

# ATLAS Version 3.8 : Overview and Status \*

R. Clint Whaley<sup>†</sup>

November 5, 2007

## Abstract

This paper describes the widely-used ATLAS (Automatically Tuned Linear Algebra Software) project as it stands today. ATLAS is an instantiation of a paradigm in high performance library production and maintenance, which we term AEOS (Automated Empirical Optimization of Software); this style of library management has been created in order to allow software to keep pace with the incredible rate of hardware advancement inherent in Moore's Law. ATLAS is the application of this AEOS paradigm to linear algebra software. ATLAS produces a full BLAS (Basic Linear Algebra Subprograms) library as well as providing some optimized routines for LAPACK (Linear Algebra PACKage). This paper overviews the basics of what ATLAS is and how it works, highlights some of the recent improvements available in version 3.8.0 (the newest stable release of ATLAS, scheduled for release in summer 2007), as well as discussing some of the current challenges and future work.

## 1 Introduction

High performance computing is differentiated from general computing by its voracious appetite for computing resources. Despite hardware performance that has been steadily improving according to Moore's Law, this is as true today as it was a decade ago. Scientific modeling provides an illustration of this phenomenon. In many of these applications, computational power is the main constraint preventing the scientist from modeling more complex problems, which would then more closely match reality. As more computational power becomes available, the scientist typically increases the complexity/accuracy of the model until the limits of the computational power are reached. Therefore, since many applications have no practical limit of "enough" accuracy, it is important that each generation of increasingly powerful computers have well optimized computational kernels, which in turn allow for efficient execution of the higher-level applications that use them.

The traditional path to achieving high performance in HPC involves compilation research combined with library production. General purpose compilers do not, in practice, achieve the very high percentages of peak on the complex kernels demanded by HPC applications. Therefore, since a user cannot write an arbitrary code and expect it to run at the extreme efficiencies demanded by HPC applications, the community has responded by emphasizing library production. In particular, APIs for reusable performance kernels are standardized, allowing these kernels to be hand-tuned by teams of experts for a given platform. Once these standard kernels are available for the platform of interest, higher-level applications that leverage them can run at high efficiencies without extensive additional tuning.

Hand-tuning performance-critical kernels for each architecture of interest suffers from two main drawbacks: First, creating software that realizes near peak rates of execution requires detailed knowledge of a complex set of interrelated factors, including the operation being optimized, the target architecture(s), and all the intervening software layers. Even when the implementer possesses

---

\*This work was supported in part by the National Science Foundation, grant # NSF CRI CNS-0551504

<sup>†</sup>Dept. of Computer Science, Univ of TX, San Antonio, TX 78249, whaley@cs.utsa.edu

such broad understanding, the interactions between various hardware/software layers guarantee that significant empirical tuning of the initial kernel will be required. Therefore, optimizing even the simplest of real-world operations for high performance usually requires a sustained effort from the most technically advanced programmers, which are in critically short supply. Second, even when the requisite programming talent is available, hand-tuning such codes is a time consuming task, so that far too often, when the optimized libraries are finally ready to come on line, the generation of hardware for which they are optimized is well on its way towards obsolescence. This difficulty of keeping software highly optimized in the face of hardware change is a persistent problem for both hand-tuning and compilers.

These problems, taken together, led to the implementation of empirically tuned library generators such as PHiPAC [3], FFTW [22, 13, 12] as well as the topic of this paper, ATLAS [28, 25, 26, 27, 30, 29]. The central idea behind these packages is that since it is difficult to predict a priori whether or by how much a given technique will improve performance, one should try a battery of known techniques on each performance-critical kernel, obtain accurate timings to assess the effect of each transformation of interest, and retain only those that result in measurable improvements for this *exact* system and kernel. Thus, the need to understand the architecture in detail is removed: we are probing the system as it stands, just as the empirical technique of the scientific method probes the natural world, and just as the scientific method discards disprovable theories, we do not retain transformations that do not result in sufficient speedup.

