Automatically Tuned Linear Algebra Software

R. Clint Whaley
Jack J. Dongarra

Computer Science Department
University of Tennessee
Knoxville, TN 37996-1301

and

Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37831

1 Abstract

This paper describes an approach for the automatic generation and optimization of numerical software for processors with deep memory hierarchies and pipelined functional units. The production of such software for machines ranging from desktop workstations to embedded processors can be a tedious and time consuming process. The work described here can help in automating much of this process. We will concentrate our efforts on the widely used linear algebra kernels called the Basic Linear Algebra Subroutines (BLAS). In particular, the work presented here is for general matrix multiply, DGEMM. However much of the technology and approach developed here can be applied to the other Level 3 BLAS and the general strategy can have an impact on basic linear algebra operations in general and may be extended to other important kernel operations.

Contents

List of Tables

List of Figures

2 Motivation

Today's microprocessors have peak execution rates ranging from 300 Mflop/s to 1.2 Gflop/s. However, straightforward implementation in Fortran or C of computations based on simple loops rarely results in such high performance. To realize such peak rates of execution for even the simplest of operations has required tedious, hand coded, programming efforts.

Since their inception, the use of defacto standards like the BLAS [5, 4] has been a means of achieving portability and efficiency for a wide range of kernel scientific computations. While these BLAS are used heavily in linear algebra computations, such as solving dense systems of equations, they have also found their way into the basic computing infrastructure of many applications. The BLAS (Basic Linear Algebra Subprograms) are high quality "building block" routines for performing basic vector and matrix operations. Level 1 BLAS do vector-vector operations, Level 2 BLAS do matrix-vector operations, and Level 3 BLAS do matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software, such as LAPACK [1] and ScaLAPACK [2], for example.

The BLAS themselves are just a standard or specification of the semantics and syntax for the operations. There is a set of reference implementations written in Fortran, but no attempt was made with these reference implementations to promote efficiency. Many vendors provide a "optimized" implementation of the BLAS for a specific machine architecture. These optimized BLAS libraries are provided by the computer vendor or by an independent software vendor (ISV).

In general, the existing BLAS have proven to be very effective in assisting portable, efficient software for sequential, vector and shared memory high-performance computers. However, hand-optimized BLAS are expensive and tedious to produce for any particular architecture, and in general will only be created when there is a large enough market, which is not the true for all platforms. The process of generating an optimized set of BLAS for a new architecture or a slightly different machine version can be a time consuming process. The programmer must understand the architecture, how the memory hierarchy can be used to provide data in an optimum fashion, how the functional units and registers can be manipulated to generate the correct operands at the correct time, and how best to use the compiler optimization. Care must be taken to optimize the operations to account for many parameters such as blocking factors, loop unrolling depths, software pipelining strategies, loop ordering, register allocations, and instruction scheduling.

Many computer vendors have invested considerable resources in producing optimized BLAS for their architectures. In many cases near optimum performance can be achieved for some operations. However the coverage and the level of performance achieved has not been uniform across all platforms. An example is that up until this point we have not had an efficient version of matrix multiply for the Pentium/Linux architecture.

Our goal is to develop a methodology for the automatic generation of highly efficient basic linear algebra routines for today's microprocessors. The process will work on processors that have an on-chip cache and a reasonable C compiler. Our approach, called Automatically Tuned Linear Algebra Software (ATLAS), has been able to match or exceed the performance of the vendor supplied version of matrix multiply in almost every case.

More complete timings will be given in section 4, where we report on the timings of various problem sizes across multiple architectures. As a preview of this more complete coverage, figure 1 shows the performance of ATLAS versus the vendor-supplied DGEMM (where available) for a 500x500 matrix multiply. See section 4 for further details on these results.

