Proceedings of the International Multiconference on Computer Science and Information Technology pp. 303–310
ISBN 978-83-60810-14-9
ISSN 1896-7094

Empirically Tuning LAPACK's Blocking Factor for Increased Performance

R. Clint Whaley
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
Email: whaley@cs.utsa.edu

***Abstract*—LAPACK (Linear Algebra PACKage) is a statically cache-blocked library, where the blocking factor (NB) is determined by the service routine ILAENV. Users are encouraged to tune NB to maximize performance on their platform/BLAS (the BLAS are LAPACK's computational engine), but in practice very few users do so (both because it is hard, and because its importance is not widely understood). In this paper we (1) Discuss our empirical tuning framework for discovering good NB settings, (2) quantify the performance boost that tuning NB can achieve on several LAPACK routines across multiple architectures and BLAS implementations, (3) compare the best performance of LAPACK's statically blocked routines against state of the art recursively blocked routines, and vendor-optimized LAPACK implementations, to see how much performance loss is mandated by LAPACK's present static blocking strategy, and finally (4) use results to determine how best to block nonsquare matrices once good square blocking factors are discovered.**

I. Introduction

LAPACK [1] (Linear Algebra PACKage) is one of the most widely-used computational libraries in the world. LAPACK is the successor to the highly successful LINPACK[2] and EISPACK[3] packages. The main difference between LAPACK and its ancestor packages is that it blocks for the cache, which allows it to run orders of magnitude faster on modern architectures. Since linear algebra is computationally intensive, it is important that these operations, which are inherently very optimizable, run as near to machine peak as possible. In order to allow for very high performance with a minimum of LAPACK-level tuning, LAPACK does the majority of its computation by calling a lower-level library, the BLAS (Basic Linear Algebra Subprograms). The BLAS are in turn split into three "levels" based on how much cache reuse they enjoy, and thus how computational efficient they are. In order of efficiency, they are: Level 3 BLAS[4], which involve matrix-matrix operations that can run near machine peak, Level 2 BLAS[5], [6] which involve matrix-vector operations and are considerably slower, and Level 1 BLAS[7], [8], which involve vector-vector operations, and usually run at the speed of memory.

In order to maximize performance, LAPACK is therefore formulated to call the Level 3 BLAS whenever possible. LAPACK's main strategy for accomplishing this can be understood by examining its factorizations routines, QR, LU, and Cholesky. For each of these factorizations, LAPACK possesses two implementations: the first is an unblocked implementation that calls the Level 1 and 2 BLAS exclusively, which results in low performance. These routines are called DGEQR2/DGETF2/DPOTF2, respectively for QR/LU/Cholesky (assuming double precision). To get better performance, LAPACK has statically blocked versions of these routines (DGEQRF/DGETRF/DPOTRF), which calls the unblocked code on a subsection of the matrix, and then updates the rest of the matrix using the Level 3 BLAS (especially matrix multiply, DGEMM). The idea is that the updating of the rest of the matrix will run near peak when using a well-tuned BLAS implementation, and that this cost will dominate the unblocked factorization cost enough to allow LAPACK to be extremely computational efficient on any cache-based architecture.

Because LAPACK is such a large and complex library, allowing almost all the system tuning to occur in a smaller kernel library like the BLAS is critical, since LAPACK is too large for hand-tuners to make the individual routines computationally efficient. Through careful design, almost all platform-specific tuning has been moved to the BLAS, leaving LAPACK to concentrate on algorithm development. However, LAPACK possesses one key parameter that must be tuned to the platform to achieve maximal performance, which is the blocking factor (NB) used in LAPACK's statically blocked routines. To understand this further, we quickly overview the algorithm of DGETRF, which operates on an $M \times N$ matrix A: DGETF2 is called to factor an $M \times$ NB column panel of A into L and U, then an NB $\times (N - NB)$ row panel is updated using the Level 3 BLAS routine DTRSM, and these two results are then used to update the entire $(M - NB) \times (N - NB)$ trailing submatrix using DGEMM. This process is repeated, with the M and N above shrinking by NB each time, until the entire matrix has been factored. This type of blocked operation is roughly what occurs in a great many of LAPACK routines, including QR and Cholesky, though the details of the exact partitionings and routines used in updates vary widely.

NB cannot be chosen arbitrarily if good performance is to be maintained. NB=1 results in unblocked operation, and so defeats the whole purpose of blocked algorithms. Increasing NB typically allows the Level 3 updates to become more efficient,

This work was supported in part by National Science Foundation CRI grant SNS-0551504

as these operations have $O(N^3)$ computations which dominate the $O(N^2)$ memory references as N is increased. However, we cannot set NB $= N$, since this would wind up calling the unblocked code for the whole algorithm (no Level-3 updates left to do). Choosing a good NB is therefore a balancing act: making it large typically improves Level 3 performance, but as it grows, the unblocked code's performance becomes an increasing percentage of computational time, resulting in reduced performance. Therefore, the best NB is essentially unknowable a priori: it depends on the LAPACK routine being tuned, on the architecture being used, the relative performance of the Level 1 and 2 BLAS vs. Level 3 for the particular BLAS library being used, etc. The designers of LAPACK were aware of this, and so they did not hardwire the NB into the algorithm. Rather, most blocked LAPACK routines query a service routine called ILAENV for what blocking factor should be used. The LAPACK documentation stresses that users should tune this blocking factor in ILAENV for their problems of interest during install. Unfortunately, almost no users do so in practice: most are unaware that they should, and the remainder find it too hard to do. Therefore, the majority of LAPACK installations use the default blocking factors, which we show can yield significantly lower performance.

