Automated Empirical Optimization of Software and the ATLAS Project *

R. Clint Whaley [†] Antoine Petitet [‡] Jack J. Dongarra, [§]

September 19, 2000

Abstract

This paper describes the ATLAS (Automatically Tuned Linear Algebra Software) project, as well as the fundamental principles that underly it. ATLAS is an instantiation of a new 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 new paradigm to linear algebra software, with the present emphasis on the Basic Linear Algebra Subprograms (BLAS), a widely used, performance-critical, linear algebra kernel library.

*This work was supported in part by: U.S. Department of Energy under contract number DE-AC05-96OR22464; National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615; University of California, Los Alamos National Laboratory, subcontract # B76680017-3Z; Department of Defense Raytheon E-Systems, subcontract# AA23, prime contract# DAHC94-96-C-0010; Department of Defense Nichols Research Corporation, subcontract#s NRC CR-96-0011 (ASC) and prime contract # DAHC-94-96-C-0005; Department of Defense Nichols Research Corporation, subcontract#s NRC CR-96-0011 (CEWES); prime contract # DAHC-94-96-C-0002

†Dept. of Computer Sciences, Univ. of TN, Knoxville, TN 37996, `rwhaley@cs.utk.edu`

‡Dept. of Computer Sciences, Univ. of TN, Knoxville, TN 37996, `petitet@cs.utk.edu`

§Dept. of Computer Sciences, Univ. of TN, Knoxville, TN 37996, and Mathematical Sciences Section, ORNL, Oak Ridge, TN 37831, `dongarra@cs.utk.edu`

Contents

Part I

Introduction

The ATLAS (Automatically Tuned Linear Algebra Software) project is an ongoing research effort focusing on applying empirical techniques in order to provide portable performance. Linear algebra routines are widely used in the computational sciences in general, and scientific modeling in particular. In many of these applications, the performance of the linear algebra operations are the main constraint preventing the scientist from modeling more complex problems, which would then more closely match reality. This then dictates an ongoing need for highly efficient routines; as more compute 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 optimized linear algebra routines available.

Linear algebra is rich in operations which are highly optimizable, in the sense that a highly tuned code may run multiple orders of magnitude faster than a naively coded routine. However, these optimizations are platform specific, such that an optimization for a given computer architecture will actually cause a slow-down on another architecture. The traditional method of handling this problem has been to produce hand-optimized routines for a given machine. This is a painstaking process, typically requiring many man-months of highly trained (both in linear algebra and computational optimization) personnel. The incredible pace of hardware evolution makes this technique untenable in the long run, particularly so when considering that there are many software layers (eg., operating systems, compilers, etc), which also effect these kinds of optimizations, that are changing at a similar, but independent rates.

Therefore a new paradigm is needed for the production of highly efficient routines in the modern age of computing, and ATLAS represents an implementation of such a set of new techniques. We call this paradigm "Automated Empirical Optimization of Software", or AEOS. In an AEOS-enabled package such as ATLAS, the package provides many ways of doing the required operations, and uses empirical timings in order to choose the best method for a given architecture. Thus, if written generally enough, an AEOS-aware package can automatically adapt to a new computer architecture in a matter of hours, rather than requiring months or even years of highly-trained professionals' time, as dictated by traditional methods.

ATLAS typically uses code generators (i.e., programs that write other programs) in order to provide the many different ways of doing a given operation, and has sophisticated search scripts and robust timing mechanisms in order to find the best ways of performing the operation for a given architecture.

One of the main performance kernels of linear algebra has traditionally been a standard API known as the BLAS (Basic Linear Algebra Subprograms) [14, 17, 6, 7, 5] This API is supported by hand-tuned efforts of many hardware vendors, and thus provides a good first target for ATLAS, as there is both a large audience for this API, and on those platforms where vendor-supplied BLAS exist, an easy way to determine if ATLAS can provide the required level of performance.

Part II
AEOS

1 AEOS in Context

Historically, the research community has pursued two separate paths towards the goal of making software run at near peak levels. The first and most generally successful of these builds on research into compilers and their associated technologies. The holy grail of compilation research is to take an arbitrary code as an input and produce completely optimal code as output for given languages and hardware platforms. Despite the immense amount of effort that has been poured into this approach, its success has been limited both by practical time constraints (viz. users will not tolerate compile-times that extend into several days) and by the amount of detailed information the compiler can obtain about the software to be compiled and the hardware on which it is supposed to execute [16, 23, 24].

A second, complementary thrust has been to identify kernel routines that constitute the dominant performance cost of a wide variety of applications. When such kernels can be identified and an API can be agreed upon by the members of the community, small groups of programmers with the required level of technical knowledge can concentrate their efforts on producing optimized kernel libraries for architectures of interest. A prime example of this kind of effort is the aforementioned BLAS. As experience with the BLAS has shown, these libraries can be produced by some combination of hardware vendors (e.g. IBM, Intel), independent software vendors (e.g. Cooke & Associates), and researchers, depending in large measure on the level of importance the different parties attach to the routines in question. Developers who write their code to call these well-known APIs can then achieve high performance across all supported architectures.

But just as there are currently boundaries to what can be done to achieve near peak performance via compiler optimization, the library-oriented approach has significant limitations as well. For instance, it is clear that in order to elicit the kind of attention required to create an optimized library for a given operation, the operation must be regarded as widely useful by the members of a programming community, who are usually already overburdened. Moreover, once an API has been agreed upon, support for various architectures becomes the dominant problem, especially since the kind of optimizations necessary to achieve high performance are by their very nature non-portable. Such performance tuning relies on a careful exploitation of the specific details the underlying hardware architecture; if that hardware is changed, a previous optimization may now cause the program to execute more slowly on the new platform.

The expensive and hardware-relative nature of kernel optimizations becomes all the more problematic when processor designs are changing at the remarkable pace dictated by Moore's Law. These increases in processor performance are, however, largely wasted unless the key libraries are updated at the same pace as the hardware. With ever-shrinking hardware generation cycles, these updates become nearly impossible to do by hand. It is a fact of the computing industry that by the time highly optimized code is available for a given architecture, that architecture is generally well on its way towards obsolescence.

We believe the AEOS methodologies address this problem directly, and have the potential to make a significant impact on how high performance libraries are produced and maintained.

2 Basic AEOS Requirements

The basic requirements for supporting a library using AEOS methodologies are:

- *Isolation of performance-critical routines.* Just as with traditional libraries, someone must find the performance-critical sections of code, separate them into subroutines, and choose an appropriate 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 routines 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 code 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 3 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, with sophisticated code 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 factorial 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.

3 Methods of Software Adaptation

There are essentially two 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 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 which do 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 implementor 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. 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 a 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 *code generation*. In code generation, a code generator (i.e., a program that writes other programs) is produced. This code 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 code generator produces source code with the requisite characteristics. The great strength of code 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 routines rapidly become almost insurmountable barriers to outside contribution.

It thus seems likely that the best approach will combine these two methods; in such a system code generation will be harnessed for its generality, but it will be supplemented by multiple implementation, allowing for the extension of the package beyond one designer's vision via community collaboration.

4 Adapting to Constantly Changing Hardware and Software Layers

As mentioned in the introduction, AEOS design for libraries is essentially an outgrowth of the rapid pace of hardware evolution. However, all layers of software separating the AEOS package from the hardware comprise what could be called the library-perceived architecture. Therefore, AEOS-centric packages adapt to both software and hardware, which change at only loosely related rates.

The history of the ATLAS project includes numerous examples of the lack of synchrony between software and hardware releases. Gnu gcc on the UltraSparc shows one such case. The first available release of gcc for this new (at the time) hardware still used a previous ISA (instruction set architecture), so that only 16 of the available 32 floating point registers were addressable to codes written in C, as ATLAS is. This meant that at that point in the hardware/software cycle, the best case of register blocking (an important optimization for floating point intensive software) used roughly 16 registers, not the ISA limit of 32. Later on, gcc was adapted to the new ISA, and ATLAS's new optimal case used the increased numbers of registers to further improve the software.

In the traditional way of supporting libraries, this would have required the development of two libraries, each requiring significant investment of time and effort from highly trained professionals. With ATLAS, the optimized library was available within hours of even a relatively unsophisticated user gaining access to the new hardware or software.

Generally, the lower-level the language an AEOS package is implemented, the smaller the gap between software and hardware release cycles are. For instance, it is likely that an assembler would be able to address the new ISA before a compiler, such as C or Fortran77, will. However, implementing in high level languages has at least two distinct advantages. First, the software is much more portable, and secondly, the excellent and ongoing research into compiler technology is leveraged automatically. Examples of the way that compiler optimizations can aid AEOS programming abound. Compiler-controlled prefetch, loop unrolling, pipelining and loop skewing are just a few of the obvious areas. Of course, many such optimizations may be done explicitly by the code generator as well. In practice, a code generator which can explicitly do such optimizations, while also being capable of generating code without such optimizations in order to allow the compiler the opportunity to perform them implicitly, will enjoy the most general success.

4.1 AEOS/Compiler Synergy

This brings us to an interesting topic, that of AEOS/compiler synergy. As previously mentioned, AEOS can readily leverage the improvements inherent in compiler advancement. The reverse is also true. As languages become higher level, they implicitly begin to rely on standard libraries for a larger and larger proportion of their performance. An already existing example of this would be Fortran95's addition of matrix multiply and matrix-vector multiplies as language primitives. These kinds of high-level abstractions will naturally call a library, and AEOS techniques can provide the adaptability required in today's compiler life cycles.