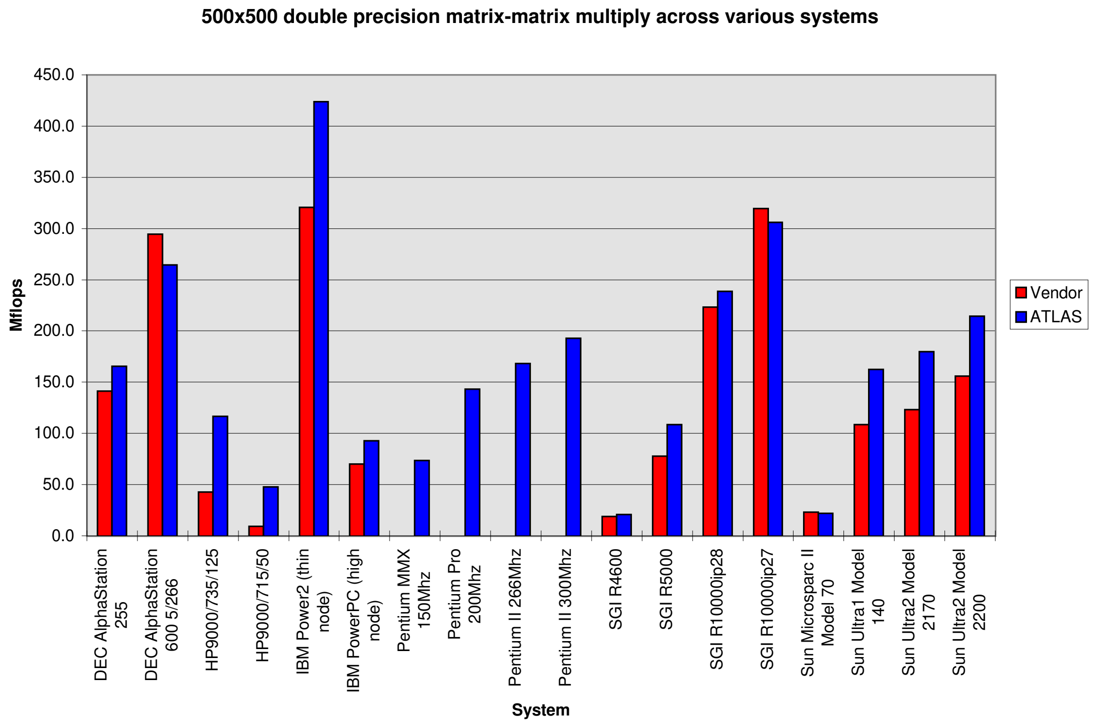

Figure 1: ATLAS/vendor performance preview

3 ATLAS

We have developed a general methodology for the generation of the Level 3 BLAS and describe here how this approach is carried out and some of the preliminary results we have achieved. At the moment, the operation we are supporting matrix multiply. We can describe matrix multiply as $C \leftarrow \alpha op(A) op(B) + \beta C$, where $op(X) = X$ or X^T. C is an $M \times N$ matrix, and A and 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 [6].

3.1 The ATLAS approach

In our approach, we have isolated the machine-specific features of the operation to several routines, all of which deal with performing an optimized on-chip (i.e., in L1 cache) matrix multiply. This section of code is automatically created by a code generator which uses timings to determine the correct blocking and loop unrolling factors to perform an optimized on-chip multiply. The user may directly supply the code generator with as much detail as desired (i.e., the user may explicitly indicate the L1 cache size, the blocking factor(s) to try, etc); if such details are not provided, the generator will determine appropriate settings via timings.

The rest of the code does not change across architectures, and handles the looping, blocking, and so on necessary to build the complete matrix-matrix multiply from the on-chip multiply.

3.2 Building the general matrix multiply from the on-chip multiply

In this section we describe the code which remains the same across all platforms: the routines necessary to build a general matrix-matrix multiply using a fixed-size on-chip multiply.

The following section describes the on-chip multiply and its code generator in detail. For this section, it is enough to know that we have an efficient on-chip matrix matrix multiply of the form $C \leftarrow A^T B + C$. This multiply is of fixed size, with all dimensions set to a system-specific value, N_B (i.e, $M = N = K = N_B$). Also available are several "cleanup" codes, which handle the cases caused by dimensions which are not multiples of the blocking factor.

When the user calls our _GEMM, the first decision is whether the problem is large enough to benefit from our special techniques. Our algorithm requires copying of the operand matrices; if the problem is small enough, this $O(N^2)$ cost, along with miscellaneous overheads such as function calls and multiple layers of looping, can actually make the "optimized" GEMM slower than the traditional 3 do loops. The size required for the $O(N^3)$ costs to dominate these lower order terms varies across machines, and so this switch point is automatically determined at installation time.

For these very small problems, a standard 3-loop multiply with some simple loop unrolling is called. This code will also be called if the algorithm is unable to allocate enough space to do the blocking (see below for further details).

Assuming the matrix is large enough, there are presently two algorithms for performing the general, off-chip 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 then try to allocate enough space to store a $N_B \times N_B$ output temporary, $\hat{C}$, 1 panel of the outermost matrix, and the entire inner matrix. If this fails, the algorithms attempt to allocate enough space to hold $\hat{C}$, and 1 panel from both A and B. The minimum workspace required by these routines is therefore ${N_B}^2 + (M + K)N_B$. If this amount of workspace cannot be allocated, the previously mentioned small case code is called instead.

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

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

If enough space for a copy of the entire innermost matrix cannot be allocated, the innermost matrix will be entirely copied for each panel of the outermost matrix (i.e., if A is our outermost matrix, we 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 2.