We have developed a framework for auto-tuning NB for LAPACK in order to fix this problem. This framework is part of our empirical tuning package, ATLAS [9], [10], [11]. ATLAS (Automatically Tuned Linear Algebra Package) provides empirically-tuned BLAS, and a few improved LAPACK routines. This new LAPACK tuning mechanism is part of ATLAS's empirical tuning framework, but it can be used to tune LAPACK for any BLAS, as this paper will describe.

The remainder of this paper is organized in the following way: Section II provides a quick overview of the empirical NB-tuning framework and service routines we have developed; Section III then presents results from empirically tuning LAPACK's factorization routines. We show that this can make a large difference in performance for a variety of platforms and BLAS implementations, and that these results point to a simple algorithmic change that could improve performance still further. Finally, Section IV summarizes our findings, and discusses future work.

II. ATLAS'S EMPIRICAL TUNING FRAMEWORK FOR LAPACK

In this section we briefly overview the tools we developed for this investigation, as other BLAS users and researchers may find them useful. There are two main routines of interest, both of which appear in ATLAS's ATLAS/bin directory (assuming ATLAS version 3.9.2 or later).

The main tool is an LAPACK timer that can empirically tune NB for a variety of LAPACK routines, ATLAS/bin/lanbtst.c. This general-purpose timer links in a specialized ILAENV that allows the timer to change the ILAENV return value as part of the timing process. It can time arbitrary problem sizes, using arbitrary blocking factors, and be built against any mixture of (system LAPACK, ATLAS LAPACK, netlib LAPACK) + (system BLAS, ATLAS BLAS, F77BLAS). We presently support timing the three LAPACK factorizations: GETRF (LU), POTRF (Cholesky) and GEQRF/GERQF/GEQLF/GELQF (QR); adding additional routines is straightforward. Therefore, this timer/tuner can be used in a variety of ways to investigate performance, but here we use it to tune the blocking factor as N changes. When used in this way, it generates either C macros or FORTRAN77 functions that take in the problem size, and return the best NB found by the empirical tuning for a problem of comparable size. The user can control what sizes of problems to tune, what range of NBs to try, how many times to repeat each timing, how many FLOPS are necessary to get reliable timings, etc. This timer automatically flushes the caches for realistic out-of-cache performance results, as described in [12]

To build the various LAPACK/BLAS combinations, the user fills in ATLAS's Make.inc BLASlib (system BLAS), FLAPACKlib (Fortran77/netlib LAPACK), and SLAPACKlib (system LAPACK). For instance, to tune LAPACK for the GotoBLAS installed in /opt/lib, you would fill in:

```
BLASlib = /opt/lib/libgoto_opteronp-r1.26.a \
          $(CBLASlib) $(ATLASlib)
FLAPACKlib = /opt/lib/libreflapack.a
SLAPACKlib = /opt/lib/libgoto_opteronp-r1.26.a \
             $(FLAPACKlib)
```

The LAPACK tuner's base name is x<pre>lanbtst, where <pre> is replaced by the precision prefix (eg., xdlanbtst for double precision). To this base name add a suffix indicating what variation should be timed. The suffix looks like: 'F_[f,s]l_[f,s,a]b. The first [f,s] before the 'l' chooses what LAPACK to use: fortran or system. The second part of the suffix determines what BLAS are linked in (thus the 'b' at the end): 's' links in the system BLAS, 'f' links in the Fortran BLAS (defined in the macro FBLASlib), and 'a' links in the ATLAS BLAS. To build a timer for ATLAS LAPACK with ATLAS BLAS, the make target is simply: 'xdlanbtst'. Thus, to build a tuner for the FORTRAN77 reference LAPACK linked to the GotoBLAS (assuming the above BLASlib setting), we would issue make xdlanbtstF_fl_sb. Calling the LAPACK tuner with invalid arguments or --help prints usage information.

The other tool of interest can be built with 'make xstattime'. This program reads in output from various ATLAS timers, and provides the results in a spreadsheet-friendly format. When timings are repeated multiple times, it also provides some basic statistical information (eg., average, max, min, standard deviation, etc.). If xstattime is used to compare two different runs, it uses a Student's T-Test [13] to compare whether the two means, from distributions with possibly different variances, are statistically different or not (this last feature requires a high number of samples to be reliable). Again, --help will provide more detailed usage information.