Many groups have begun to utilize automated and empirical approaches to optimization, resulting in a plethora of differing terminologies, including “self-tuning libraries”, “adaptive software”, “empirical compilation”, “iterative compilation”, etc. While these approaches differ strongly in details, in order to fall into the classification related to our research they must have some commonalities:

1. The search must be *automated* in some way, so that an expert hand-tuner is not required.
2. The decision of whether a transformation is useful or not must be *empirical*, in that an actual timing measurement on the specific architecture in question is performed, as opposed to the traditional application of transformations using static heuristics or profile counts.
3. These methods must have some way to vary/adapt the software being tuned.

With these broad outlines in mind, we lump all such empirical tunings under the acronym AEOS, or Automated Empirical Optimization of Software, and Section 1.1 outlines the requirements of such systems, while Section 1.2 discusses the studied methods of software adaptation.

## 1.1 Basic AEOS Requirements

The basic requirements for supporting high performance kernel optimization using AEOS methodologies are:

- *Isolation of performance-critical routines*: Just as with traditional libraries, the performance-critical sections of code must be isolated (usually into subroutines, which dictates the need for a standardized API).
- *A method of adapting software to differing environments*: Since AEOS depends on iteratively trying differing ways of performing the performance-critical operation, the author must be able to provide implementations that instantiate a wide range of optimizations. This may be done very simply, for instance by having parameters in a fixed code which, when varied, correspond to differing cache sizes, etc., or it may be done much more generally, for instance by supplying a highly parameterized source generator which can produce an almost infinite number of implementations. No matter how general the adaptation strategy, there will be limitations or built-in assumptions about the required architecture which should be identified in order to estimate the probable boundaries on the code’s flexibility. Section 1.2 discusses software adaptation methods in further detail.

- *Robust, context-sensitive timers:* Since timings are used to select the best code, it becomes very important that these timings be accurate. Since few users can guarantee single-user access, the timers must be robust enough to produce reliable timings even on heavily loaded machines. Furthermore, the timers need to replicate as closely as possible the way in which the given operation will be used. For instance, if the routine will normally be called with cold caches, cache flushing will be required. If the routine will typically be called with a given level of cache preloaded, while others are not, that too should be taken into account. If there is no known machine state, timers allowing for many different states, which the user can vary, should be created.
- *Appropriate search heuristic:* The final requirement is a search heuristic which automates the search for the most optimal available implementation. For a simple method of code adaptation, such as supplying a fixed number of hand-tuned implementations, a simple linear search will suffice. However, when using sophisticated source generators with literally hundreds of thousands of ways of doing an operation, a similarly sophisticated search heuristic must be employed in order to prune the search tree as rapidly as possible, so that the optimal cases are both found and found quickly (obviously, few users will tolerate heavily parameterized search times with exponential growth). If the search takes longer than a handful of minutes, it needs to be robust enough to not require a complete restart if hardware or software failure interrupts the original search.

## 1.2 Methods of Software Adaptation

We employ three different methods of software adaptation. The first is widely used in programming in general, and it involves parameterizing characteristics which vary from machine to machine. In linear algebra, the most important of such parameters is probably the blocking factor used in blocked algorithms, which, when varied, varies the data cache utilization. In general, parameterizing as many levels of data cache as the algorithm can support can provide remarkable speedups. With an AEOS approach, such parameters can be compile-time variables, and thus not cause a runtime slowdown. We call this method *parameterized adaptation*.

Not all important architectural variables can be handled by parameterized adaptation (simple examples include instruction cache size, choice of combined or separate multiply and add instructions, length of floating point and fetch pipelines, etc), since varying them actually requires changing the underlying source code. This then brings in the need for the second method of software adaptation, *source code adaptation*, which involves actually generating differing implementations of the same operation.