This kind of mixture of compiler, language, and libraries can be mined much more extensively in the search for synergy between these disciplines, as with the research presently being done on telescoping languages [16].

Finally, we believe that many of the empirical techniques incorporated in AEOS will eventually make their way into compilers. For instance, if a very high level of optimizations is set, a compiler could generate and time various unrollings, etc, just as is done presently in ATLAS, in order to find the best for the given operation. I.e., in cases where heavy use makes the cost worthwhile, the almost purely a priori techniques presently used by compilers can be supplemented with what amounts to automated tuning on the fly. Interpreted languages using techniques similar to Java's hot-spot can iteratively improve code in a like manner across multiple calls.

No matter how good compilers get, it is unlikely that the need for optimized libraries, and thus of AEOS, will ever go away (although their use may no longer be apparent to the programmer, as in a paradigm such as telescoping languages). The hard limits of what a compiler can do will always be dictated by the amount of information the compiler can extract from the provided code. Library building, where an operation is dictated (as opposed to an implementation of that operation), with its associated enormously expanded high-level understanding, allows for much greater variance in implementation.

Part III
ATLAS

ATLAS is the project from which our current understanding of AEOS methodologies grew, and now provides a test bed for their further development and testing. ATLAS was not, however, the first project to harness AEOS-like techniques for library production and maintenance. As far as we know, the PHiPAC [3] project was the first attempt to apply them to Linear Algebra. The FFTW [11, 9, 10] project, which uses AEOS techniques to optimize Fast Fourier transforms, was also released around this time frame. Other projects with AEOS-like designs include [18, 20, 19]. The philosophies, approach and application success of these projects vary widely, but they are all built around the idea of using empirical results and some degree of automation to adapt libraries for greater performance.

The initial goal of ATLAS was to provide a portably efficient implementation of the BLAS. ATLAS now provides at least some level of support for all of the BLAS, and the first tentative extensions beyond this one API have been taken (for example, the most recent ATLAS release contained some higher level routines from the LAPACK [1] API). Due to space limitations, this paper will concentrate on ATLAS's BLAS support.

The BLAS (Basic Linear Algebra Subroutines) are building block routines for performing basic vector and matrix operations. The BLAS are divided into three levels: Level 1 BLAS do vector-vector operations, Level 2 BLAS do matrix-vector operations, and the Level 3 BLAS do matrix-matrix operations. The performance gains from optimized implementations is strongly affected by the level of the BLAS.

Level 1 BLAS, where no memory reuse is possible, gain only minuscule speedups from all but the best implementations (as a back of envelope estimate, consider these speedups to typically be in the range of 0-15%). Essentially, the only optimizations to be done at this level involve floating point unit usage, loop optimizations, etc. However, since these routines are very simple, the compiler can usually do an excellent job of these optimizations, so real performance gains are typically found only when a compiler is poorly adapted to a given platform.

In the Level 2 BLAS, memory blocking can allow for reuse of the vector operands, but not, in general, of the matrix operand (the exception is that some matrix types, for instance symmetric or Hermitian, can effectively use each matrix operand twice). Reducing the vector operands from $O(N^2)$ to $O(N)$ represents considerable savings over naive code, but due to the irreducible matrix costs, the memory load remains of the same order ($O(N^2)$) as the operation count. Therefore, the Level 2 BLAS can enjoy modest speedup (say, roughly in the range of 10-300%), both because memory blocking is effective, and because the loops are complex enough that more compilers begin having problems doing the floating point optimizations automatically.

Finally, the Level 3 BLAS can display orders of magnitude speedups. To simplify greatly, these operations can be blocked such that the natural $O(N^3)$ fetch costs become essentially $O(N^2)$. Further, the triply-nested loops used here are almost always too complex for the compiler to figure out without hints from the programmer (eg, some explicit loop unrolling), and thus the $O(N^3)$ computation cost can be greatly optimized as well.

The following sections discuss our handling of the Level 3 and 2 BLAS in ATLAS. Because of the amount of effort required to provide high-quality AEOS software, it becomes critical to find the smallest possible kernels which can be leveraged to supply all required functionality. Thus, each section describes the low level performance kernels, the techniques used to create them, and how these kernels are utilized to produce all required functionality. The Level 1 BLAS are not discussed; at present ATLAS provides hand-tuned codes for these operations, essentially relying on the compiler for the lion's share of the optimization.

5 Limits of ATLAS's Approach

As previously mentioned, any AEOS approach is bound to have some restrictions on its adaptability. ATLAS is no exception, and the following assumptions need to hold true for ATLAS to perform well:

1. *Adequate ANSI C compiler* ATLAS is written entirely in ANSI/ISO C, with the exception of the Fortran77 interface codes (which are simple wrappers written in ANSI Fortran77, calling the C internals for computation). ATLAS does not require an excellent compiler, since it uses code generation to perform many optimizations typically done by compilers. However, too-aggressive compilers can transform already optimal code into suboptimal code, if flags do not exist to turn off certain compiler optimizations. On the other hand, compilers without the ability to effectively use the underlying ISA (eg., inability to utilize registers, even when the C code calls for them), will yield poor results as well.

2. *Hierarchical memory* ATLAS assumes a hierarchical memory is present. Best results will be obtained when both registers and at least an L1 data cache are present.

Of these two restrictions, the most important is the need for an adequate C compiler. Lack of hierarchical memory would at worst turn some of ATLAS's blocking and register usage into overheads. Even with this handicap, ATLAS's code adaptation may still yield enough performance to provide an adequate BLAS. If the ANSI C compiler is poor enough, however, this can result in the computational portion of the algorithms being effectively unoptimized. Since the computational optimizations are the dominant cost of a blocked Level 3 BLAS, this can produce extremely poor results.

6 Level 3 BLAS Support in ATLAS

As previously mentioned, all Level 3 BLAS routines (for each real data type there are six Level 3 BLAS, and nine routines for each complex data type) can be efficiently implemented given an efficient matrix-matrix multiply (hereafter shortened to matmul, or the BLAS matmul routine name, GEMM). Thus the main performance kernel is GEMM. As subsequent sections show, however, GEMM itself is further narrowed down to an even smaller kernel before code generation takes place.

The BLAS supply a routine GEMM, which performs a general matrix-matrix multiplication of the form $C \leftarrow \alpha op(A) op(B) + \beta C$, where $op(X) = X$ or X^T. C is an $M \times N$ matrix, and $op(A)$ and $op(B)$ are matrices of size $M \times K$ and $K \times N$, respectively.

In general, the arrays A, B, and C will be too large to fit into cache. Using a block-partitioned algorithm for matrix multiply it is still possible to arrange for the operations to be performed with data for the most part in cache by dividing the matrix into blocks. For additional details see [8].

Using this BLAS routine, the rest of the Level 3 BLAS can be efficiently supported, so GEMM is the Level 3 BLAS computational kernel. ATLAS supports this kernel using both parameterized adaptation and code generation. There are hand-written high-level codes that use compile- or run-time variables to adapt to machines. These high level codes utilize a generated L1 (Level 1) cache-contained matrix multiply as their kernel.

6.1 Building the General Matrix Multiply From the L1 Cache-contained Multiply

This section describes the non-generated code, whose only variance across platforms come from parameterization. These codes are used to form the BLAS's general matrix-matrix multiply using a L1 cache-contained matmul (hereafter referred to as the L1 matmul).

Section 6.2 describes the L1 matmul and its generator in detail. For our present discussion, it is enough to know that ATLAS has at its disposal highly optimized routines for doing matrix multiplies whose dimensions are chosen such that cache blocking is not required (i.e., the hand-written code discussed in this section deals with cache blocking; the generated code assumes things fit into cache).

When the user calls GEMM, ATLAS must decide whether the problem is large enough to tolerate copying the input matrices A and B. If the matrices are large enough to support this $O(N^2)$ overhead, ATLAS will copy A and B into block-major format. ATLAS's block-major format breaks up the input matrices into contiguous blocks of a fixed size N_B, where

N_B is chosen as discussed in section 6.2 in order to maximize L1 cache reuse. Once in block-major format, the blocks are contiguous, which eliminates TLB problems, minimizes cache thrashing and maximizes cache line use. It also allows ATLAS to apply alpha (if alpha is not already one) to the smaller of A or B, thus minimizing this cost as well. Finally, the package can use the copy to transform the problem to a particular transpose setting, which for load and indexing optimization, is set so A is copied to transposed form, and B is in normal (non-transposed) form. This means our L1-cache contained code is of the form $C \leftarrow A^T B$, $C \leftarrow A^T B + C$, and $C \leftarrow A^T B + \beta C$, *where all dimensions, including the non-contiguous stride, are known to be* N_B. Knowing all of the dimensions of the loops allows for arbitrary unrollings (i.e., if the instruction cache could support it, ATLAS could unroll all loops completely, so that the L1 cache-contained multiply had no loops at all). Further, when the code generator knows leading dimension of the matrices (i.e., the row stride), all indexing can be done up front, without the need for expensive integer or pointer computations.