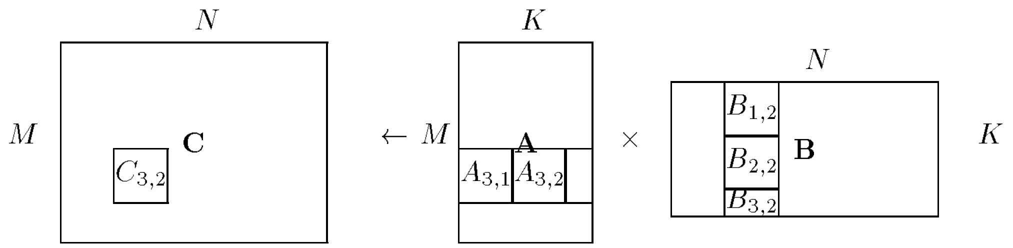

Figure 2: One step of matrix-matrix multiply

In this operation, 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 $0 \leq i < \lceil M/N_B \rceil$, $0 \leq j < \lceil N/N_B \rceil$:

1. Zero out $N_B \times N_B$ section of workspace to store AB; call this workspace $\hat{C}_{i,j}$.

```
work = allocate(M*K + NB*(NB+K))
if (allocated(work)) then
   PARTIAL_MATRIX = .FALSE.
   copy A into block major format
else
   PARTIAL_MATRIX = .TRUE.
   work = allocate(NB*(NB+2*K))
   if (.NOT.allocated(work)) call small_case_code
   return
end if
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
      Cwork(1:NB,1:NB) = 0
      do k = 1, K, NB
         ON_CHIP_MATMUL(Awork, Bwork, Cwork)
      end do
      C(i:i+NB-1, j:j+NB-1) = BETA*Cwork
   end do
end do
```

Figure 3: General matrix multiplication with A as innermost matrix

2. Call on-chip multiply to multiply block k of the row panel i of A with block k of the column panel j of B, $\forall k, 0 \leq k < \lceil K/N_B \rceil$. The on-chip multiply 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.

3. Perform block operation $C_{i,j} \leftarrow \hat{C}_{i,j} + \beta C_{i,j}$.

Building on this inner loop, we have the loop orderings giving us our two algorithms for off-chip matrix multiplication. Figures 3, 4 give the pseudo-code for these two algorithms. We simplify this code by not showing the cleanup code necessary for cases where dimensions do not evenly divide N_B. The matrix copies are shown as if coming from the notranpose, notranpose case. If they do not, only the array access on the copy changes.

3.2.1 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:

- If either matrix will fit completely into L2 cache, put it as the innermost matrix (we get L2 cache reuse on entire inner matrix)

- If neither matrix fits completely into L2 cache, put the one with the largest panel that

```
work = allocate(N*K + NB*(NB+K))
if (allocated(work)) then
   PARTIAL_MATRIX = .FALSE.
   copy B into block major format
else
   PARTIAL_MATRIX = .TRUE.
   work = allocate(NB*(NB+2*K))
   if (.NOT.allocated(work)) call small_case_code
   return
end if
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
      Cwork(1:NB,1:NB) = 0
      do k = 1, K, NB
         ON_CHIP_MATMUL(Awork, Bwork, Cwork)
      end do
      C(i:i+NB-1, j:j+NB-1) = BETA*Cwork
   end do
end do
```

Figure 4: General matrix multiplication with B as innermost matrix

will fit into L2 cache as the outermost matrix (we get L2 cache reuse on the panel of the outer matrix)

The present code does no explicit L2 blocking, however (for instance, the size of the L2 cache is not known anywhere in the code) , and so these criteria are not presently used for this selection. Rather, if one matrix must be accessed by row-panels during the copy (for instance, the matrix A when `TRANSA='N'`), that matrix will be put where it can be copied most efficiently.

This means that if we have enough workspace to copy it up front, it will be accessed column-wise by putting it as the innermost loop and copying the entire matrix; otherwise it will be placed as the outermost loop, where the cost of copying the row-panel is a lower order term. If both matrices have the same access patterns, B will be made the outermost matrix, so that C is accessed by columns.

3.2.2 L2 Cache blocking

As previously mentioned, the present algorithms perform no explicit L2 cache blocking, but we achieve substantial implicit caching in certain cases. If the innermost matrix fits completely into L2 cache, the present algorithm will get maximal cache reuse. If there is room in the cache only for one panel from each matrix (and for the copy of the innermost panel, if we don't have the space to copy all of the innermost matrix up front), we will again get extremely good cache reuse, by reusing the outermost matrix panel against all panels of the innermost matrix. There are two other obvious opportunities for cache reuse, which the code presently does not take advantage of.