There are at least two different ways to do source code adaptation. Perhaps the simplest approach is for the designer to supply various hand-tuned implementations, and then the search heuristic may be as simple as trying each implementation in turn until the best is found. At first glance, one might suspect that supplying these multiple implementations would make even this approach to source code adaptation much more difficult than the traditional hand-tuning of libraries. However, traditional hand-tuning is not the mere application of known techniques it may appear when examined casually. Knowing the size and properties of your level 1 cache is not sufficient to choose the best blocking factor, for instance, as this depends on a host of interlocking factors which usually defy a priori understanding in the real world. Therefore, it is common in hand-tuned optimizations to utilize the known characteristics of the machine to narrow the search, but then the programmer writes various implementations and chooses the best.

For the simplest AEOS implementation, this process remains the same, but the programmer adds a search and timing layer to accomplish what would otherwise be done by hand. In the simplest cases, the time to write this layer may not be much if any more than the time the implementer would have spent doing the same process in a less formal way by hand, while at the same time capturing at least some of the flexibility inherent in AEOS-centric design (eg., a kernel written for the PIII may turn out to also be efficient on an Opteron, or a generic implementation may yield good performance

for some simple kernels on a system with an excellent compiler, etc). We will refer to this source code adaptation technique as *multiple implementation*. Due to its obvious simplicity, this method is highly parallelizable, in the sense that multiple authors can meaningfully contribute without having to understand the entire package. In particular, various specialists on given architectures can provide hand-tuned routines without needing to understand other architectures, the higher level codes (e.g. timers, search heuristics, higher-level routine which utilize these basic kernels, etc). This makes multiple implementation a very good approach if the user base is large and skilled enough to support an open source initiative along the lines of, for example, Linux.

The second method of source code adaptation is *source generation*. In source generation, a source generator (i.e., a program that writes other programs) is produced. This source generator takes as parameters the various source code adaptations to be made. As before, simple examples include instruction cache size, choice of combined or separate multiply and add instructions, length of floating point and fetch pipelines, and so on. Depending on the parameters, the source generator produces a routine with the requisite characteristics. The great strength of source generators is their ultimate flexibility, which can allow for far greater tunings than could be produced by all but the best hand-coders. However, generator complexity tends to go up along with flexibility, so that these programs rapidly become almost insurmountable barriers to outside contribution.

ATLAS therefore combines these two methods of source adaptation, where a kernel generator emits transformed ANSI C code for maximal architectural portability, and multiple implementation is utilized to encourage outside contribution and to allow for extreme architectural specialization via hand-tuned (often in assembly) implementations.

## 2 ATLAS Overview

ATLAS presently provides a complete BLAS implementation, and a handful of important routines from LAPACK. ATLAS currently does not tune the banded and packed BLAS: packed and banded are unoptimized reference implementations only (note that ATLAS does have some highly efficient prototype packed Level 3 routines, as discussed in [29], but since the BLAS standard does not provide Level 3 packed BLAS, we ignore them here). In the rest of this section, we briefly describe the nature of ATLAS's present support for each level of routine, starting from the highest to the lowest. Therefore, Section 2.1 describes ATLAS's LAPACK support and Sections 2.2, 2.3, 2.4 describe ATLAS's dense Level 3, 2 and 1 BLAS support, respectively.

### 2.1 Explicit LAPACK Support In ATLAS

LAPACK [2] (Linear Algebra PACKage) is an extremely comprehensive Fortran 77 package for solving the most commonly occurring problems in numerical linear algebra. The size and complexity of LAPACK make it highly unlikely that ATLAS will ever provide a complete implementation. Therefore, ATLAS natively provides only a relative handful of LAPACK's full suite of routines. However, ATLAS is constructed so that it can automatically add its optimized routines to the standard LAPACK library from netlib, so that a complete library is easily achieved. ATLAS presently provides 10 basic routines from LAPACK, each of which is available in all four data types (double and single precision real and complex), for a total of 40 routines. These are all routines using or providing for the LU or Cholesky factorizations, including matrix inversion using these methods. The routines provided are: GESV, GETRF, GETRS, GETRI, TRTRI, POSV, POTRF, POTRS, POTRI, and LAUUM. Standard LAPACK defines only a Fortran interface, but for these routines, ATLAS also provides its own C interface, modeled after the official C interface to the BLAS [5, 4], which includes support for row-major storage in addition to the standard column-major implementations.