If the matrices are too small, the $O(N^2)$ data copy cost can actually dominate the algorithm cost, even though the computation cost is $O(N^3)$. For these matrices, ATLAS will call an L1 matmul which operates on non-copied matrices (i.e. directly on the user's operands). The non-copy L1 matmul will generally not be as efficient as the copy L1 matmul; at this problem size the main drawback is the additional pointer arithmetic required in order to support the user-supplied leading dimension.

The choice of when a copy is dictated and when it is prohibitively expensive is an AEOS parameter; it turns out that this crossover point depends strongly both on the particular architecture, and the shape of the operands (matrix shape effectively sets limits on which matrix dimensions can enjoy cache reuse). To handle this problem, ATLAS simply compares the speed of the copy and non-copy L1 matmul for variously shaped matrices, varying the problem size until the copying provides a speedup (on some platforms, and with some shapes, this point is never reached). These crossover points are determined at install time, and then used to make this decision at runtime. Because it is the dominant case, this paper describes only the copied matmul algorithm in detail.

There are presently two algorithms for performing the general matrix-matrix multiply. The two algorithms correspond to different orderings of the loops; i.e., is the outer loop over M (over the rows of A), and thus the second loop is over N (over the columns of B), or is this order reversed. The dimension common to A and B (i.e., the K loop) is currently always the innermost loop.

Let us define the input matrix looped over by the outer loop as the outer or outermost matrix; the other input matrix will therefore be the inner or innermost matrix. Both algorithms have the option of writing the result of the L1 matmul directly to the matrix, or to an output temporary $\hat{C}$. The advantages to writing to $\hat{C}$ rather than C are:

1. address alignment may be controlled (i.e., the code can ensure during the malloc that $\hat{C}$ begins on a cache-line boundary)

2. Data is contiguous, eliminating possibility of unnecessary cache-thrashing due to ill-chosen leading dimension (assuming a non-write-through cache)

The disadvantage of using $\hat{C}$ is that an additional write to C is required after the L1 matmul operations have completed. This cost is minimal if GEMM makes many calls to the

L1 matmul (each of which writes to either C or $\hat{C}$), but can add significantly to the overhead when this is not the case. In particular, an important application of matrix multiply is the rank-K update, where the write to the output matrix C can be a significant portion of the cost of the algorithm. For the rank-K update, writing to $\hat{C}$ essentially doubles the write cost, which is clearly unacceptable. The routines therefore employ a heuristic to determine if the number of times the L1 matmul will be called in the K loop is large enough to justify using $\hat{C}$, otherwise the answer is written directly to C.

Regardless of which matrix is outermost, both algorithms try to allocate enough space to store $N_B \times N_B$ output temporary, $\hat{C}$ (if needed), 1 panel of the outermost matrix, and the entire inner matrix. If this fails, the algorithms attempt to allocate smaller work arrays, the smallest acceptable workspace being enough space to hold $\hat{C}$, and 1 panel from both A and B. The minimum workspace required by these routines is therefore $2KN_B$, if writing directly to C, and ${N_B}^2 + 2KN_B$ if not. If this amount of workspace cannot be allocated, the previously mentioned non-copy code is called instead.

If there is enough space to copy the entire innermost matrix, there are several benefits to doing so:

- Each matrix is copied only one time
- If all of the workspaces fit into L2 cache, the algorithm enjoys complete L2 reuse on the innermost matrix
- Data copying is limited to the outermost loop, protecting the inner loops from unneeded cache thrashing

Of course, even if the allocation succeeds, using too much memory might result in unneeded swapping. Therefore, the user can set a maximal amount of workspace that ATLAS is allowed to have, and ATLAS will not try to copy the innermost matrix if this maximum workspace requirement is exceeded.

If enough space for a copy of the entire innermost matrix is not allocated, the innermost matrix will be entirely copied for each panel of the outermost matrix (i.e., if A is our outermost matrix, ATLAS will copy B $\lceil M/N_B \rceil$ times). Further, our usable L2 cache is reduced (the copy of a panel of the innermost matrix will take up twice the panel's size in L2 cache; the same is true of the outermost panel copy, but that will only be seen the first time through the secondary loop).

Regardless of which looping structure or allocation procedure used, the inner loop is always along K. Therefore, the operation done in the inner loop by both routines is the same, and it is shown in figure 1.

If GEMM is writing to $\hat{C}$, the following actions are performed in order to calculate the $N_B \times N_B$ block $C_{i,j}$, where i and j are in the range $0 \le i < \lceil M/N_B \rceil$, $0 \le j < \lceil N/N_B \rceil$:

1. Call L1 matmul of the form $C \leftarrow AB$ to multiply block 0 of the row panel i of A with block 0 of the column panel j of B.
2. Call L1 matmul of form $C \leftarrow AB + C$ to multiply block k of the row panel i of A with block k of the column panel j of B, $\forall k, 1 \le k < \lceil K/N_B \rceil$. The L1 matmul is performing the operation $C \leftarrow AB + C$, so as expected this results in multiplying the row panel of A with the column panel of B.

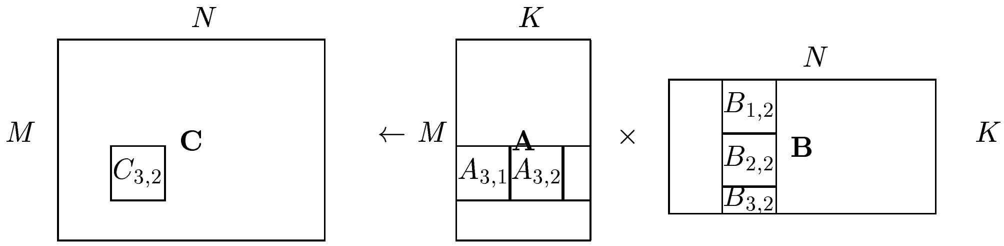

Figure 1: One step of matrix-matrix multiply

3. $\hat{C}$ now holds the product of the row panel of A with the column panel of B, so ATLAS now performs the block write-back operation $C_{i,j} \leftarrow \hat{C}_{i,j} + \beta C_{i,j}$.

If ATLAS is writing directly to C, this action becomes:

1. Call L1 matmul of the correct form based on user-defined β (eg. if $\beta == -1$, use $C \leftarrow AB - C$) to multiply block 0 of the row panel i of A with block 0 of the column panel j of B.

2. Call L1 matmul of form $C \leftarrow AB + C$ to multiply block k of the row panel i of A with block k of the column panel j of B, $\forall k, 1 \leq k < \lceil K/N_B \rceil$.

Building from this inner loop, ATLAS has differing loop orderings which provide two algorithms for the full matmul. Figures 2 and 3 give the pseudo-code for these two algorithms, assuming the write is directly to C (writing to $\hat{C}$ is only trivially different). For simplicity, this pseudo-code skips the cleanup necessary for cases where dimensions do not evenly divide N_B. The matrix copies are shown as if coming from the notranspose, notranspose case. If they do not, only the array access on the copy changes.

```
work = allocate((M+NB)*K)
if (allocated(work)) then
   PARTIAL_MATRIX = .FALSE.
   copy A into block major format
else
   PARTIAL_MATRIX = .TRUE.
   work = allocate(NB*2*K)
   if (.NOT.allocated(work)) call small_case_code
   return
end if
NBNB = NB * NB
do j = 1, N, NB
   Bwork = ALPHA*B(:,J:J+NB-1); Bwork in block major format
   do i = 1, M, NB
      if (PARTIAL_MATRIX) Awork = A(i:i+NB-1,:); Awork in block major format
      ON_CHIP_MATMUL(Awork(1:NB*NB), Bwork(1:NB*NB), BETA, C(i:i+NB-1, j:j+NB-1), ldc)
      do k = 2, K, NB
         ON_CHIP_MATMUL(Awork((k-1)*NBNB+1:k*NBNB), Bwork((k-1)*NBNB+1:k*NBNB),
                        1.0, C(i:i+NB-1, j:j+NB-1), ldc)
      end do
   end do
end do
```

Figure 2: General matrix multiplication with A as innermost matrix

```
work = allocate(N*K + NB*K)
if (allocated(work)) then
   PARTIAL_MATRIX = .FALSE.
   copy B into block major format
else
   PARTIAL_MATRIX = .TRUE.
   work = allocate(NB*2*K)
   if (.NOT.allocated(work)) call small_case_code
   return
end if
NBNB = NB * NB
do i = 1, M, NB
      Awork = ALPHA*A(i:i+NB-1,:); Awork in block major format
   do j = 1, N, NB
      if (PARTIAL_MATRIX) Bwork = B(:,J:J+NB-1); Bwork in block major format
      ON_CHIP_MATMUL(Awork(1:NBNB), Bwork(1:NBNB), BETA,
                     Cwork(i:i+NB-1, j:j+NB-1), ldc)
      do k = 2, K, NB
         ON_CHIP_MATMUL(Awork((k-1)*NBNB+1:k*NBNB), Bwork((k-1)*NBNB+1:k*NBNB),
                        1.0, Cwork(i:i+NB-1, j:j+NB-1), ldc)
      end do
   end do
end do
```

Figure 3: General matrix multiplication with B as innermost matrix

6.1.1 Choosing the Correct Looping Structure

When the call to the matrix multiply is made, the routine must decide which loop structure to call (i.e., which matrix to put as outermost). If the matrices are of different size, L2 cache reuse can be encouraged by deciding the looping structure based on the following criteria:

1. If either matrix will fit completely into the usable L2 cache, put it as the innermost matrix (algorithm gets L2 cache reuse on the entire inner matrix)

2. If neither matrix fits completely into L2 cache, put largest matrix as the outermost matrix (algorithm gets L2 cache reuse on the panel of the outer matrix, if it fits in cache, and memory usage is minimized)

The size of the usable L2 cache is not directly known by ATLAS (although the AEOS variable `CacheEdge` described in section 6.1.2 will often serve the same purpose) and so these criteria are not presently used for this selection. Rather, in order to minimize workspace, and maximize the chance that condition one above occurs, the smallest matrix will always be used as the innermost matrix. If both matrices are the same size, A is selected as the innermost matrix (this implies a better access pattern for C).

