$) and TRSM time ($t_s(B,B)$) for block sized input can be estimated through prior timing of these operations for all available block factors.

Recall that a series of *gemmμ* is performed after each TRSM operation is done on a certain panel. The number of calls to *gemmμ* for column-panel j is $\sum_{i=1}^{j-1}(Np-i)$.

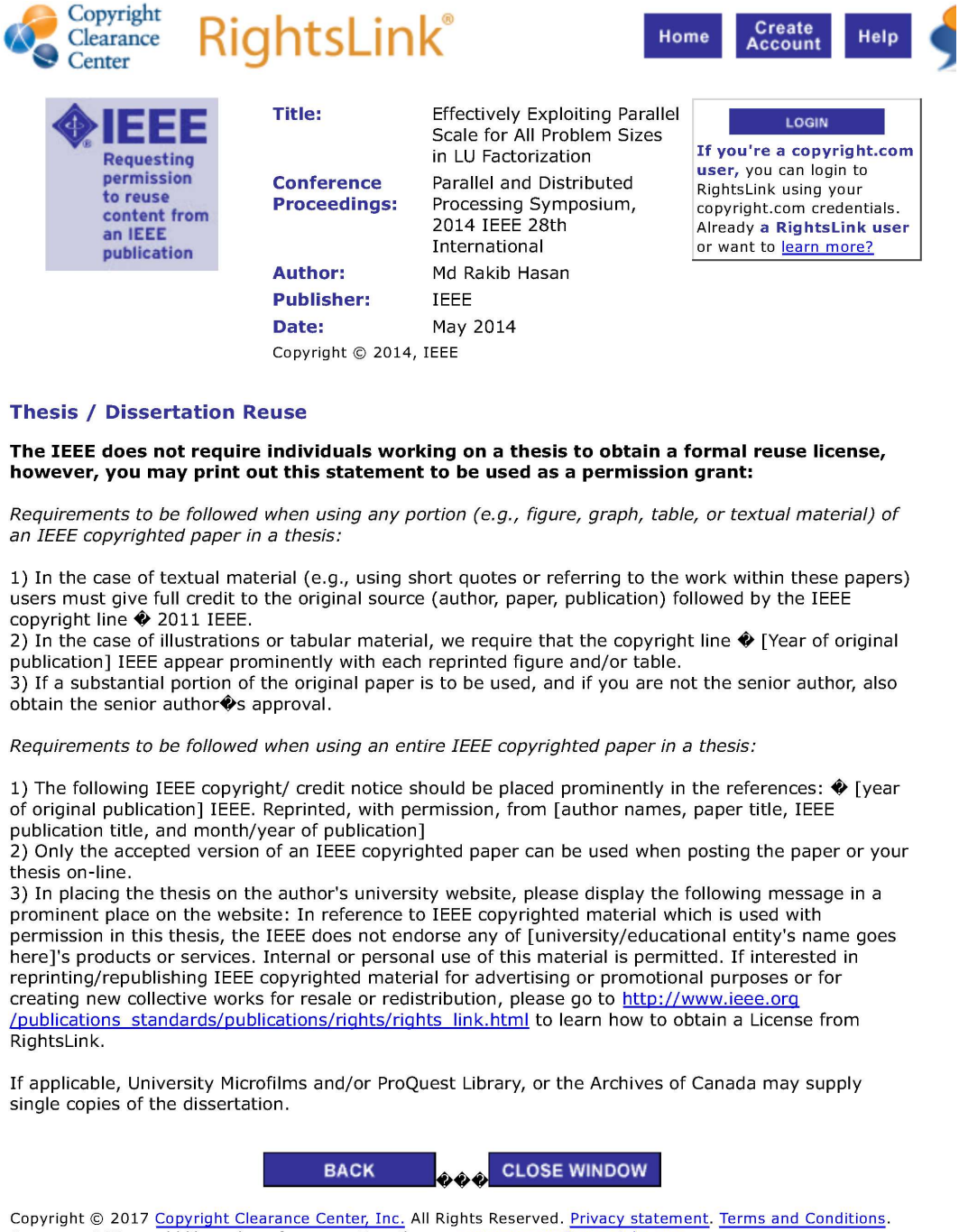
$$\text{Total number of calls to } \textit{gemm}\mu = \sum_{j=1}^{N_p} \sum_{i=1}^{j-1} (Nb-i) \tag{B.50}$$

$$\text{Total time for GEMM updates}, T_m = \sum_{j=1}^{N_p} \sum_{i=1}^{j-1} (Nb-i) \times t_m(B,B,B) \tag{B.51}$$

$t_m(B,B,B)$ is the time taken by one *gemmμ* call on block-sized square input and this estimated time is already saved by the *gemmμ* framework for each available B. Using the above equations, we can estimate the time for LU factorization for each available B given an input problem and select the best B for performing the factorization.

APPENDIX C
COPYRIGHT PERMISSIONS

Copyright permissions for the previously published materials are provided here.



Copyright Clearance Center RightsLink®

Home | Create Account | Help

IEEE Requesting permission to reuse content from an IEEE publication

Title: Effectively Exploiting Parallel Scale for All Problem Sizes in LU Factorization

Conference Proceedings: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International

Author: Md Rakib Hasan

Publisher: IEEE

Date: May 2014

Copyright © 2014, IEEE

LOGIN

If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials. Already **a RightsLink user** or want to learn more?

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACK CLOSE WINDOW

Copyright © 2017 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

Figure C.1: Reuse permission from IEEE for the paper on LU factorization

CONSENT AND RELEASE

In consideration of my participation in the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), I grant to the Association for Computing Machinery ("ACM"):

1. The unlimited, worldwide, irrevocable right to use, distribute, publish, exhibit, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) my presentation and comments at the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15); (b) any videotaped interviews of me; and (c) any materials, including written, audio and visual works that I submit for use in connection with the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15) (collectively, the "Materials").

2. The rights granted include the transcription and reproduction of the Materials for inclusion in products sold or distributed by ACM and live broadcast of my presentations at the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).

3. In connection with the rights granted in Section 1, I further grant ACM the unlimited, worldwide, irrevocable right to use my name, picture, likeness, voice and biographical information in connection with the advertisement, distribution and sale of products by ACM, and release ACM from any claim based on right of privacy or publicity.

4. Where necessary, I have obtained all third party permissions and consents to grant the rights above and have provided copies of such permissions and consents to ACM.

5. Except for the rights expressly granted to ACM above, I retain ownership of the intellectual property rights in the Materials.

Enter your Submission type and ID (e.g., Tutorial, tut101): post134s1

Submission Title: Reliable Performance Auto-Tuning in Presence of DVFS

Author(s) PRINTED: Md Rakib Hasan, Eric Van Hensbergen, Wade Walker

Author(s) SIGNED: MD RAKIB HASAN, ERIC VAN HENSBERGEN, WADE WALKER

Return this form through the Submission Consent and Release Form
Submission Form available at https://submissions.supercomputing.org/

Figure C.2: Consent form submitted to ACM: poster and extended abstract ownership retained by author

VITA

Md Rakib Hasan was born in Dhaka City, Bangladesh. He finished his undergraduate studies in Computer Science and Engineering at Bangladesh University of Engineering and Technology in March 2009.

After working at a software development company for more than a year, Rakib came to United States of America to pursue his graduate studies at University of Texas at San Antonio. In 2013, he transferred to Louisiana State University to continue his research with Dr. R. Clint Whaley. He is currently a candidate for the degree of Doctor of Philosophy in Computer Science, which will be awarded in August 2017.