Because LAPACK is so large, we add support for a routine only when we believe we can provide substantially better performance than the LAPACK implementation, and that the routine is widely used enough to justify the substantial development and maintenance cost of inclusion in ATLAS.

All the above routines center around the LU and Cholesky factorizations, where ATLAS’s implementations have the key algorithmic advantage of using recursion, rather than statically blocking as LAPACK does. Prior work [23, 14, 15, 1, 11] had shown the considerable advantage recursion provides for these routines, and so we added their support. Essentially, recursion provides two key benefits: (1) L3BLAS calls are substituted for L2BLAS calls, resulting in performance improvements for most cases where  $N > 4$ , and (2) Due to the dynamic blocking applied by recursion, the L3BLAS are called with much larger dimensions than when statically blocked, allowing asymptotic performance to be reached for large problems.

There are three basic factorizations in LAPACK (sometimes called the ‘three amigos’ due to their ubiquity), and they are some of the most heavily used routines in dense linear algebra, so it obviously makes sense to provide optimized versions. ATLAS has not yet provided a recursive QR factorization, because unlike for Cholesky and LU, the recursion is not free, and is more numerically complicated. Therefore, since this routine requires higher development and maintenance costs, and does not provide quite as much performance improvement (essentially, QR benefits mainly from substituting the L3 BLAS for L2, since extra costs prevent recursion from achieving asymptotic performance), ATLAS has not yet provided an optimized QR, though we plan to do so eventually.

LAPACK has a routine called `ILAENV` which is called by most routines in order to tune LAPACK’s static blocking parameters on an individual basis. ATLAS provides this routine, with values that have been somewhat adapted for ATLAS usage, but only in a static and crude way. Ultimately, this routine should be optimized empirically at install time, and this is definitely in our future plans.

## 2.2 Dense Level 3 BLAS Support in ATLAS

The Level 3 BLAS [8] perform matrix matrix operations, and consist of six routines for each real precision, and nine routines for each complex precision, for a total of 30 Level 3 BLAS. The L3BLAS have  $O(N^3)$  operations, but need only  $O(N^2)$  data. Because these routines can be easily reordered and blocked for cache reuse, highly tuned L3BLAS can be made to run fairly close to theoretical FPU peak on most architectures, and thus the Level 3 are the most optimizable of the BLAS routines. Therefore, LAPACK is designed so that the execution time of most routines is dominated by the L3BLAS, and so ATLAS concentrates most of its empirical tuning for this BLAS level.

It has long been known that the entire L3BLAS can be efficiently supported using only a very efficient matrix multiply (GEMM: General Matrix Multiply) routine. Such BLAS are known as GEMM-based BLAS [18, 19, 20, 7, 15, 16] and ATLAS uses a recursive formulation of the GEMM-based developed by Antoine Petitet [29]. In fact, ATLAS speeds up the entire L3BLAS (including packed) using only a single simplified gemm kernel, which we call *gemmK*. This simplified kernel is blocked to constant dimensions (usually for the Level 1 Cache), and then heavily optimized for both the FPU and memory hierarchy using parameterization combined with both multiple implementation and code generation. ATLAS is heavily polyalgorithmic, choosing the approach based on both matrix shape and empirically discovered architectural features. More details about some of these choices are provided in [30], while [29] discusses some of the issues in leveraging *gemmK* for supporting the entire L3BLAS.