The first is the case where the L2 cache is not big enough to hold an entire panel of both A and B. In this case, the K loop must be intermixed with the outer loops in order to achieve L2 cache reuse. In this algorithm, X blocks of one of the panels is retained in the L2 cache, and the appropriate sections of the other input matrix are brought in to perform the multiplication. This requires either X $N_B \times N_B$ panels of $\hat{C}$ to be retained in L2 cache, or multiple writes to C. Either option tends to degrade performance. We have run some experiments to test to the feasibility of this idea, and have yet to see any performance gain.

More interesting is the case where multiple matrix panels will fit into L2 cache, but the entire innermost matrix will not. In this case the looping mechanism must become slightly more complex. This algorithm copies X panels of the outermost matrix into L2 cache, and then reuses them by applying them to the panels of the innermost matrix. If X becomes as small as 1 or as large as the number of panels in the outermost matrix, we are in a previously discussed case. Within this restriction, cache reuse grows with X.

This idea was promising enough that it was implemented. Of the 14 platforms surveyed in the timing section, only the DEC AlphaStation 600 showed appreciable performance gains; this platform showed maximal speedup of 1.08; obviously many problem sizes showed no speedup at all . No other platform gained speedup greater than 1.02. It was decided that this modest gain did not at this point justify the added code complexity of supporting explicit L2 blocking.

There are other reasons to consider explicit L2 references (for instance, deciding which matrix should be innermost), and if for one of these reasons we support the explicit calculation of L2 size, this algorithm should be easy to add.

3.3 Generation of the on-chip multiply

As previously mentioned, the on-chip matrix-matrix multiply is the only code which must change depending on the platform. Since we copy the input matrices into blocked form, only one case is required, which we have chosen as $C \leftarrow A^T B + C$. This case was chosen (as opposed to, for instance $C \leftarrow AB + C$), because it generates the largest (flops)/(cache misses) ratio possible when the loops are written with no unrolling. Machines with hardware allowing a smaller ratio can be addressed using loop unrolling on the M and N loops (this could also be addressed by permuting the order of the K loop, but we do not at present use this technique).

In a multiply designed for L1 cache reuse, one brings one of the input matrices completely into the L1 cache, and reuses that matrix 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 the L2 cache. Ignoring this 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. For the on-chip matrix multiplication, other relevant factors are:

- Instruction cache overflow
- Floating point instruction ordering
- loop overhead
- Exposure of possible parallelism
- The number of outstanding cache misses the hardware can handle before execution is blocked

3.3.1 Instruction cache reuse

Instructions are cached, and it is therefore important to fit our on-chip multiply's instructions into the L1 cache. This means that we will not be able to completely unroll all three loops, for instance.

3.3.2 Floating point instruction ordering

When we discuss floating point instruction ordering in this paper, it will usually be in reference to *latency hiding*.

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 3 or 5). Remember that our on-chip matrix multiply 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 a unnecessary execution stall. The operation `register = A[Y] * B[Z]` is issued to the floating point unit, and the add cannot started until the result of this computation is available, X cycles later, and thus 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. This reordering of operations can be done in hardware (out-of-order execution) or by the compiler, but this will sometimes generate code that is not quite 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.

3.3.3 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 the resulting memory access patterns.

3.3.4 Exposing parallelism

Many modern architectures have multiple floating point units. There are two barriers to achieving perfect parallel speedup with floating point 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.

3.3.5 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 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 over running the instruction cache, and the generation of such precision instruction sequence, for instance). So the method we use to control the cache-hit ratio is the more classical one of M and N loop unrolling.

3.3.6 Putting it all together

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. The user may enter the size of the L1 cache, or have the program attempt to calculate it. This in turn allows the routine to choose a range of blocking factors to examine. The user may specify the maximum number of registers to use (or use the default of 6), and thus dictate the maximum amount of M and/or N loop unrolling to perform.

Because it has not caused cache overflow anywhere, the present code always completely unrolls the K loop. This drastically reduces the number of cases the search routine must test.

The search then tries a number of possible blocking factors with a set amount of M and N loop unrolling (at the present, none), from which an initial blocking factor is chosen. Using this blocking factor, a range of latency hiding factors (presently, 1 through 6) is tested. If any of the latency factors produce speedup over the case with no latency hiding, the latency factor showing the maximal performance will be tested for every timing.

With an initial blocking factor and an idea of what latency factor to employ, 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, we again try all blocking factors, and all latency factors, and choose 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.