III. RESULTS

A. Experimental Methodology

Version information: `gcc/gfortran 4.2`, ATLAS3.9.2 (labeled **atl**), ACML4.1.0 (**acm**), and GotoBLAS v2.91 (**got**). All performance results are gathered with a timer that flushes the caches, as described in [12]; therefore the small-case performance here may be much lower than those typically published, but more indicative of most real-world performance. Since MFLOP results are easier to understand in terms of peak performance, we report MFLOP rates rather than time. This could be problematic, in that some LAPACK routines (eg. GEQRF) do extra FLOPS as `NB` grows. In order to avoid this kind of problem, we compute the FLOP count for all algorithms as if they used the unblocked code, regardless of the blocking factor actually used. Therefore, these MFLOP results may be converted to time by simply by dividing by the FLOP count of the algorithm and inverting. The FLOP counts for (Cholesky, LU, QR) are very roughly $(\frac{N^3}{3}, \frac{2N^3}{3}, \frac{4N^3}{3})$, assuming square matrices. The exact FLOP counts can be found in LAPACK's `dopla.f`[14] routine.

Our main results are for two machines: a 2.4Ghz Core2Duo (abbreviated as **c2d**) and 2.2Ghz AMD Athlon-64 X2 (**ath**), both of which were running Kubuntu 8.04 (2.6 Linux kernel). On the **c2d**, we provide results for ATLAS[11] and Goto[15] BLAS: we are unable to provide results for Intel's MKL[16], since their libraries are not freely available for academic use. On the AMD system, we provide results for ATLAS, GotoBLAS, and AMD's ACML[17]. For both these machines, we time 64-bit libraries (which tend to have better performance than 32-bit). Unless otherwise specified, all timings were taken with a cycle-accurate wall-time, all results are the best of eight runs, with small problems being repeated within the timing interval to avoid resolution problems, as described in [12]. We also present a small subset of results for a 900Mhz Itanium 2 system. The Itanium is not featured more prominently because we only have ATLAS results on this machine: MKL was not available, and the GotoBLAS seg faulted during installation when compiled with `gcc 4.2` (we know GotoBLAS install without error with `gcc 3.x` on this platform, but were unable to install `gcc 3.x` on the Itanium in time for this paper). As it is, the limited Itanium results are provided mainly to demonstrate that this technique is in no way x86-specific. The Itanium timings use Linux's `gettimeofday` as the system timer.

B. Tuning Speedups and Performance Overview

Figure 1 shows the performance in MFLOP for the appropriate BLAS's `DGEMM`, and netlib LAPACK's factorizations, using various BLAS. For each problem size in each figure, the first three bars are timings on the **ath**, while the last two are on **c2d**. The first column (white with black horizontal bars) uses ACML BLAS on **ath**, the second column (solid blue/gray) uses ATLAS BLAS on **ath**, the third column (slanting cross-hatching) uses GOTO BLAS on **ath**, the fourth column (textured solid blue/gray) uses ATLAS BLAS on **c2d**, and the final column (square cross-hatching) uses GotoBLAS on **c2d**.

The DGEMM timings of Figure 1(a) should provide an estimate on the peak performance the respectively libraries can achieve on these platforms. The Core2Duo has twice the theoretical peak of the Athlon-64, and so we see that the **c2d** is considerably faster regardless of the library. On the **ath**, the libraries' DGEMM performance is roughly equal, with ATLAS trailing ACML and GotoBLAS slightly (roughly 3%) for large problems. On the Core2Duo, the GotoBLAS have a noticable advantage, particularly for small problems.

Figures 1(b-d) show the performance of `DGEQRF` (QR), `DGETRF` (LU), and `DPOTRF` (Cholesky), respectively. Each of these timings is for the netlib LAPACK's factorization routine using the default blocking factor returned by the stock `ILAENV` (`NB` = 32, 64, and 64, respectively). It is the performance of these routines that we show can be improved with an empirical tuning of `NB`. On the **ath**, all libraries have similar performance, with the GotoBLAS doing generally better for small problems. We will see that tuning for the blocking strongly improves small-case performance, particularly for `ATLAS` and `ACML`, which are weak for small problems when using the default `NB`.

We then used the ATLAS infrastructure to tune the blocking factor for each operation on each architecture. We don't have room for full results, but blocking factors for all `DGEQRF` and some `DPOTRF` settings are shown in Table I. We give full results for `DGEQRF`, because neither the ATLAS nor GotoBLAS libraries supply an optimized version of this routine (they *do* supply optimized `DGETRF` and `DPOTRF`, as we will see in Section III-D). Therefore, the LAPACK version we tune here is actually the best one available when using these libraries. The `DGETRF` results are very similar to those for QR, but `DPOTRF` is very different, so we show full `DPOTRF` results for the Athlon-64 architecture.

For QR, the general trend is that the larger the problem, the larger the best blocking factor, up until a maximum is reached. This maximum represents the point at which the DGEMM performance gain from further increasing `NB` is offset by the performance loss due to spending greater time in the unblocked code. If we can optimize the unblocked part of the algorithm (a promising approach for doing this is outlined in Section III-C), then we can almost certainly increase performance further by using larger blocking factors for large problems.