For simplicity, we will discuss only the most common case here, where *gemmK* is used to build a full real GEMM for square matrices of non-trivial size. In this case, ATLAS’s *gemmK* kernel is a simplified matmul where it is known that A is in transpose format, B is in no-transpose format, and the matrix dimensions are all fixed to an empirically determined cache blocking factor,  $N_b$  (ATLAS uses a specialized data copy in order to leverage this kernel for all the L3BLAS operations). In this case, ATLAS first empirically searches the optimization space supported by the *gemmK* code generator in order to find the best generated kernel possible. The code generator takes a host of tuning parameters, controlling various factors such as  $N_b$ , type of FPU instruction (FMAC or separate multiply and add), FPU pipeline depth, loop unrolling on all three loops (the outer loops are unrolled and jammed into the innermost loop), register blocking, etc. The code generator is written in, and generates, strict ANSI-C for maximal portability. This search should yield reasonable results on any

cache-based architecture with a decent C compiler. Originally, ATLAS achieved results as good as, and often better than, the vendor BLAS using this search alone. However, architectures have become more and more complex, and compilers (even assisted by ATLAS’s empirical search) have not been able to keep pace. In particular, compilers have historically done a poor job of autovectorizing complex kernels for SIMD vector instructions such as SSE. Since Intel, in particular, has almost abandoned scalar FPU performance in pursuit of vector performance, the code generator+compiler combination now often lags considerably the performance that can be obtained via careful hand-tuning.

Therefore, ATLAS’s empirical tuning for *gemmK* has a second step, where we optimize the same routine using multiple implementation. The *gemmK* that is used by ATLAS will actually be the best performing kernel found by these two searches. In the multiple implementation search, a set of index files describe many different *gemmK* implementations (usually hand-tuned for a prior architecture of some sort) which could possibly be used for the architecture being tuned for. These kernels are fixed implementations, but can take some parameterization tuning, which is controlled primarily by the index file settings (see [24] for further details). These routines can be written in almost any language (some languages depend on support libraries not available to a C-compiled library, and thus are unsuitable), including assembly. Many of the most efficient routines are written in highly-tuned assembly (Section 2.2.1 highlights why assembly use is particularly useful in ATLAS).

### 2.2.1 Using assembly in ATLAS

It is common knowledge that assembly language is useful in order to achieve high performance, but another reason ATLAS uses it is to achieve persistent performance in the face of compiler change. Once ATLAS has tuned itself to a given architecture, we save the results of good searches, so that the install is not so onerous. However, this is prone to problems, since even trivial changes in a compiler’s optimization phases may cause a formerly efficient routine to experience a catastrophic performance loss. Even worse, our experience has been that compilers tend not to stress real-world floating point performance, and so system-wide performance regressions are common for our types of kernels (i.e. usually *gemmK* just needs to be re-tuned for a new compiler version, but even this is not a solution when the compiler as a whole gets slower for a particular machine, which happens fairly regularly). Therefore, even when the code generator achieves near-peak performance, we will often use the generated routine as a model in order to write an assembly routine, which we know will maintain its performance in the face of compiler change.

The other reasons to use assembly all relate to achieving higher performance. After writing ATLAS, we were surprised to find that on almost every platform where ATLAS failed to achieve adequate efficiency, it was not a weakness in our algorithm, or selection of blocking sizes, etc., but due to problems with the compiler not being able to generate efficient backend code. In particular, SIMD vectorization is the Achilles’ heal of most compilation frameworks when used on real numerical kernels (as opposed to static benchmark codes). Less obviously, there are a host of instruction-related optimizations that are increasingly critical on modern machines. Simple ones include CISC optimizations such as code alignment, code compaction, etc. As modern machines take on complex frontends (eg. x86 frontends take in CISC instructions and translate them to RISC-like macro-ops, and some PowerPC frontends take in RISC instructions and translate them to VLIW-like groups, etc), we have found it to be increasingly important to tune the exact instruction groupings and size to the architecture’s frontend. For instance, on the original Athlon, the frontend needed to be fed precisely sized (using nops) bundles of instructions in order to drive the backend at it’s maximal rate. The best performance for any C-compiled frontend was something like 70% of peak, but when written in assembly with these frontend optimizations, over 92% of theoretical peak could be achieved.