6.1.2 Blocking for Higher Levels of Cache

Note that this paper defines the Level 1 (L1) cache as the "lowest" level of cache: the one closest to the processor. Subsequent levels are "higher": further from the processor and thus usually larger and slower. Typically, L1 caches are relatively small (eg., 8-32KB), employ least recently used replacement policies, have separate data and instruction caches, and are often non-associative and write-through. Higher levels of cache or more often non-write-through, with varying degrees of associativity, differing replacement polices, and combined instruction and data cache.

ATLAS detects the actual size of the L1 data cache. However, due to the wide variance in high level cache behaviors, in particular the difficulty of determining how much of such caches are usable after line conflicts and data/instruction partitioning is done, ATLAS does not presently detect and use a explicit Level 2 cache size as such. Rather, ATLAS employs a empirically determined value called `CacheEdge`, which represents the amount of the cache that is usable by ATLAS for its particular kind of blocking.

Explicit cache blocking for the selected level of cache is only required when the cache size is insufficient to hold the two input panels and the $N_B \times N_B$ piece of C. This means that users will have optimal results for many problem sizes without employing `CacheEdge`. This is expressed formally below; Notice that conditions 1 and 2 below do not require explicit cache blocking, so the user gets this result even if `CacheEdge` is not set.

Therefore, the explicit cache blocking strategy discussed in 4 below assumes that the panels of A and B overflow a particular level of cache. In this case, the problem can be easily partitioned along the K dimension of the input matrices such that the panels of the partitioned matrices A_p and B_p *will* fit into the cache. This means that we get cache reuse on the input matrices, at the cost of writing C additional times.

It is easily shown that the footprint of the algorithm computing a $N_B \times N_B$ section of C in cache is roughly $2KN_B + {N_B}^2$, where $2KN_B$ stores the panels from A and B, and the section of C is of size ${N_B}^2$. If the above expression is set equal to `CacheEdge`, and solved for K, it will yield the maximal K (call this quantity K_m) which will, assuming the inner matrix was copied up front, allow for reusing the outer matrix panel N/N_B times. This partitioning transforms the original matrix multiply into $\lceil K/K_m \rceil$ rank-K_m updates.

Since the correct value of `CacheEdge` is not known a priori, ATLAS empirically determines it at install time by using large matrices (whose panel sizes can be expected to overflow the cache, and thus induce the need for explicit, rather than implicit, L2 or higher blocking), and simply tries various settings. Extremely large caches will probably not be detected in this manner (i.e., if the user cannot allocate enough memory to cause a panel to overflow the cache, the large cache will not be detected), in which case `CacheEdge` will not be set or used (very large caches will have implicit cache reuse for all but the largest matrices anyway). Some caches will not give a clear enough optimization using `CacheEdge` for timings to reliably detect the difference, and in these cases, where no noticeable benefit is detected, `CacheEdge` will not be set or used.

Assuming that matrix A is the innermost matrix, and we are discussing cache level L, of size S_L, and that main memory is classified as a level of "cache" greater than L, there are four possible states (depending on cache and problem size, and whether `CacheEdge` is set) which ATLAS may be in. These states and their associated memory access costs are:

1. If the entire inner matrix, a panel of the outer matrix, and the $N_B \times N_B$ section of C fits into the cache (eg. $MK + KN_B + {N_B}^2 \leq S_L$)
   - $K(M+N)+MN$ reads (of A, B and C, respectively) from higher level(s) cache
   - $\frac{MNK}{N_B}$ writes to first level of non-write-through cache; higher levels of cache receive only the final MN writes
2. If the cache cannot satisfy the memory requirements of 1, it may still be large enough to accommodate the two active input panels, along with the relevant section of C (eg., ($2KN_B + {N_B}^2 \leq S_L$ AND ATLAS copies the entire inner matrix) OR ($3KN_B + {N_B}^2 \leq S_L$ AND ATLAS copies a panel of the inner matrix in the inner loop, thus doubling the inner panel's footprint in the cache))
   - $NK + \frac{MNK}{N_B} + MN$ reads (B, A and C, respectively) from higher level(s) of cache
   - $\frac{MNK}{N_B}$ writes to first level of non-write-through cache; higher levels of cache receive only the final MN writes
3. If the cache is too small for either of the previous cases to hold true, (eg., $2KN_B + {N_B}^2 > S_L$) and `CacheEdge` is not set, and thus no explicit level L blocking is done, the memory access becomes:
   - $\frac{2MNK}{N_B} + MN$ reads (A, B, and C) from higher level(s) of cache
   - $\frac{MNK}{N_B}$ writes to first level of non-write-through cache; higher levels of cache receive only the final MN writes
4. Finally, if the first two cases do not apply (eg., $2KN_B + {N_B}^2 > S_L$), but `CacheEdge` is set to S_L, ATLAS can perform cache blocking to change the memory access from that given in 3 to:
   - $NK + \frac{MNK}{N_B} + \frac{MNK}{K_m}$ (B, A, C) reads from higher level(s) of cache

- $\frac{MNK}{N_B}$ writes to first level of non-write-through cache; higher levels of cache receive at most $\frac{MNK}{K_m}$ writes

As mentioned above, case 4 is only used if `CacheEdge` has been set, and cases 1 and 2 do not apply (i.e, it is used as an alternative to case 3). At first glance, changing case 3 to 4 may appear to be a poor bargain indeed, particularly since writes are generally more expensive than reads. There are, however, several mitigating factors that make this blocking nonetheless worthwhile. If the cache is write-through, 4 does not increase writes over 3, so it is a clear win. Second, ATLAS also does not allow $K_m < N_B$, and in many cases $K_m \gg N_B$, so the savings are well worth having. With respect to the expense of writes, the writes are not flushed immediately; This fact has two important consequences:

1. The cache can schedule the write-back during times when the algorithm is not using the bus.

2. Writes may be written in large bursts, which significantly reduces bus traffic; this can tremendously optimize writing on some systems

In practice, 4 has been shown to be at least roughly as good as 3 on all platforms. The amount of actual speedup varies widely depending on problem size and architecture. On some systems the speedup is negligible; on others it can be significant: for instance, it can make up to 20% difference on DEC 21164 based systems (which have three layers of cache). Note that this 20% improvement is merely the difference between cases 3 and 4, not between ATLAS and some naive implementation, for instance.

The analysis given above may be applied to any cache level greater than 1; it is not for level 2 caches only. However, this analysis is accurate only for the algorithm used by ATLAS in a particular section of code, so it is not possible to recur in order to perform explicit cache blocking for arbitrary levels of cache. To put this another way, ATLAS explicitly blocks for L1, and only one other higher level cache. If an architecture has 3 levels of cache, ATLAS can explicitly block for L1 and L2, or L1 and L3, but not all three.

If ATLAS performs explicit cache blocking for level L, that does not mean that level $L+1$ would be useless; depending on cache size and replacement policy, level $L+1$ may still save extra read and writes to main memory through implicit cache blocking.

6.2 L1 Cache-contained Matmul

The only code generator required to support the Level 3 BLAS produces a L1 cache-contained matmul. The operation supported by the kernel is still: $C \leftarrow \alpha op(A)op(B) + \beta C$, where $op(X) = X$ or X^T. C is an $M \times N$ matrix, and $op(A)$ and $op(B)$ are matrices of size $M \times K$ and $K \times N$, respectively. However, by L1 cache-contained we mean that the dimensions of its operands have been chosen such that Level 1 cache reuse is maximized (see below for more details). Therefore, the generated code blocks for the L1 cache using the dimensions of its operand matrices (M, N, and K), which, when not in the cleanup section of the algorithm, are all known to be N_B.

In a multiply designed for L1 cache reuse, one of the input matrices is brought completely into the L1 cache, and is then reused in looping over the rows or columns of the other input matrix. The present code brings in the matrix A, and loops over the columns of B; this was an arbitrary choice, and there is no theoretical reason it would be superior to bringing in B and looping over the rows of A.

There is a common misconception that cache reuse is optimized when both input matrices, or all three matrices, fit into L1 cache. In fact, the only win in fitting all three matrices into L1 cache is that it is possible, assuming the cache is not write-through, to save the cost of pushing previously used sections of C back to higher levels of memory. Often, however, the L1 cache *is* write-through, while higher levels are not. If this is the case, there is no way to minimize the write cost, so keeping all three matrices in L1 does not result in greater cache reuse.

Therefore, ignoring the write cost, maximal cache reuse for our case is achieved when all of A fits into cache, with room for at least two columns of B and 1 cache line of C. Only one column of B is actually accessed at a time in this scenario; having enough storage for two columns assures that the old column will be the least recently used data when the cache overflows, thus making certain that all of A is kept in place (this obviously assumes the cache replacement policy is least recently used).