`DPOTRF` looks very different, in that `NB` has apparently stopped growing only for ACML. One advantage Cholesky has over LU is that it doesn't require pivoting, which makes the unblocked portion of the algorithm much cheaper. In LU or QR, vastly increasing `NB` will speed up DGEMM, but the corresponding slowdown in the unblocked code will make it a net loss. Since the unblocked cost is relatively trivial in Cholesky, the block factor is free to grow much larger before the unblocked costs become prohibitive. There is further pressure in Cholesky for larger blocking factors: Cholesky's performance is strongly affected not only by DGEMM, but

TABLE I
OPTIMIZED NBS FOUND BY ATLAS'S EMPIRICAL TUNING STEP

	DPOTRF			DGEQRF					
	ath64			ath64			c2d		Itan2
N	acm	atl	got	acm	atl	got	atl	got	atl
25	12	8	12	12	4	4	4	4	4
50	24	8	24	12	8	8	12	12	8
75	28	8	28	12	8	12	12	16	8
100	28	4	36	12	4	16	8	12	8
125	28	8	32	12	12	16	12	16	8
150	52	16	52	12	12	16	12	16	8
175	28	16	60	16	12	24	12	16	8
200	28	16	68	12	12	24	16	16	16
250	28	16	84	16	12	28	16	24	16
300	16	16	76	24	28	28	12	24	16
350	16	52	88	28	28	32	56	28	24
400	16	24	144	28	28	32	56	28	40
450	52	52	96	24	28	32	56	32	24
500	56	24	88	32	28	36	56	40	24
600	80	52	88	28	28	40	56	40	24
700	112	52	144	32	36	44	56	40	24
800	128	52	144	32	52	52	56	48	24
900	192	112	192	32	52	52	56	44	40
1000	128	112	192	56	52	52	56	52	40
1200	192	128	192	56	52	52	56	64	40
1400	192	160	192	56	52	52	56	72	56
1600	192	160	192	56	52	52	56	80	56
1800	192	160	192	56	52	52	56	80	56
2000	192	224	224	56	52	52	56	96	56

also DSYRK. DSYRK's performance, like DGEMM, grows strongly with problem size.

We note that the GotoBLAS `DGEQRF`'s NB on the **c2d** is still growing at our last problem size, though it is nowhere near as large as the Cholesky block. There are probably at least two factors in play here. The first is that such an excellent `DGEMM` (Goto achieves as much as 94% of theoretical peak) can hide quite a bit of poor unblocked performance before being washed out. The second reason the GotoBLAS's blocking factor is still growing is linked to the fact that the GotoBLAS block for the Level 2 cache, which we discuss further below.

Figures 2(b-d) show the % improvement from empirically tuning `NB` for a range of problem sizes for each BLAS library; each of Figures 2(b-d) shows the % improvement over the MFLOP rates provided in Figures 1 (b-d), respectively. More specifically, the X-axis of Figure 2 is computed as: $100.0 \times \left(\frac{\texttt{NB-tuned factorization performance}}{\texttt{default-blocked factorization performance}}\right)$. Since LAPACK's default `NB` is chosen to optimize mid-range problems, it is unsurprising that the general trend is that tuning `NB` helps most for small and large problems, with less benefit generally in the middle areas. We notice that the tuning is particularly helpful in the small-case problems for ACML and ATLAS, which displayed relatively poor performance in this area when using the default blocking factors. For large problems, we see that the **goto** BLAS on the Core2Duo almost always show substantial speedups. This is probably due to the fact that the GotoBLAS block for the L2 cache [18], and thus utilizing a larger block factor as N is increased allows the factorization's GEMM calls to perform nearer peak. In Figure 2 (b), we see that ATLAS on the Core2Duo shows substantial speedups across the entire range. This is mainly due to their being a mismatch between the default `NB` of 32, which is not large enough for ATLAS's `DGEMM` to reach its full potential. All in all, we observe substantial speedups for both small (as high as 30%) and large (as high as 20%) problems.

In order to demonstrate that there is nothing x86-specific in this tuning advantage, Figure 2 (a) shows the improvement from tuning LAPACK's factorizations using the ATLAS BLAS on a 900Mhz Itanium 2. We see a huge (as much as 75%!) improvement for small cases when `NB` is tuned; large problems are improved substantially (as much as 15%).

With these results, we see that the empirical tuning of LAPACK's factorization routines provides substantial speedups, and it seems this should hold for most of LAPACK's blocked routines. Therefore, we strongly believe that this is a simple, general technique that can be used to markedly increase the performance of an overwhelming majority of LAPACK routines.

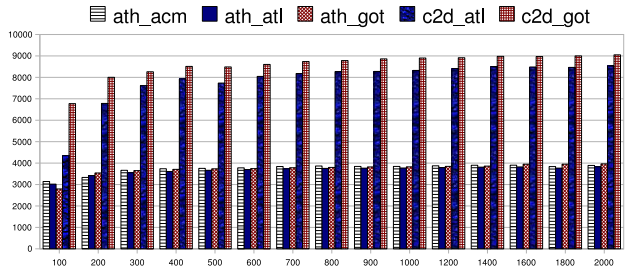
C. Tuning NB for Non-Square Problems

Many LAPACK routines operate on non-square matrices, and some operate almost exclusively on highly rectangular arrays. In particular, many service routines operate on matrices where one dimension is constrained to the blocking factor of a higher-level routine (for instance, N is always 64), but the other dimension is unconstrained (and so might be something like 1000). For this discussion, the constrained (small) matrix dimension will be referred to as the *minor dimension*, while the unconstrained dimension is the *major dimension*. Both LU (DGETRF) and QR (DGEQRF) can operate on nonsquare matrices, so we use these operations to probe nonsquare behavior.