Similarly, our original Core2Duo kernel (written in assembly) only achieved 71% of peak, but after CISC compaction and code alignment, we could boost this to 78%. On the PowerPC970FX, the best kernel (C or assembly) peaked around 69%, until we discovered that the frontend worked best

when like instructions were issued in groups of four (eg. 4 loads, followed by 4 FMACS, etc), which boosted performance to 86% of peak. Finally, even on the Opteron, where the frontend does not seem to be a bottleneck, CISC code compaction gave us a roughly 5% speedup in LU performance (even though GEMM ran at the same speed) due to decreased instruction cache thrashing. None of these optimizations are done effectively by compilers (though most compilers have some crude code alignment optimizations, they have not yet proven widely effective), and so we see that with modern machines assembly is often required to get adequate performance.

### 2.3 Dense Level 2 BLAS Support in ATLAS

The Level 2 BLAS [9, 10] perform matrix-vector operations, such as matrix-vector multiply, rank-1 update and triangular forward/backward solve. Most of these matrix-vector operations can be performed on general rectangular matrices, symmetric matrices, or triangular matrices. In all, there are 16 routines for each real precision, and 17 for each complex precision, for a total of 66 L2BLAS routines. We recall that ATLAS required only one kernel to support all 30 L3BLAS, but this is not true of the L2BLAS. The L2BLAS have  $O(N^2)$  operations, and  $O(N^2)$  data, which means we cannot compress multiple cases into one via a data copy of the matrix (since copying the matrix would be roughly as expensive as doing the operation itself), as we do in the L3BLAS. Therefore, ATLAS must tune three (five) kernels for each real (complex) precision. Essentially, we need only two classes of kernels, a tuned general matrix-vector multiply (GEMV), and a tuned rank-1 update (GER) to support a GEMV- and GER-based L2BLAS. However, since we can't afford to copy the matrix, we must support a different GEMV kernel for each transpose setting (yielding two GEMV kernels for real precision, and four for complex). More details can be found in [30].

ATLAS presently tunes these kernels using only parameterization (for cache blocking) and multiple implementation. ATLAS's factorizations actually don't call the L2BLAS, and so this level probably receives the least attention in the provision of hand-tuned routines. However, they remain important for the performance of many LAPACK routines, including finding eigenvalues (which are widely used in a variety of fields). Therefore, this is an area where ATLAS needs to improve. We have so far not found the time to do so, mainly because the possible win is so much lower. Essentially, because of their massive data reuse, the peak for the L3BLAS performance is determined by the floating point computational peak of the machine. The L2 and L1BLAS, however, have the same order data and operations, which means that their theoretical peak is usually set by the speed of the data bus, which is typically orders of magnitude slower than the computational peak. Therefore, the amount gained by optimization is necessarily less, since even moderately optimized routines can get a reasonable percentage of the achievable bus peak.

### 2.4 Dense Level 1 BLAS Support in ATLAS

The Level 1 BLAS [17, 21] do vector-vector operations such as dot product ( $dot \leftarrow x^T y$ ) or axpy ( $y \leftarrow \alpha x + y$ ). These routines perform  $O(N)$  computation on  $O(N)$  data, and therefore there is little room performance-wise for doing optimizations such as data copy. ATLAS tunes the L1BLAS only by multiple implementation (along with some simple parameterization, which is occasionally used to tune things like prefetch distance), and each routine must essentially be tuned independently. The only kernel reuse possible is that for some of the routines, the complex data type can call the underlying real kernel of the same name when the vector stride is 1. In most dense linear algebra routines, the L1BLAS contribute only marginally to the total performance, and therefore ATLAS does not tune them as well as it could, even using only multiple implementation. For instance, we provide assembly routines only for a few of the more important routines for the more common platforms. Compilers can typically do a decent job of optimization for these simple one-loop kernels. Therefore, in addition to the usual optimized routines (which perform unrolling, pipelining, prefetch, etc), there is at least one simple reference implementation designed to allow the compiler to do as much as possible. Because the L1BLAS are not nearly as highly tunable as the rest of the BLAS,

ATLAS is typically nonetheless fairly competitive with the vendor BLAS on most platforms (see [31] for some L1BLAS timing information).