While cache reuse can account for a great amount of the overall performance win, it is obviously not the only factor. The following sections outline some of these non-data cache related optimizations.

6.2.1 Instruction Cache Reuse

Instructions are cached, and it is therefore important to fit the L1 matmul's instructions into the L1 instruction cache. This means optimizations that generate massive amount of instruction bloat (completely unrolling all three loops, for instance) cannot be employed.

6.2.2 Floating Point Instruction Ordering

When this paper discusses floating point instruction ordering, it will usually be in reference to *latency hiding*, and its associated *loop skewing*.

Most modern architectures possess pipelined floating point units. This means that the results of an operation will not be available for use until X cycles later, where X is the number of stages in the floating point pipe (typically somewhere around 3-8). Remember that our L1 matmul is of the form $C \leftarrow A^T B + C$; individual statements would then naturally be some variant of `C[X] += A[Y] * B[Z]`. If the architecture does not possess a fused multiply/add unit, this can cause an unnecessary execution stall. The operation `register = A[Y] * B[Z]` is issued to the floating point unit, and the add cannot be started until the result of this computation is available, X cycles later. Since the add operation is not started until the multiply finishes, the floating point pipe is not utilized.

The solution is to remove this dependence by separating the multiply and add, and issuing unrelated instructions between them (requiring the loop to be skewed, since the multiply must now be issued X cycles before the add, which comes X cycles before the store). This reordering of operations can be done in hardware (out-of-order execution) or by the compiler, but this will oftentimes generate code that is not as efficient as doing it explicitly. More importantly, not all platforms have this capability (for example, gcc on a Pentium), and in this case the performance win can be large.

6.2.3 Reducing Loop Overhead

The primary method of reducing loop overhead is through loop unrolling. If it is desirable to reduce loop overhead without changing the order of instructions, one must unroll the loop over the dimension common to A and B (i.e., unroll the K loop). Unrolling along the other dimensions (the M and N loops) changes the order of instructions, and thus the resulting memory access patterns.

6.2.4 Exposing Parallelism

Many modern architectures have multiple floating point units. There are two barriers to achieving perfect parallel speedup with floating point computations in such a case. The first is a hardware limitation, and therefore out of our hands: All of the floating point units will need to access memory, and thus, for perfect parallel speedup, the memory fetch will usually also need to operate in parallel.

The second prerequisite is that the compiler recognize opportunities for parallelization, and this is amenable to software control. The fix for this is the classical one employed in such cases, namely unrolling the M and/or N loops, and choosing the correct register allocation so that parallel operations are not constrained by false dependencies.

6.2.5 Finding the Correct Number of Cache Misses

Any operand that is not already in a register must be fetched from memory. If that operand is not in the L1 cache, it must be fetched from further up in the memory hierarchy, possibly resulting in large delays in execution. The number of cache misses which can be issued simultaneously without blocking execution varies between architectures. To minimize memory costs, the maximal number of cache misses should be issued each cycle, until all memory is in cache or used. In theory, one can permute the matrix multiply to ensure that this is true. In practice, this fine a level of control would be difficult to ensure (there would be problems with overflowing the instruction cache, and the generation of such a precise instruction sequence, for instance). So the method ATLAS uses to control the cache-hit ratio is the more classical one of M and N loop unrolling.

6.2.6 Code Generator Parameters

The code generator is heavily parameterized in order to allow for flexibility in all of the areas. In particular, the options are:

- Support for A and/or B being either standard form, or stored in transposed form
- Register blocking of "outer product" form (the most optimal form of matmul register blocking). Varying the register blocking parameters provides many different implementations of matmul. The register blocking parameters are:
  - a_r : registers used for elements of A,
  - b_r : registers used for elements of B

  Outer product register blocking then implies that $a_r \times b_r$ registers are then used to block the elements of C. Thus, if N_r is the maximal number of registers discovered during the floating point unit probe, the search needs to try all a_r and b_r that satisfy $a_r b_r + a_r + b_r \leq N_r$.

- Loop unrollings: Their are three loops involved in matmul, one over each of the provided dimensions (M, N and K), each of which can have its associated unrolling factor (m_u, n_u, k_u). The M and N unrolling factors are restricted to varying with the associated register blocking (a_r and b_r, respectively), but the K-loop may be unrolled to any depth (i.e., once a_r is selected, m_u is set as well, but k_u is an independent variable).

- Choice of floating point instruction:
  - Combined multiply/add with required pipelining
  - Separate multiply and add instructions, with associated pipelining and loop skewing

- User choice of utilizing generation-time constant or run-time variables for all loop dimensions (M, N, and K; for non-cleanup copy L1 matmul, $M = N = K = N_B$). For each dimension that is known at generation, the following optimizations are made:
  - If unrolling meets or exceeds the dimension, no actual loop is generated (no need for loop if fully unrolled)
  - If unrolling is non-one, correct cleanup can be generated without using an if (thus avoiding branching within the loop)

  Even if a given dimension is a run-time variable, the generator can be told to assume particular, no, or general-case cleanup for arbitrary unrolling.

- For each operand array, the leading dimension can be either a generation time constant (for example, it is known to be N_B for copied L1 matmul), with associated savings in indexing computations, or it may be a run-time variable.

- For each operand array, the leading dimension can have a stride (stride of 1 is most common, but stride of 2 can be used to support complex arithmetic).

- The generator can eliminate unnecessary arithmetic by generating code with special alpha (1, -1, and variable) and beta (0, 1, -1, and variable) cases. In addition, there is a special case for when alpha and beta are both variables, but it is safe to divide beta by alpha (this can save multiple applications of alpha).

- Various fetch patterns for loading A and B registers

6.2.7 Putting It All Together – Outline of the Search Heuristic

It is obvious that with this many interacting effects, it would be difficult, if not impossible to predict a priori the best blocking factor, loop unrolling etc. Our approach is to provide a code generator coupled with a timer routine which takes in some initial information, and then tries different strategies for loop unrolling and latency hiding and chooses the case which demonstrated the best performance.

The timers are structured so that operations have a large granularity, leading to fairly repeatable results even on non-dedicated machines, and all intermediate results are written to output files so that interrupted installs may be restarted from the point of interruption.

The first step of the timing figures the size of the L1 cache. This is done by performing a fixed number of memory references, while successively reducing the amount memory addressed. The most significant gap between timings for successive memory sizes is declared to mark the L1 cache boundary. For speed, only powers of 2 are examined. This means that a 48K cache would probably be detected as a 32K cache, for instance. We have not found this problem severe enough to justify the additional installation time it would take to remedy it.

Next, ATLAS probes to determine information regarding the floating point units of the platform. First ATLAS needs to understand whether the architecture possesses a combined muladd unit, or if independent multiply and add pipes are required. To do this, ATLAS generates simple register-to-register code which performs the required multiply-add using a combined muladd and separate multiply and add pipes. Both variants are tried using code which implies various pipeline lengths. ATLAS then replicates the best of these codes in such a way that increasing numbers of independent registers are required, until performance drops off sufficiently to demonstrate that the available floating point registers have been exceeded. With this data in hand, ATLAS is ready to begin actual L1 matmul timings.

These general timings give ATLAS the L1 cache size, the kind of floating point instructions to issue (muladd or separate multiply and add), the pipeline depth, and a rough idea of the number of floating point registers. Given the size of the L1 cache, ATLAS is able choose the relevant range of blocking factors to examine. Knowing the type of floating point instruction the underlying hardware needs cuts the cases to be searched in half, while the maximum number of registers implies what register blockings are feasible, which in turn dictates the M and/or N loop unrollings to perform. Thus, the matmul search (and indeed many other searches) is shortened considerably by doing these general architecture probes.

In practice, K loop unrollings of 1 or K have tended to produce the best results. Thus ATLAS times only these two K loop unrolling during our initial search. This is done to reduce the length of install time. At the end of the install process, ATLAS attempts to ensure optimal K unrollings have not been missed by trying a wide range of K loop unrolling factors with the best case code generated for the unrollings factors of 1 or K.

The theoretically optimal register blocking in terms of maximizing flops/load are the near-square cases that satisfy the aforementioned equation $a_r b_r + a_r + b_r \leq N_r$ (see section 6.2.6 for details). Since the ATLAS generator requires that $a_r = m_u$ and $b_r = n_u$, these M and N loop unrollings are then used to find an initial blocking factor. The initial blocking factor is found by simply using the above discussed loop unrollings, and seeing which of the blocking factors appropriate to the detected L1 cache size produce the best result.

With this initial blocking factor, which instructions set to use (muladd or separate multiply and add), and a guess as to pipeline length, the search routine loops over all M and N loop unrollings possible with the given number of registers.

Once an optimal unrolling has been found, ATLAS again tries all blocking factors, and various latency and K-loop unrolling factors, and chooses the best.

All results are stored in files, so that subsequent searches will not repeat the same experiments, allowing searches to build on previously obtained data. This also means that if a search is interrupted (for instance due to a machine failure), previously run cases will not need to be re-timed. A typical install takes from 1 to 2 hours for each precision.

6.2.8 Timing Results

Figure 4 shows the performance of double precision matmul across multiple architectures for a problem of size 500. This graph compares performance obtained by ATLAS, the Fortran77 reference BLAS, and on those platforms where they exist, the vendor-supplied BLAS. The problem size 500 is chosen as an intermediate problem size (i.e., it is not the problem size which ATLAS performs best on, for instance). These timings utilize ATLAS's cache-flushing mechanism, and so may be lower than those reported elsewhere. More complete timings can be found in [21, 22].