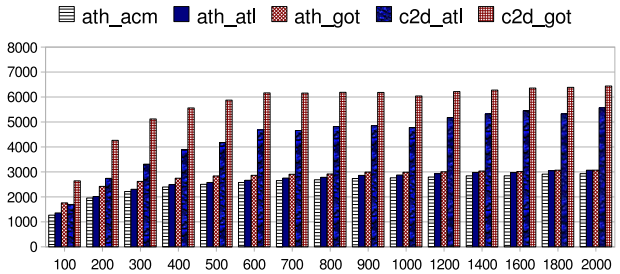
One obvious idea is to attempt to use the blocking factors discovered in the square matrix tuning for non-square. Obviously, we can't use the major dimension to select the tuned `NB`, since blockings greater than the smallest dimension of the matrix don't make sense (assuming the square blockings used in LAPACK). Therefore, our first question is: how much performance is lost if we use our square-tuned `NB`s, where our decision is based on the minor dimension? Table II provides the answer to this question.

In Table II, we have surveyed all three libraries on the **ath** architecture, with varying operations. The first column gives the architecture, library, and LAPACK operation, the second column indicates which dimension (matrix is $M \times N$) is constrained, and to what value it is constrained to. The third column indicates the blocking suggested by our square-tuned `NB`s, where `NB` choice is based on the indicated minor dimension. Then, for this minor dimension, we tune the blocking factor for a variety of major dimension sizes. For each size, we report the best blocking factor found, and in parenthesis, the percentage of its performance that the square-tuned `NB` achieves.

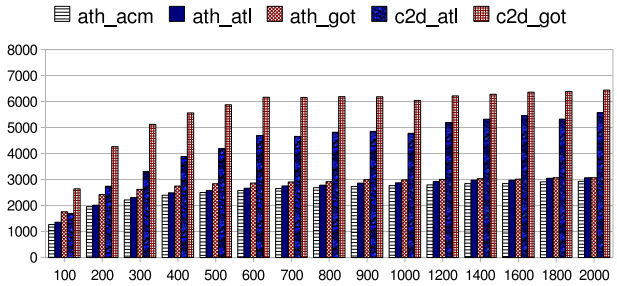
While the details change with architecture, operation, and matrix dimensions, several important trends are immediately obvious. The first is that if the operation is called on these



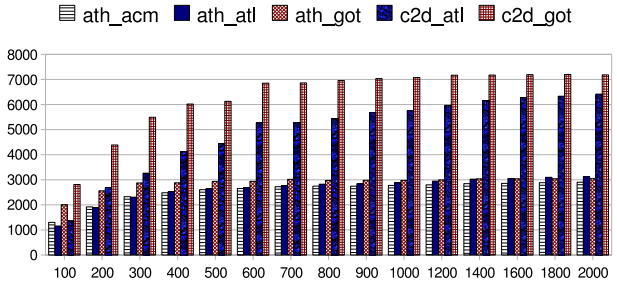
(a) DGEMM (matrix multiply)



(b) DGEQRF (QR), NB=32

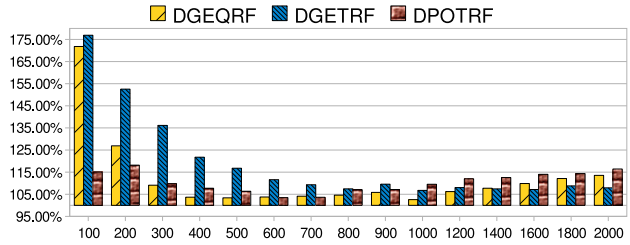


(c) DGETRF (LU), NB=64

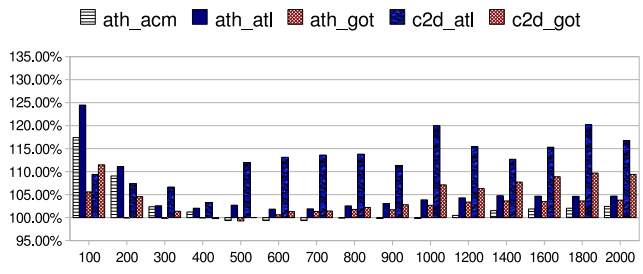


(d) DPOTRF (Cholesky), NB=64

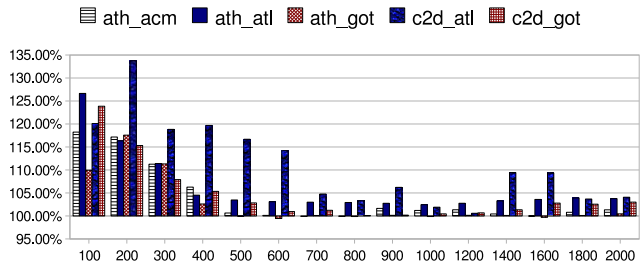
Fig. 1. Basic DGEMM and LAPACK Factorization Performance in MFLOPS for ACML, ATLAS, and GOTO on 2.2 Ghz Athlon (first three bars) and ATLAS and GOTO on 2.4Ghz Core2Duo (last two bars). Factorizations use LAPACK's default blocking.