3.3.7 Cleanup code

After all of the above operations are done, we have a square on-chip multiply of fixed dimension N_B. Since the input matrices may not be a multiple of N_B, there is an obvious need for a way to handle the remainder.

It is possible to write the cleanup code in one routine, with 3 loops of arbitrary dimension. Practice shows that on some platforms, this results in unacceptably large performance drops for matrices with dimensions which are not multiples of N_B. Generating the code for all possible cleanup cases is not difficult, but is not a usable solution in practice. This would result in ${N_B}^3$ routines, which would take an unacceptable amount of compilation time, and make the user's executable too large.

The key is to note that a majority of the time spent in cleanup code will be the case where only 1 dimension is not equal to N_B. Therefore we generate roughly $4N_B$ routines for cleanup: $3N_B$ routines for the cases where a given dimension is less than N_B. The remaining routines accept arbitrary M and N, but K is known so that we can unroll the inner loop (critical for reducing loop overhead). Thus the N_B routines generated for the general case correspond to the differing values K is allowed. These routines where more than one dimension is less than N_B will still not be as efficient as the other routines, but the time spent in them should be negligible.

3.4 Why Can't the Compiler Do This?

It would be ideal if the compiler where capable of performing the optimization needed automatically. However, compiler technology is far from mature enough to perform these optimizations automatically. This is true even for the BLAS on widely marketed machines which can justify the great expense of compiler development. Adequate compilers for less widely marketed machines are almost certain not to be developed.

3.5 Requirements for Good Performance

The approach we have taken has two requirements necessary for achieving good performance:

1. There is a cache from which the floating point unit can fetch operands cheaply (i.e., a Level-1 cache)
2. The platform possesses an adequate C compiler.

If either of these requirements is not met, poor performance may result. There are three systems where we attempted an ATLAS installation, and had unacceptable performance. The platforms, and the reason for the performance loss is summarized below:

- *Cray T3E*: inadequate C compiler
- *Intel i860*: inadequate C compiler
- *SGI R8000*: No L1 cache accessible by floating point unit

The following sections describe how we drew these conclusions, and give some performance numbers so that the user can see the magnitude of the performance loss.

3.5.1 Effects of poor compilers

It might seem that the compiler would play little role in achieving performance, when the code generator does so much of the work usually reserved for compilers (eg., loop unrolling, latency hiding, etc.). The compiler must still be capable of doing a good job of register management and overall floating point unit control, however. Also, for the code other than the on-chip multiply, the compiler must do the brunt of optimization (this is a low-order cost, however).

We have several examples the role a compiler can play in determining performance. Perhaps the most direct evidence comes from our experiments on the SGI/CRAY T3E. The nodes of the T3E we had access to are DEC Alpha 21164 RISC processors, running at 450MHz. The DEC AlphaStation 600 (AS600) discussed in this report has the same chip, running at 266MHz. Cray, however, supplies their own compiler rather than using DEC's, and here we see a large difference in performance. Despite having the same chip (with the same L1 caches) running at roughly 1.7 times the clock rate, our timing numbers for the T3E are quite a bit slower than for the AS600. Table 1 shows the ATLAS timing numbers for the two platforms.

	Matrix Order									
PLATFORM	100	200	300	400	500	600	700	800	900	1000
AS600	170.8	252.1	264.7	280.2	262.7	256.4	259.8	258.1	257.0	257.3
T3E	171.6	183.1	186.6	189.7	189.3	192.6	193.1	194.4	194.1	193.9

Table 1: ATLAS performance comparison on AS600 and T3E

On the Intel i860, we were unable to get better than 12Mflop for the on-chip multiply itself, much less for the general case. We had access to only one compiler for this platform, so we cannot state for certain that the compiler is at fault. However, the i860 has an L1 cache, and has no obvious architectural peculiarities that would explain this poor performance. The system supplied matrix multiply exceeds 40Mflop, so we know there is plenty of room in the achievable peak. Further, the case that got the best performance was with no M or N loop unrolling, something that happens on no other platform. This anomalous result may be due to the compiler's inability to handle the increased register use inherit in outer loop unrolling.

One option when faced with a poor C compiler is to try another language. In the future we hope to provide the option to generate the on-chip multiply in F77. For some of the legacy platforms, this might offer a speed improvement over coding in C.

3.5.2 Performance loss from the lack of an on-chip cache

If a system does not possess an on-chip cache, the blocking we perform will not help performance. The copy into the block format becomes a pure overhead. The only machine we had access to where this is the case is the SGI R8000 processor, which possesses an on-chip cache which is not accessible by the floating point unit. The floating point unit has access only to the off-chip cache (level 2). Our performance on this architecture is extremely poor. Table 2 shows the performance of ATLAS versus the vendor supplied BLAS.