### 3 The new ATLAS stable release

From January 2002 through 2007, ATLAS was mostly being maintained only in the free time of the author, and very little development or research could be accomplished. The last stable release of ATLAS (3.6.0) came in December 2003, when some short-term funding from AMD allowed for some active work aimed at exploiting AMD's new Opteron line of machines. New funding was obtained in 2006, and ATLAS has been under intensive development for roughly the last year, which should culminate in the release of the new stable release (ATLAS 3.8.0) in September of 2007. Because ATLAS development has not been active for quite some time, there were a host of infrastructure and install framework issues that needed to be addressed before real research could begin. The biggest of these has been a complete rewrite of the ATLAS configure and install mechanism, to make it work more like the standard `gnu configure/build/install` process. We have also had to revamp the ATLAS architectural support mechanisms, and get gcc performance regressions fixed so that newer compilers could build a high performance ATLAS library across all machines.

This infrastructure work dominated the recent ATLAS development efforts, but some performance and research related work was done. Some highlights include:

- *gemmK* code generator improvements, including ability to perform prefetch on most architectures, choice of loading *C* at top or bottom of K-loop (for error amelioration), and ability to peel the first iterations of the K-loop to lower the cost of K-loop startup,
- Expanded support to allow easier use of a wider range of assembly dialects (ATLAS now provides support for 64 and 32 bit assembly dialects for x86, x86-64, PowerPC, PA-RISC, and MIPS assembly),
- Extensive `gemmK` assembly multiple implementation tuning for newer architectures, including P4, P4E, Efficion, Core2Duo, Opteron, statically scheduled MIPS architectures, and PowerPC,
- An improved `ILAENV` for better non-ATLAS LAPACK performance,
- Improved performance for some non-square matrix shapes, including faster handling of GEMM's very low-rank-K update and long-K, small M and N shapes, both of which are seen in some types of recursive applications,
- Improved performance for both complex precisions on some architectures,
- Initial work that lowers floating point error on some architectures. These improvements are based on research reported in [6]; a more formal paper has been submitted in SISC for review.

### 4 Status and Future work

Now that the ATLAS framework has been modernized, there are several areas that need to be investigated in order to improve performance. With the latest release of MKL (9.1), Intel's BLAS have improved so that ATLAS runs almost 20% slower than MKL on the Core2Duo. AMD's ACML is also slightly faster than ATLAS on their present flagship product (Athlon-64/Opteron), though the advantage is quite a bit less there (roughly 5%). In the Intel case, the problem is clearly that ATLAS's *gemmK* is inferior, and our experiments make it seem probable that there are some architectural frontend optimizations that we must still discover in order to improve our present assembly kernel. More important than these architecture-specific problems, however, is finding ways to speed up the framework across all the architectures, and there are several areas we are presently planning to investigate to help with this.

We have just started to examine cache blocking from a theoretical point of view, so that we can find out if ATLAS's cache blocking strategies are deficient. ATLAS has slightly different strategies

depending on matrix shape and size, and we believe this needs to be extended, with perhaps some wholly new algorithms added. This is foundational work that we hope will guide the two other areas of proposed work which we discuss next, namely multiple and non-square blocking factors.