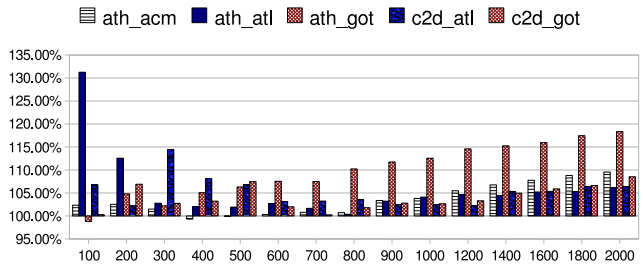
(a) Percentage LAPACK factorization improvement using ATLAS BLAS on 900Mhz Itanium 2



(b) DGEQRF (QR) on Athlon-64 and Core2Duo



(c) DGETRF (LU) on Athlon-64 and Core2Duo



(d) DPOTRF (Cholesky) on Athlon-64 and Core2Duo

Fig. 2. NB-optimized LAPACK factorization performance as a percentage of default NB performance.

TABLE II
SELECTED RESULTS FOR TUNING NB FOR NON-SQUARE PROBLEMS. FOR EACH PROBLEM SIZE, WE REPORT BOTH THE BEST NB, AND, IN PARENTHESIS, THE % OF ITS PERFORMANCE THAT WE GET USING THE SQUARE-TUNED NB WITH N = MIN(M,N).

arch/ lib/op	minor dim	square NB	Major Dimension 50	100	300	500	1000	2000	10000
ath/acml/QR	M=32	12	1(74%)	1(84%)	8(100%)	12(100%)	12(100%)	12(94%)	16(93%)
ath/acml/QR	N=32	12	1(76%)	1(90%)	8(100%)	8(97%)	8(92%)	8(92%)	8(83%)
ath/acml/QR	N=64	12	1(82%)	16(98%)	8(x4%)	8(99%)	8(97%)	8(100%)	8(88%)
ath/goto/QR	M=32	4	1(79%)	12(89%)	16(81%)	16(78%)	24(64%)	24(61%)	24(65%)
ath/goto/QR	N=32	4	1(73%)	1(78%)	1(84%)	16(85%)	8(89%)	8(92%)	8(92%)
ath/atlas/LU	N=32	8	8(100%)	8(100%)	8(100%)	12(99%)	12(97%)	12(95%)	8(100%)
ath/atlas/LU	N=64	16	8(98%)	16(100%)	12(94%)	12(92%)	12(92%)	12(94%)	12(84%)

highly nonsquare shapes, then it makes sense to tune for it explicitly, as the best rectangular NB may substantially outperform the square-tuned cases (this is particularly true for the GotoBLAS). Even for a given minor dimension, we see that the best NB often grows as the major dimension is increased; this observation alone is a strong indication that even very well-tuned square-case NBs will not provide optimal non-square results as the gap between minor and major dimensions grow. Another take-home lesson from this table is that the best tuning parameter for a column panel (eg, minor dimension is N) is not necessarily the same as that for an equal-sized row panel (minor dimension is M): row panels tend to need larger NBs, (this would provide more TLB reuse, which is not a problem for column panels).

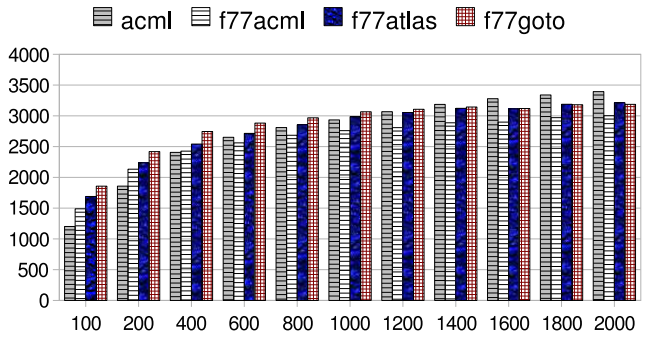
This table also points us to an obvious way to speed up LAPACK beyond merely tuning the blocking factor. Note that choosing NB=1 in these timings essentially means that the unblocked service routines (DGETF2 for DGETRF and DGEQR2 for DGEQRF), which call the Level 1 and 2 BLAS only, are used instead of the Level-3 BLAS-based blocked implementations. Blocked LAPACK routines typically call their unblocked equivalents with one dimension (usually M) set to the size of the original problem (in the first step; its size will decrease by NB during each iteration), while the second dimension is set to NB. DGETRF will factor an $M \times$NB portion of A into L and U using the unblocked routine DGETF2 in its first step. Now, if we assume NB is 32 or 64, then Table II actually shows the performance we can expect from a NB-tuned DGETRF (DGEQRF) routine when used in place of DGETF2 (DGEQR2). Anytime the best NB $\neq 1$, then it would be faster to have the blocked routine call a second blocked routine, which uses the smaller blocking factor indicated in the appropriate column of the table. Unfortunately, we don't have room to indicate how much of a performance win this would be for all table entries, but we can bring out a few highlights: For $N = 1000, N = 64$, the tuned blocked implementation is (1.67,1.34,1.76) times faster than the unblocked implementation of (**ath**/ACML/QR, **ath**/Goto/QR, **ath**/ATLAS/LU), respectively. Note that this would allow LAPACK's statically blocked routines to block the "unblocked" code, just as the recursive formulation does, which is key, in that it allows the outer implementation to use larger blocking factors, and thus achieve a higher percentage of DGEMM peak (i.e., the speedup from this technique can go beyond that of merely speeding up the "unblocked" code, since faster "unblocked" code means that we can afford a large NB, which may substantially improve DGEMM performance).