It should also be noted that the vendor-supplied BLAS usually achieve a much greater performance than this (performance in the upper 200's instead of lower), but apparently the leading dimension we used in our timings was poor, and performance was degraded. This merely shows that ATLAS cannot compete on this platform even with the system's poor cases. It should be noted, however, that the Fortran77 BLAS available on netlib run at roughly 3.6Mflop for all problem sizes for the same timing.

		Matrix Order									
SYSTEM	LIB	100	200	300	400	500	600	700	800	900	1000
R8000	SYS	243.9	289.6	207.0	210.4	213.5	213.5	214.1	213.5	213.6	214.5
R8000	ATL	111.8	138.1	146.1	145.9	148.4	149.1	147.3	145.9	146.4	145.4

Table 2: DGEMM performance in MFLOPS for the SGI R8000 processor

3.6 Other BLAS

At this point we consider how the general method outlined in this section can be extended to other BLAS.

With the exception of the triangular solve, all level 3 BLAS can naturally be expressed in terms of the previously mentioned on-chip matrix multiply. This means that no more system-specific code must be generated to support these routines, which implies our installation time should not increase when these additional BLAS are supported. To support these routines should require only the development of the off-chip codes. In the meantime, a gemm-based or "poor-man's BLAS" [7] may be utilized in order to generate a wider set of Level 3 BLAS.

The triangular solve can be written in terms of matrix multiply as well, and further research will be needed to see if this is a win compared with directly generating an on-chip solve. It seems likely that optimal performance would demand a mixture of these two approaches.

The level 1 and 2 BLAS require a different approach. In level 3, the luxury of $O(N^3)$ operations allows us to perform data copies, and thereby concentrate most optimization, and thus system-specific code, in a few routines. When the order of operations to be done is the same as your data, this is not feasible.

For these routines, it will likely be necessary to generate code for each operation. Thus, for instance, a separate code would be generated for the transpose cases of GEMV. This will lead to an explosion of routines to be generated and timed, implying extremely long installation times.

One promising idea is to create more general scheme which tries mainly to optimize the memory fetch, which would be generally usable within a level, and perform the complete generation/timing sequence for only a few select routines of special interest (eg. GEMV).

4 Results

In this section we present single and double precision timings across various platforms. These timings are different than many BLAS timings in that we flush cache before each call, and set the leading dimensions of the arrays to greater than the number of rows of the matrix (all timings in this section set the leading dimension to the maximal size timed, 1000). This means our performance numbers, even when timing the same routine (for instance the vendor-supplied DGEMM) are lower than those reported in other papers. However, these numbers are in general a much better estimate of the performance a user will see in his application. We devote a brief section to this topic.

Next, we show timings for square matrix multiply on all systems. To demonstrate that the performance shown in these timings translates to actual applications, we then give LU timings for various systems. On platforms that support it, we show that these routines respond well to threading.

Table 3 shows the configurations of the various platforms which we have installed and timed the package on.

Appendix 8 has several tables providing further details. Table 12 shows the system BLAS that were used for the timings. We should note that we did not have access to HP's most optimal BLAS, and so had to compare against their vector library (which describes itself as optimized for the 9000 series) instead. Tables 13 and 14 show the compiler version and flags used in compiling the on-chip matrix multiply.

Abbr. Name	Full Name	Clock (MHz)	L1 Data Cache(KB)	L1 Instr Cache (KB)	L2 Cache (KB/MB)
AS255	DEC AlphaStation 255	300	16	16	1MB
AS600	DEC AlphaStation 600 5/266	266	8	8	96KB & 4MB
HP9K/735	HP 9000/735/125	125	256	256	NONE
HP9K/715	HP 9000/715/50	50	64	Unknown	NONE
POWER2	IBM Power2 (thin node)	120	128	32	NONE
POWERPC	IBM PowerPC 604 (high node)	112	16	16	1MB
P5	Pentium	166	8	8	256KB
P5MMX	Pentium with MMX	150	16	16	256KB
PPRO	Pentium Pro	200	8	8	512KB
PII266	Pentium II	266	16	16	512KB
PII300	Pentium II	300	16	16	512KB
R4600	SGI R4600 IP22	100	16	16	NONE
R5000	SGI R5000 IP32	180	32	32	512KB
R10Kip28	SGI R10000 IP28	195	32	32	1MB
R10Kip27	SGI R10000 IP27	195	32	32	4MB
MS70	Sun MicroSPARC II 70	70	8	16	NONE
US140	Sun Ultra1 Model 140	143	16	16	512KB
US2170	Sun Ultra2 Model 2170	167	16	16	512KB
US2200	Sun Ultra2 Model 2200	200	16	16	1MB

Table 3: System Summary

4.1 Results with varying timing methods

There are numerous ways to perform timings. Perhaps the most common method is to generate the matrices A B and C, and then call the appropriate matmul routine. Depending on the matrix and cache sizes, this can make a large difference in the timings. For medium-sized matrices, a significant portion of the matrices will remain in L2 cache from the matrix generation, and thus the memory costs of main memory will not be as prevalent in the timings. For very small matrices, a significant portion of the matrices may remain in L1 cache, and thus the timings will be truly misleading.