Another area that we plan to explore is in allowing for multiple blocking factors. ATLAS already has two possibly different blocking factors (one is used when the matrices are copied, and a possibly smaller one is used when they are not), but we need a much better infrastructure to get good performance across a range of problem sizes. In particular, our initial investigations have shown that for small-to-moderate size problems, a smaller blocking factor can yield considerably better application performance than an  $N_b$  which is selected for asymptotic GEMM performance. However, these extra kernels must be added in a way that doesn't cause uncontrolled library expansion or instruction cache thrashing, and some investigation will be required in order to both determine the best range of blocking factors required to support the full array of problem sizes and also how to best select among the candidate  $N_b$ 's. These varied  $N_b$ 's should add greatly to our ability to vary our cache blocking strategies, allowing us to block kernels for both the L1 and L2 caches, depending on both architecture and problem size.

We also need to investigate how much win there is in using non-square blocking parameters. In particular, with vectorization, there is a computational advantage to using a longer  $K$  dimension, due to the costs of vector startup and shutdown. This problem may be even more acute when small blocking factors are used.

We must also improve and extend ATLAS's threading support. Now that OpenMP is supported by gcc, we will investigate using it in addition to our present approach, which uses pthreads. ATLAS's threading support at present is rather simplified, with little to no install-time tuning. We need to write probes that empirically discover the crossover points at which additional processors become useful for a particular class of operations. As architectural trends continue to drive up the number of processors available on typical workstations, this will become increasingly important.

Similarly, our present ILAENV support for tuning LAPACK routines has not yet been automated. We plan on providing a series of timers that can be used to empirically find good parameters for ILAENV for the more important LAPACK routines across a range of problem sizes.

These and host of other improvements will drive further improvements in the search and install framework, all of which will be undertaken in the next developer series which will be started after the 3.8 stable is released.

## References

- [1] Bjarne S. Andersen, Fred G. Gustavson, and Jerzy Wasniewski. A recursive formulation of cholesky factorization of a matrix in packed storage. Technical Report UT CS-00-448, LAPACK Working Note No.146, University of Tennessee, 2000.
- [2] 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.
- [3] 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.
- [4] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.
- [5] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Pe-

- titet, R. Pozo, K. Remington, W. Walster, C. Whaley, J. Wolff, and V. Gudenberg. Document for the Basic Linear Algebra Subprograms (BLAS) Standard: BLAS Technical Forum. <http://www.netlib.org/cgi-bin/checkout/blast/blast.pl>, 1999.
- [6] Anthony M. Castaldo, R. Clint Whaley, and Anthony T. Chronopoulos. Error Analysis of Various Forms of Floating Point Dot Products. Technical report, University of Texas at San Antonio, May 2007. <http://www.cs.utsa.edu/research/tr/2007/CS-TR-2007-002.pdf>.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.
- [13] M. Frigo and S. G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, 1997.
- [14] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.
- [15] 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.
- [16] 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.
- [17] R. Hanson, F. Krogh, and C. Lawson. A Proposal for Standard Linear Algebra Subprograms. *ACM SIGNUM Newsl.*, 8(16), 1973.
- [18] 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.
- [19] 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.

- [20] 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.
- [21] 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.
- [22] See page for details. FFTW homepage. <http://www.fftw.org/>.
- [23] S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4), 1997.
- [24] R. Clint Whaley. User contribution to atlas. [http://math-atlas.sourceforge.net/devel/atlas\\_contrib/](http://math-atlas.sourceforge.net/devel/atlas_contrib/) also available in `ATLAS/doc/atlas_contrib.pdf` of tarfile.
- [25] 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>.
- [26] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Super-Computing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.** [http://www.cs.utsa.edu/~whaley/papers/atlas\\_sc98.ps](http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps).
- [27] 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.
- [28] R. Clint Whaley and Antoine Petit. Atlas homepage. <http://math-atlas.sourceforge.net/>.
- [29] R. Clint Whaley and Antoine Petit. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. <http://www.cs.utsa.edu/~whaley/papers/spercw04.ps>.
- [30] R. Clint Whaley, Antoine Petit, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
- [31] R. Clint Whaley and David B. Whalley. Tuning high performance kernels through empirical compilation. In *The 2005 International Conference on Parallel Processing*, June 2005.