D. Tuned Reference vs. Library-Provided LAPACK

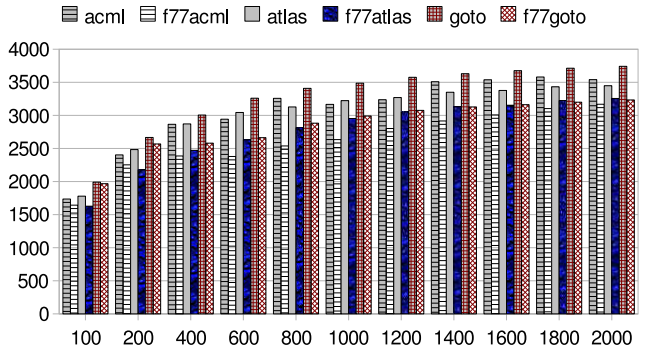
The surveyed optimized libraries (ACML/ATLAS/Goto) provide LAPACK routines in addition to the BLAS. The natural question is how well do the empirically tuned reference LAPACK routines compare with their optimized-library counterparts. The answer is surprisingly well, as we can see in Figure 3. In this figure, we compare the performance of various implementations of DGEQRF, DGETRF, and DPOTRF on the Athlon-64. The native implementations (i.e. LAPACK routines present in the optimized library) are annotated with only the library name (eg., "acml"), while the version that use the netlib FORTRAN77 reference implementations with an empirical tuning step have f77 prefixed to the library that provided the BLAS (eg., "f77acml" is the reference implementation of the LAPACK routine from netlib, with the blocking factor empirically tuned for the ACML BLAS).

Figure 3 (a) shows the performance of various implementations of DGEQRF on selected problem sizes on the Athlon-64 (the number of problem sizes shown has been reduced as an aid to legibility). As mentioned, neither GotoBLAS nor ATLAS provides a native QR, so for these libraries we show only the performance of the tuned netlib LAPACK. ACML provides a native QR, and so we see its performance as the first bar for each problem size. For small problems, it is the worst performing QR, with the best performance coming from the empirically tuned LAPACK using the GotoBLAS. For large problems, ACML's native implementation provides the best performance.

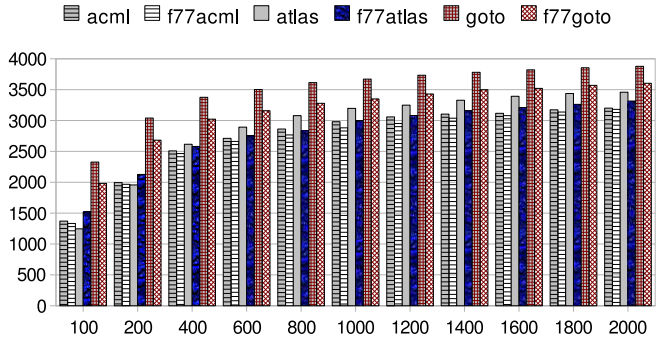
For LU factorization (Figure 3 (b)), all libraries provide a native routine, and the native routines beat their tuned netlib versions across the entire surveyed range. For small problems, there is little gap between the f77 and native implementations, but this gap gets noticeably larger for large problems, particularly for the GotoBLAS library. The same basic trend is seen for Cholesky (Figure 3 (c)), with the marked exception of the GotoBLAS-optimized DPOTRF. The GotoBLAS provides a noticeably superior Cholesky regardless of LAPACK used,



(a) DGEQRF



(b) DGETRF



(c) DPOTRF

Fig. 3. Tuned reference vs. library-provided (native) LAPACK on the Athlon-64. Native routines have same pattern as reference, but the background color is light gray rather than white (or blue/dark gray for ATLAS). From left to right: ACML-LA+ACML-BLAS, NETLIB-LA+ACML-BLAS, ATLAS-LA+ATLAS-BLAS, NETLIB-LA+ATLAS-BLAS, GOTO-LA+GOTO-BLAS, NETLIB-LA+GOTO-BLAS (there is no ATLAS- or GOTO-provided DGEQRF, and so these bars do not show up in figure (a)).

but its native implementation is commandingly better for small problems.