Some timers will perform the same operation X times in a row, and report the best timing obtained. This will result in even more optimistic numbers. Obviously, if all matrices fit into some level of the cache, the timings will enjoy cache reuse just as above. However, if only one matrix will fit into cache, there may still be significant cache reuse. For instance, if the off-chip multiply has A in the inner loop, and A fits entirely into some level of cache, the performance reported will not reflect the cost of bringing A into that level of cache.

Finally, many timers set $LDA = M$; in other words, they make all of their matrices contiguous memory. This rules out problems where an ill-chosen leading dimension causes only part of the cache to be used, for instance. It also insures maximal cache reuse. Unfortunately, in actual applications, it is rarely the case that DGEMM is called with the leading dimension equal to the size of matrix (usually, DGEMM is called on submatrices of some larger array).

In all of the timings presented in this paper, a section of memory corresponding to the size of the L2 cache is written to and read from after the matrix generation, so that the matrices must be fetched from main memory by the matmul. We set the leading dimension to the maximal size being timed.

It is readily observed that the method we are using gives a *lower* bound on performance, while the more commonly used method gives an *upper* bound. Why then do we not also just report the upper bound? The reason is that this upper bound will be achieved only in very particular applications (ones that repeatedly use the same memory space, without corrupting the cache between invocations), where the problem size is small or the L2 cache is very large. In short, most users will never see it, and these timings are therefore not indicative of true performance.

Use of appropriate timings is much more important when one is basing software decisions upon it, as our package does. In this case, timing the matmul where things are in cache causes non-optimal code to be produced.

To give the reader a feeling for the kinds of differences the method of timing can cause, we provide a few examples below. In these tables, method 1 is with $LDA = 1000$, and cache flushing before and after the call. Method 2 is setting $LDA = M$, and running the problem 5 times, and choosing the best result. Note that we use the system BLAS for these timings, so that it is clear this is not specific to our implementation.

First, for the machines with large L2 caches, table 4 shows the standard sizes we time in the rest of the paper. As one would expect, as the matrices get larger, caching effects play less and less of a role.

Table 5 shows the same thing for smaller sizes, where the problem is more severe. The Pentium II timings use ATLAS, since we did not have access to the vender BLAS under Linux.

	TIMING	Matrix Order									
SYSTEM	METHOD	100	200	300	400	500	600	700	800	900	1000
AS600	1	227.7	264.4	278.0	282.0	288.8	291.8	291.3	290.3	286.0	291.7
AS600	2	341.5	309.3	323.6	302.2	304.9	291.0	296.4	291.5	286.2	295.1
R10Kip27	1	307.7	307.2	316.2	311.6	296.8	317.9	319.5	316.0	313.0	316.8
R10Kip27	2	317.2	331.5	325.6	317.1	320.3	319.8	324.8	318.6	320.0	318.8

Table 4: Cache flushing with large matrices

	TIMING	Matrix Order					
SYSTEM	METHOD	50	60	70	80	90	100
AS600	1	128.1	147.5	140.6	209.8	186.7	227.7
AS600	2	256.1	442.6	351.4	349.7	298.8	341.5
PII300	1	87.1	103.1	116.4	123.8	134.0	143.0
PII300	2	164.1	173.6	179.3	182.2	184.0	186.7
R10Kip28	1	151.4	176.0	205.5	223.0	232.8	224.7
R10Kip28	2	299.4	300.4	298.8	301.0	300.8	304.4
US2200	1	126.7	141.6	137.9	150.3	150.7	153.1
US2200	2	152.7	166.9	158.4	162.2	158.0	159.4

Table 5: Cache flushing with small matrices

4.2 Square matrix multiply

Table 6 shows the theoretical and observed peaks for matrix multiplication. By observed peak, we mean the best repeatable timing produced on the platform, for any problem size. Where the observed peak differs from the best timings reported in tables 7 and 8, the difference is usually due to using a multiple of the blocking factor.