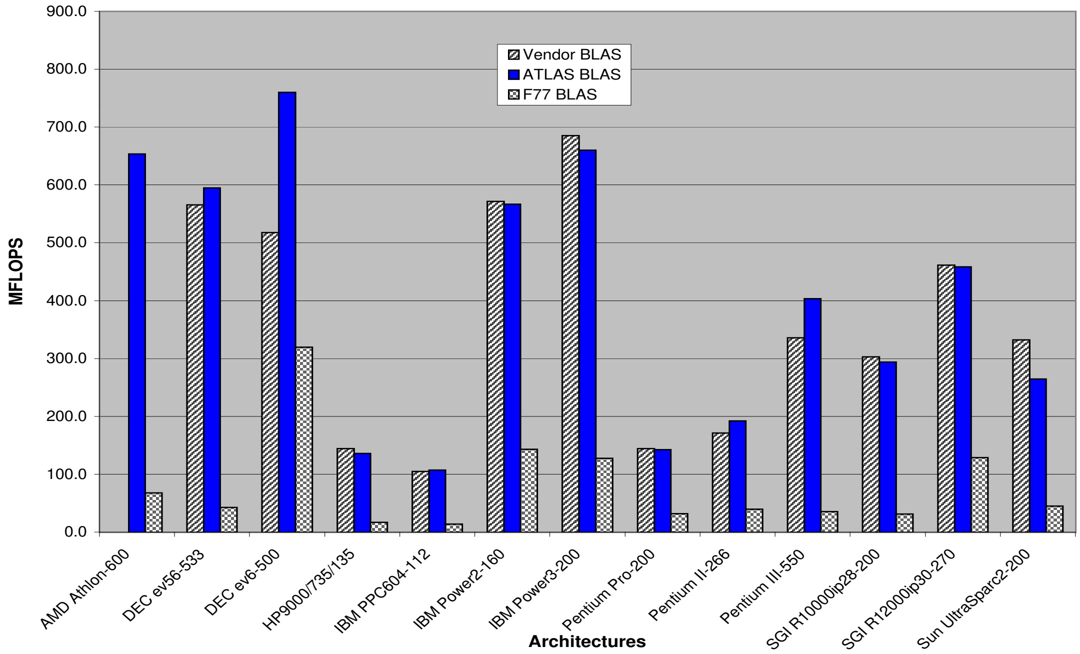

Figure 4: Performance of 500x500 DGEMM across multiple architectures

6.3 GEMM-based Level 3 BLAS

The Level 3 BLAS specify six (respectively nine) routines for the real (respectively complex) data types. In addition to the general matrix-matrix multiplication (GEMM) described above, the Level 3 BLAS API [5] specifies routines performing triangular matrix-matrix multiply (TRMM), triangular system solve (TRSM), symmetric or Hermitian matrix-matrix multiply (SYMM, HEMM), and symmetric or Hermitian rank-k and rank-2k updates (SYRK, SYR2K, HERK and HER2K).

From a mathematical point of view, it is clear that all of these operations can be expressed in terms of general matrix-matrix multiplies (GEMM) and floating-point division. Such a design is highly attractive due to the obvious potential for code reuse. It turns out that such formulations of these remaining Level 3 BLAS operations can be made highly efficient, assuming the implementation of the GEMM routine is. Such Level 3 BLAS designs are traditionally referred to as *GEMM-based.*

The basic idea is to partition the computations across submatrices so that the calculations can be expressed in terms of explicit calls to GEMM and the appropriate Level 3 BLAS primitives. This idea can be illustrated using the triangular matrix-matrix multiply operation $B \leftarrow A \times B$, where A is an M-by-M upper triangular matrix, and B is a general M-by-N matrix.

$$\left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline 0 & A_{22} \end{array}\right) \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) \tag{1}$$

Equation 1 illustrates a simple partitioning scheme, where the triangular matrix A has been partitioned once in both dimensions, and the right-hand side matrix B has been accordingly decomposed in the row dimension only. The overall computation can then be expressed as follows:

1. $B_1 \leftarrow A_{11}B_1$ (TRMM)
2. $B_1 \leftarrow B_1 + A_{12}B_2$ (GEMM)
3. $B_2 \leftarrow A_{22}B_2$ (TRMM)

This example shows two main features of GEMM-based Level 3 BLAS: first, explicit calls to the Level 3 BLAS GEMM routine are made, and second, such a design is naturally recursive. GEMM-based Level 3 BLAS are further classified according to their partitioning policy. There are many possible partitioning algorithms, and a great deal of past and continuing research has been done on this problem. For instance, partitioning schemes may utilize fixed and machine-specific blocking as in [15, 4], or more generalized recursive schemes such as presented in [12, 13].

ATLAS implements a relative simple recursive GEMM-based BLAS design. The row and column dimensions of the triangular, symmetric or Hermitian matrix and only the appropriate dimension of the general matrix operands are halved at each step. Recursion stops when the order of the square block diagonal is less than or equal to GEMM's Level 1 cache blocking factor, N_B. The N_B or less sized Level 3 BLAS primitives (TRMM in the above example) used at the leaves of the tree are implemented both as simple loops, and in terms of GEMM, and which one is used depends on the problem sizes and relative efficiency between GEMM and the simple loop implementation.

This design can be implemented both simply and elegantly in a very small amount of code in any language natively supporting recursion. The design's most important feature is that all performance optimizations, both memory and computational, are isolated in GEMM. Most other GEMM-based designs instead perform the memory optimizations to at least some degree in the GEMM-based routines, and rely on GEMM mainly for computational optimizations.

To understand this, recall that optimizing memory access involves blocking the matrices in order to encourage cache reuse. However, the partitioning scheme used by the Level 3 BLAS is itself a blocking, and if chosen unwisely, can prevent GEMM from doing cache blocking.

The drawbacks of this approach are obvious. As we have seen in previous sections, partitioning schemes can become quite complex; this complexity is naturally reflected in the implementation. Reproducing both this code complexity and the architectural-dependent caching information throughout the Level 3 BLAS robs the GEMM-based design of its greatest strength: its reliance on a centralized kernel for performance wins.

In ATLAS's GEMM-based approach, the only parameter that changes with the architecture is N_B, which is supplied automatically by GEMM. Further, the submatrices implied by the recursion are square, which tends to allow greater cache blocking opportunities to GEMM than non-square shapes.

It is clear that the most optimal implementation would not be GEMM-based, but would instead have specialized cache and compute parameters just as in our previously described GEMM implementation. However, we believe that the performance loss inherent in using ATLAS's GEMM-based approach is in practice negligible, and thus the simplicity and platform-independence of the GEMM-based approach used by ATLAS constitutes a clear win. The timings presented in the following section appear to substantiate this idea.

6.3.1 Timing Results

Again, space considerations rule out presenting extensive timing results. We have chosen to show results for all double precision BLAS operations on two architectures, again with problems of size 500, and comparing results for vendor, ATLAS, and Fortran77 reference implementations.

Figure 5 shows the performance results for the Sun UltraSparc (200Mhz). This architecture is interesting because it is the one on which ATLAS's GEMM (the compute kernel for all of ATLAS's Level 3 BLAS) performs worst in regards to the vendor-supplied version. This graph shows that, even when ATLAS's GEMM is not as good as the vendor version, the rest of the Level 3 BLAS, which are built in terms of it, may nonetheless still compare favorably with the vendor BLAS. This highlights one of the disadvantages of codes that do not use the kernel approach to library building: uneven optimization, based on the priorities of the library producer (which may well not match the needs of the end user).

Figure 6 shows the same data for a 533Mhz DEC ev56. This second architecture is more typical in that ATLAS and vendor GEMM are much closer to parity. Here we see that when ATLAS's GEMM compares favorably with the vendor implementation, this advantage carries over for the entire BLAS. As with the UltraSparc results, we again see that the vendor optimization effort has varied widely between routines, while ATLAS maintains a more even level of optimality.