These results are fairly easy to understand for ATLAS: ATLAS uses recursive implementations (based on the work of [19], [20], [21]) of LU and Cholesky, and for large problems it is expected that the much larger DGEMM calls seen at the top of the recursion will naturally lead to the performance advantage we see for large problems. Recursion is essentially the only advantage ATLAS has over netlib LAPACK: we don't currently do any system tuning at this level. For Cholesky, the ATLAS `DPOTRF` actually loses to the empirically-tuned netlib LAPACK. This is probably due to recursive overhead: remember that in LU and QR, the recursion cuts only the N dimension, while in Cholesky it cuts both dimensions, which leaves very few operations over which to amortize the cost of the recursive calls. It seems likely that if the unblocked code in LAPACK was replaced with blocked code, as discussed in Section III-C, that the netlib LAPACK could win for small problems on LU as well, again mostly due to overhead savings. It seems unlikely that even this improved version could compete for very large problems: the recursive advantage should grow with problem size, so eventually recursion will begin to win for large enough matrices. However, the gap is not too large for the problem sizes surveyed here, so with better unblocked code, it is possible the netlib LAPACK could be made to run competitively across this entire range.

It is difficult to know what the other libraries are doing to produce these performance curves. GotoBLAS does get better `DSYRK` (particularly small-case) performance than ATLAS, which in turn gets much better `DSYRK` performance than ACML, which probably helps explain `DPOTRF` results. GotoBLAS's small-case performance advantage is large enough that there are probably additional factors needed to explain this gap, however.

IV. Summary and Future work

We have presented a survey of factorization performance on two important architecture families (with a few results from a third). Using our empirical tuning system, we have demonstrated that tuning `NB` can provide substantial performance improvements regardless of BLAS used or the architecture it runs on. This simple tuning can be extended to speed up an overwhelming majority of LAPACK routines, in addition to the factorizations discussed here. We note that many routines may be further improved by rewriting them as recursive algorithms, but this is not currently possible for several important LAPACK routines. Therefore, empirically-tuned `NB` LAPACK implementations of such routines would actually represent the best available implementations today. Even in the case where recursive formulations are possible, we have shown that statically-blocked LAPACK performance can be made competitive with these techniques. We have also investigated tuning for nonsquare matrices, which could improve the performance of many of LAPACK's service routines, and this investigation has pointed out a straightforward adaptation of LAPACK's present strategy that can yield further performance

improvements without requiring LAPACK to be rewritten in languages supporting recursion.

A. Future work

In this paper, we concentrated on the factorizations, since they are probably the most widely used routines in the library (they are called by many higher-level routines, in addition to being called directly), in addition to being well understood routines which possess optimized implementations we can compare against. The tuning framework can be easily extended to other operations, and we plan to do so (we are particularly interested in tuning additional routines used in finding eigenvalue/vectors). The framework already allows other data types/precisions to be tuned, and probing the effect of varying type and precision on optimal blocking is an experiment worth doing. This investigation concentrated on serial performance: an obvious extension of great interest would be to see how results vary when the same libraries are used, but threading is enabled. Finally, our investigations pointed to significant speedups if we block the present unblocked codes with the small blocking factors discovered in our rectangular tuning. This should be investigated for the factorizations, and see how the resulting performance stacks up against state-of-the-art recursive formulations. If successful here, we should see if this technique can be used on some of the LAPACK routines that do not currently possess recursive formulations.

REFERENCES

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

[2] J. Dongarra, J. Bunch, C. Molar, and G. Stewart, *LINPACK Users' Guide*. Philadelphia, PA: SIAM, 1979.

[3] B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ikebe, V. Klema, and C. Molar, "Matrix Eigensystem Routines - Eispack Guide, Second Edition," in *Matrix Eigensystem Routines - Eispack Guide*, ser. Lecture Notes in Computer Science, no. 6. Springer-Verlag, 1976.

[4] J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–17, 1990.

[5] J. Dongarra, J. D. 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*, vol. 14, no. 1, pp. 18–32, 1988.

[6] ——, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–17, 1988.

[7] R. Hanson, F. Krogh, and C. Lawson, "A Proposal for Standard Linear Algebra Subprograms," *ACM SIGNUM Newsl.*, vol. 8, no. 16, 1973.

[8] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, 1979.

[9] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.

[10] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, February 2005, `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[11] ——, "Atlas homepage," http://math-atlas.sourceforge.net/.

[12] R. C. Whaley and A. M. Castaldo, "Achieving accurate and context-sensitive timing for code optimization," Accepted for publication in *Software: Practice and Experience*, 2008.

[13] R. Walpole, R. Myers, S. Myers, and K. Ye, *Probability & Statistics for Engineers & Scientists*, 7th ed. Upper Saddle River, NJ: Prentice-Hall, 2002.

[14] LAPACK, "`LAPACK/TIMING/LIN/dopla.f`," http://ib.cnea.gov.ar/~fiscom/Libreria/lapack_blas/source/TIMING/LIN/dopla.f.

[15] K. Goto, "Gotoblas homepage," http://www.tacc.utexas.edu/resources/software/.

[16] Intel, "Mkl homepage," http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.

[17] AMD, "Acml homepage," http://www.amd.com/acml.

[18] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," Accepted for publication in *Transactions on Mathematical Software*, 2008.

[19] S. Toledo, "Locality of reference in lu decomposition with partial pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 18, no. 4, 1997.

[20] F. Gustavson, "Recursion leads to automatic variable blocking for dense linear-algebra algorithms," *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 737–755, 1997.

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