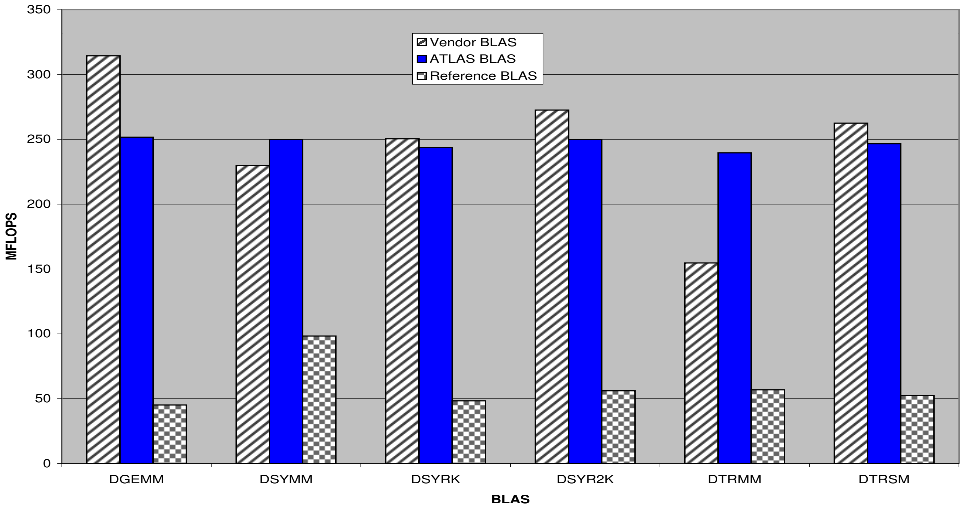

Figure 5: Performance of double precision BLAS on Sun UltraSparc 2200

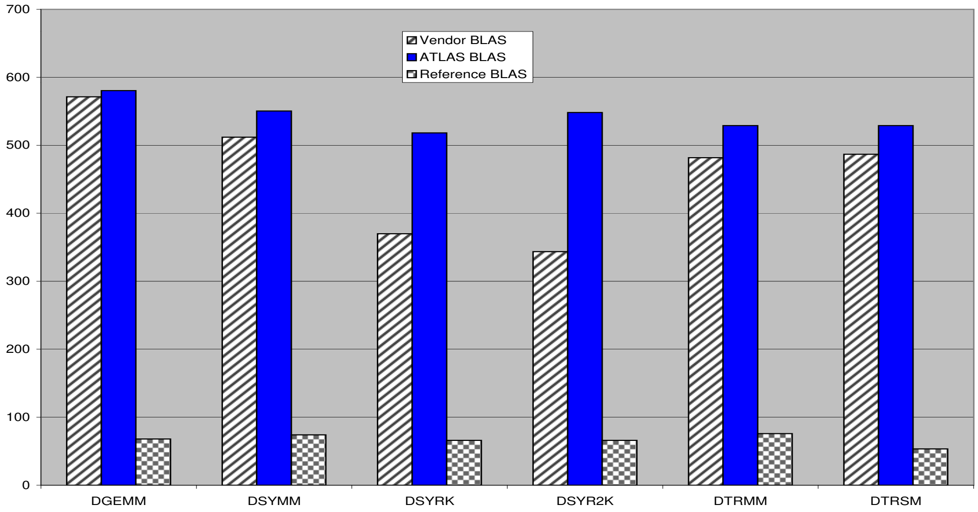

Figure 6: Performance of double precision BLAS on DEC ev56

7 Optimizing the Level 2 BLAS

The Level 2 BLAS perform matrix-vector operations of various sorts. All routines have at most one matrix operand, and one or two vector operands. Unfortunately, space considerations rule out covering ATLAS's Level 2 BLAS implementation in any real detail. Therefore, this section will explain the theoretical underpinnings of all Level 2 optimizations: the basic memory optimization techniques that allow the vector operand(s) main memory access to be reduced from $O(N^2)$ to $O(N)$. We then describe, in the broadest possible strokes, how these and other optimizations are used by ATLAS.

7.1 Register and Cache Blocking for the Level 2 BLAS

If no register or cache blocking were done, the Level 2 operations would require $O(N^2)$ data access on each operand. With the appropriate register and cache blocking, the vector operands' access can be reduced to $O(N)$. Obviously enough, the $O(N^2)$ matrix access cannot be reduced, since the matrix is actually of size $O(N^2)$.

To understand this in detail, we look at the matrix vector multiply operation. In the BLAS, the matrix-vector multiply routine performs $y \leftarrow \alpha op(A)x + \beta y$, where $op(A) = A$, A^H or A^T and A has M rows and N columns. For our discussion, it is enough to examine the case $y \leftarrow Ax + y$, where A is a square matrix of size N.

This operation may be summarized as $\sum_{i=1}^{N}(y_i = \sum_{j=1}^{N} A_{ij}x_j + y_i)$; From this equation it is clear that calculating an element of y requires reading the entire N length vector x, reading and writing the ith element of y N times, and reading the entire N length row i of the matrix A. Since there are N elements of y, it follows that this algorithm requires N^2 reads of A, N^2 reads of x, N^2 reads and N^2 writes of y. Just as with the Level 3 operations, the number of references cannot be changed without changing the definition of the operation, but by using appropriate cache and register blockings, the number of the references that must be satisfied out of main memory or higher levels of cache can be drastically reduced.

The minimum number of main memory references required to do this operation results in accessing each element from main memory only once, which reduces the accesses from ($3N^2$ reads + N^2 writes) to ($N^2 + N$ reads + N writes).

As an interesting aside, even this trivial analysis is sufficient to understand the large performance advantage enjoyed by the Level 3 over the Level 2 BLAS routines. All Level 2 BLAS require $O(N^2)$ FLOPs (Floating Point Operations); a completely optimal implementation can at best reduce the number of main memory accesses to the same order, $O(N^2)$. The Level 3 BLAS, in contrast, require $O(N^3)$ FLOPs, but the number of main memory accesses can be reduced to a lower order term, $O(N^2)$. Since most modern machines have relatively slow memory when compared to their peak FLOP rate, this analysis dictates that Level 3 BLAS will achieve a much higher percentage of the peak FLOP rate than the Level 2 BLAS.

Getting back to Level 2 BLAS, we now examine the register and cache blocking, which are used in order to reduce the vector accesses.

7.1.1 Register Blocking

Registers are scalars which are directly accessed by the floating point unit. In a way, registers thus correspond to a "Level 0" cache, which operates at infinite speed. Given an infinite number of registers, only one main memory access per element would be required for all operations. Unfortunately, the number of user-addressable floating point registers available on modern architectures typically varies between 8 and 32, and thus all but the most trivial operations will overflow the registers.

For this reason, register blocking alone can reduce either the y or x access term from $O(N^2)$, to $O(N)$, but not both. This is easily seen using the simplified GEMV operation introduced in the previous section. The basic algorithm required to reduce the accesses of y to $O(N)$ is most easily shown in the following pseudo-code:

```
do I = 1, N
   r = y(I)
   do J = 1, N
      r += A(I,J) * x(J)
   end do
end do
```

This is an "inner product" or dot product-based matrix vector multiply. If we unroll the I loop and use R_y registers to hold the elements of y, we can reduce the N^2 accesses of x to $\frac{N^2}{R_y}$, by using a register to reuse the element `x(J)` R_y times for each load.

Unrolling the loop like this essentially creates a hybrid algorithm, in the sense that the R_y y access constitute a small outer product. However, since registers cannot hold both y and x throughout the algorithm, one or the other must be flushed as the loop progresses (thus necessitating multiple loads to registers), and since we drop the value of x and maintain y in the registers, this "hybrid" algorithm is still essentially inner product.

Reducing the x component to $O(N)$ accesses requires the "outer product" or AXPY-based (AXPY being a Level 1 BLAS routine performing the operation $y \leftarrow \alpha x + y$) version of GEMV:

```
do J = 1, N
   r = x(J)
   do I = 1, N
      y(I) += A(I,J) * r
   end do
end do
```

This gives us N read accesses on x, and, just as with the inner product, unrolling the J loop and using R_x registers to hold the elements of x, we can reduce the accesses of y to $\frac{N^2}{R_x}$ reads **and** writes, by using an additional register to reuse `y(I)` R_x times.

Therefore, strictly for register blocking purposes, the inner product formulation is superior to the outer product: the total number of reads of both formulations is $O(N^2) + O(N)$, but the number of writes is $O(N)$ for inner product, but $O(N^2)$ for outer product. In practice, when array columns are stored contiguously, a heavily unrolled AXPY-based algorithm may in fact be used, since it better utilizes hardware prefetch, cache line fetch, TLB access, etc. As mentioned before, however, such details are beyond the scope of this paper, so we will assume the register blocking used will be the inner product formulation.

As another practical note, the number of registers available for doing multiple AXPYs or dot products is severely limited, even beyond the 8 or 32 ISA (instruction set architecture) limit. In the inner product formulation, where R_y registers are used to form the R_y simultaneous dot products, at least two registers must be available for loading elements of x and A. Further registers will be used in order to support pipelining and prefetch. Large unrollings also mean accessing many more memory locations simultaneously, which can swamp the memory fetch capabilities of the architecture. This means that R_y is usually kept to a relatively small number (typically in the range of $2-8$).

In summation, register blocking reduces one vector access to $O(N)$ cost; the vector usually chosen for this reduction is the output vector (i.e., an inner product type register block), due to its higher cost. In order to reduce the remaining vector to $O(N)$, we must apply cache blocking.

While it is tempting to regard register blocking as a special case of cache blocking, their implementations are fundamentally different. As we will see, cache blocking can be easily done by simply parameterizing the relevant code, so that properly blocked sections of the operands are accessed. Register blocking, as this section has demonstrated, relies on source adaptation, since varying it requires changing the loop order, number of registers, loop unrollings, etc., all of which change the code in ways that cannot be supported via simple parameterization.

7.1.2 Cache Blocking

As previously discussed, register blocking has reduced the access of y to $O(N)$, leaving the x access at $O(N^2)$. Therefore, loading x to registers $O(N^2)$ times cannot be avoided; However, the optimal algorithm will guarantee that main memory satisfies only $O(N)$ of these requests, leaving lower levels of cache to satisfy the rest.

Again, GEMV can be used to better understand this idea. The register block is doing R_y simultaneous dot products, so that the y access is N reads and N writes, while the x fetch to registers is $\frac{N^2}{R_y}$. Since x is reused in forming each successive dot product, x is a candidate for cache reuse. It is easily seen that forming R_y dot products accesses R_y elements of y, all N elements of x, and $R_y \times N$ elements of A. Thus the footprint in cache of one step of this algorithm is roughly $R_y + N + R_y N$.

Therefore, we can effectively guarantee L1 cache reuse by partitioning the original problem so that the footprint in cache is small enough that the relevant portion of x is not flushed between successive sets of dot products. Therefore, the correct blocking for x may be determined by solving an equation, whose simplified expression would be: $R_y + N_p + R_y N_p = S_1 \Rightarrow N_p = \frac{S_1 - R_y}{R_y + 1}$, where S_1 is the size, in elements, of the Level 1 cache, and N_p is the partitioning of x that we are solving for.

In practice, this equation is more complicated: some memory unrelated to the algorithm will always be in cache, there will be problems associated with cache line conflicts, etc. In addition, the equation needs to be adapted to the underlying register blocking so that the initial load of the next step does not unnecessarily flush x. However, these details, while important in extracting the maximal performance, are not required for conceptual understanding, and so are omitted here.

With the correct partitioning (N_p) known, the original $N \times N$ GEMV is then blocked into $\lceil N/N_p \rceil$ separate problems of size $N \times N_p$ (the last such problem will obviously be smaller if N_p does not divide N evenly). The data access to main memory is then $\lceil N/N_p \rceil N$ reads and writes of y, N reads of X, and N^2 reads of A.

N_p is typically very close to N in size, and so this algorithm is very near optimal in its memory access. N_p will typically be in the range 350 - 1500, so even very large problems still have extremely small coefficients on the y access term. Note that any problem with $N \leq N_p$ will achieve the optimal result (N^2 access of A, N access of x and y) without any need for any cache blocking (register blocking is still required).

There is little point in explicitly blocking for higher levels of cache in the Level 2 BLAS. However, if the machine possesses a level of cache large enough to hold the footprint of the entire L1-blocked algorithm (with the previously stated simplifications, this is roughly $N_pN + N_p + R_y$), y will be reused without need for explicit blocking, and the main memory access will be reduced to its theoretical minimum.

7.2 ATLAS's Level 2 Compute Kernels

As we have seen, ATLAS employs one low-level compute kernel (the L1 matmul), from which the BLAS's more general GEMM routine is built. The L1 matmul and GEMM are then used in turn to generate the rest of the Level 3 BLAS. With this method, only this one relatively simple kernel needs to be supported using code adaptation, and its performance dictates that of the entire Level 3 BLAS.

The same strategy is employed for the Level 2 BLAS, but two types of compute kernels are needed rather than one. Just as with the L1 matmul, these kernels perform register blocking and various floating point optimizations, but do no cache blocking, as it is assumed that the dimensions of the arguments have been blocked by higher level codes in order to ensure L1 cache reuse. The compute kernels for the Level 2 BLAS are:

- *L1 matvec*: An L1-contained matrix vector multiply, with four variants:
    1. No Transpose – matrix A's rows are stored in rows of input array
    2. Conjugate (complex only) – matrix A's rows are stored in conjugated form in rows of input array
    3. Transpose – matrix A's rows are stored in columns of input array
    4. Conjugate Transpose (complex only) – matrix A's rows are stored in conjugated form in columns of input array
- *L1 update1*: An L1-contained rank-1 update

Both of these kernels further supply three specialized β cases (0, 1, and variable).

7.3 Building ATLAS's Level 2 BLAS

This section presents a very rough outline of how ATLAS supports the Level 2 BLAS. The install of the Level 3 BLAS precedes that of the Level 2, and from this process ATLAS knows the size of the L1 cache. Thus, using a slightly more complicated version of the equations given in Section 7.1.2, ATLAS has a good idea of the correct Level 1 cache partitioning to use. With this in hand, ATLAS is ready to find the best compute kernels for the Level 2 BLAS.

Presently, ATLAS relies solely on multiple implementation to support these kernels (e.g. code generation is not employed). Therefore, the search simply tries each implementation in turn, and chooses the best. The conjugate forms of the L1 matvec have the same performance characteristics as their nonconjugate equivalents, so ATLAS need search only 3 differing kernels: notranspose matvec, transpose matvec, and L1 update1.

Using these best algorithms, ATLAS empirically discovers the optimum percentage of the L1 cache to use. These empirically-discovered blockings and kernel implementations are then used to build the Level 2 BLAS routines GEMV and GER (much as GEMM was built using the L1 matmul), and all of this information and these building blocks are then used to produce the rest of the Level 2 BLAS.

Part IV
Conclusion and future work

Results presented and referenced here demonstrate unambiguously that AEOS techniques can be utilized to build portable performance-critical libraries, which compete favorably with machine-specific, hand-tuned codes. We believe that the AEOS paradigm will ultimately have a major impact on high performance library development and maintenance.

ATLAS has produced a complete BLAS, and the ATLAS BLAS are already widely used in the linear algebra community. Further information, including the software and documentation, is available at the ATLAS homepage, `www.netlib.org/atlas`.

This paper has given a very high level overview of the methods used in the ATLAS project to support the BLAS, and the ATLAS project is continuing to improve and extend on this work. There are many more areas of ATLAS/AEOS research than can be investigated by any one group. Some of the areas we are currently considering are:

- *Generalizations of architecture information.* We are examining to what degree the information ATLAS discovers during the install process can be generalized and made available to other packages.

- *Code Generation for GEMV and GER kernels.* The present dense Level 2 BLAS may not be optimal for all platforms because their compute engine (L1 matmul and L1 update1) are supported solely by multiple implementation. For maximal performance, it will be necessary to supplement this with code generation, as we have done in the Level 3 BLAS.

- *Code generation for some Level 1 BLAS routines.* Many Level 1 BLAS routines cannot be optimized much more than a standard compiler will do, and so do not need special attention via ATLAS's empirical techniques. However, operations such as AXPY ($y \leftarrow y + \alpha x$) are complex enough that the potential performance benefit makes it worth investigating the optimizations provided by code generation. Also, this investigation is necessary in order to support sparse operations, which use Level 1, or near-Level 1, operations relatively often.

- *SMP support via pthreads.* Providing shared memory processing support for the BLAS via pthreads is not difficult. However, making such support portable across differing pthreads implementations is more challenging, and finding reliable timing methods for threaded codes so that they may be adapted via AEOS techniques has proven quite difficult indeed. We are investigating these areas now.

- *Packed storage optimizations.* One important area that has been traditionally mishandled is packed storage, where only the relevant portion of a triangular or symmetric matrix is stored, allowing for larger problems to be solved in the same memory space. Present implementations are orders of magnitude slower than they need to be due to BLAS interface issues, and vector-based algorithms. This work may require extending the present generators, or development of specialized routines.

- *Sparse optimizations.* This is an open-ended research area that encompasses many different areas of optimization. We hope to use our experience with dense optimizations in order to gain insight into the more tractable storage schemes. This will later pave the way for more advanced work, such as structure analysis and dynamic libraries, as well as providing a springboard to handling the less dense-like structures.

- *Algorithmic research and higher level routines.* We have already extended ATLAS beyond the BLAS and into higher level kernels such as LAPACK's LU and Cholesky. This trend should continue, with perhaps some interesting algorithmic research. For instance, with the known performance provided by ATLAS, alternative algorithms may become attractive in the search for the best performance (an example might be use of the sign function for eigenvalues, due to the relative performance advantage its Level 3 BLAS operations enjoy over the Level 2 operations used by traditional methods) [2].

References

[1] 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, Second Edition.* SIAM, Philadelphia, PA, 1995.

[2] Z. Bai, J. Demmel, J. Dongarra, A. Petitet, H. Robinson, and K. Stanley. The Spectral Decomposition of Nonsymmetric Matrices on Distributed-Memory Computers. *SIAM Journal on Scientific Computing*, 18(5):1446–1461, 1997.

[3] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C. Chin. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. Technical Report UT CS-96-326, LAPACK Working Note No.111, University of Tennessee, 1996.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] J. Dongarra, P. Mayes, and G. Radicati di Brozolo. The IBM RISC System 6000 and linear algebra operations. *Supercomputer*, 8(4):15–30, 1991.

[9] M. Frigo. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the ICASSP Conference*, volume 3, page 1381, 1998.

[10] M. Frigo. A Fast Fourier Transform Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99), Atlanta, GA*, 1999.

[11] M. Frigo and S. G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, 1993.

[12] 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.

[13] 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.

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

[15] 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.

[16] K. Kennedy. Personal Communication.

[17] 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.

[18] Jakob Ostergaard. OptimQR - A Software-package to create near-optimal solvers for sparse systems of linear equations. `http://ostenfeld.dk/ jakob/OptimQR`.

[19] See homepage for a complete list of the people involved. Signal Processing algorithms Implementation Research for Adaptable Libraries. `http://www.ece.cmu.edu/ spiral`.

[20] See homepage for a complete list of the people involved. TUNE - Mathematical Models, Transformations and System Support for Memory-Friendly Programming. `http://www.cs.unc.edu/Research/TUNE`.

[21] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. http://www.cs.utk.edu/ rwhaley/ATL/INDEX.HTM, 1998. Winner, best paper in the systems category, SC98: High Performance Networking and Computing.

[22] 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.

[23] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London, 1989.

[24] M. Wolfe. *High Performance Compilers for Parallel Computers.* Addison-Wesley, 1996.