Abbr.	Clock	Theoretical	DGEMM (MFLOPS)		SGEMM (MFLOPS)	
Name	Rate (MHz)	Peak	VENDOR	ATLAS	VENDOR	ATLAS
AS255	300	Unknown	141.5	175.5	199.6	232.9
AS600	266	532	299.7	282.0	381.8	328.6
HP9K/715	50	100	20.0	49.3	23.2	60.3
HP9K/735	125	250	59.3	119.6	61.5	146.9
POWER2	120	480	337.5	444.3	374.0	450.0
POWERPC	112	224	70.1	100.0	106.9	131.6
P5	166	166	–	75.5	–	91.9
P5MMX	150	150	–	74.5	–	87.3
PPRO	200	200	–	145.4	–	164.2
PII266	266	266	–	170.2	–	215.8
PII300	300	300	–	193.7	–	239.1
R4600	100	33.3	18.9	21.0	21.7	22.9
R5000	180	360	78.8	111.2	119.9	219.3
R10Kip28	195	390	238.7	258.3	307.8	304.4
R10Kip27	195	390	328.5	306.2	353.2	322.3
MS70	70	23.33	23.2	22.0	25.0	23.5
US140	143	286	109.8	164.5	130.3	193.0
US2170	167	334	131.6	188.1	151.9	227.6
US2200	200	400	157.6	221.7	186.1	297.8

Table 6: Theoretical and observed peak MFLOPS

Table 7 (8) shows the times for the vendor-supplied dgemm (sgemm) and ATLAS dgemm (sgemm) across all platforms, with problems sizes ranging from 100 to 1000. In these tables, the LIB column indicates which library the timings are for:

- SYS: system or vendor-supplied GEMM
- ATL: ATLAS GEMM

		Matrix Order									
SYSTEM	LIB	100	200	300	400	500	600	700	800	900	1000
AS255	ATL	157.6	159.2	150.8	167.9	165.5	168.0	169.8	169.8	171.5	170.4
AS255	SYS	120.5	135.5	141.1	138.8	141.1	140.0	141.7	140.4	139.4	140.8
AS600	ATL	170.8	252.2	267.3	282.0	264.6	256.9	259.1	262.5	259.9	259.8
AS600	SYS	227.7	264.4	286.7	288.9	294.4	290.6	292.5	290.7	285.9	290.1
HP9K/735	ATL	100.0	114.3	114.9	112.3	116.8	115.5	116.7	117.8	119.6	119.3
HP9K/735	SYS	50.0	59.3	54.0	43.4	42.6	41.8	41.1	41.5	40.7	41.2
HP9K/715	ATL	40.0	44.4	47.0	46.7	47.6	49.3	47.8	48.7	48.9	48.7
HP9K/715	SYS	20.0	13.6	16.0	9.2	9.4	8.8	9.2	9.2	—	—
POWER2	ATL	—	—	415.4	441.4	423.7	432.0	436.8	428.5	428.8	421.9
POWER2	SYS	—	—	337.5	304.8	320.5	289.9	300.9	295.1	313.5	294.1
POWERPC	ATL	100.0	94.1	98.2	97.7	92.9	93.7	93.5	89.7	89.8	89.4
POWERPC	SYS	66.7	66.7	70.1	69.6	70.0	67.7	67.7	68.3	68.1	66.9
P5	ATL	65.7	68.7	73.4	75.5	75.2	72.1	73.1	—	—	—
P5MMX	ATL	66.0	68.7	70.7	72.6	73.4	74.0	74.2	74.5	—	—
PPRO	ATL	116.8	135.6	136.5	140.8	143.1	142.9	144.4	143.1	142.2	142.1
PII266	ATL	123.8	159.6	160.0	163.6	168.1	170.2	168.9	170.2	169.0	167.6
PII300	ATL	141.5	176.8	182.5	187.5	192.8	193.5	192.8	192.2	191.1	190.9
R4600	ATL	19.2	20.1	20.5	20.6	20.6	20.7	20.8	20.8	20.8	20.9
R4600	SYS	17.7	18.4	18.6	18.7	18.8	18.9	18.9	18.9	18.9	19.0
R5000	ATL	97.3	107.2	107.4	106.7	108.7	109.2	108.7	107.9	109.4	108.9
R5000	SYS	71.8	78.3	76.0	78.8	77.9	77.7	76.3	76.7	75.3	76.4
R10Kip28	ATL	203.8	253.6	256.6	243.9	238.6	242.9	242.2	247.0	246.6	247.7
R10Kip28	SYS	216.7	218.7	238.4	216.6	223.2	235.0	210.0	230.4	220.3	226.6
R10Kip27	ATL	232.8	274.5	292.3	293.4	306.2	304.4	303.6	298.5	296.7	296.7
R10Kip27	SYS	265.4	308.8	319.9	314.1	319.4	327.9	328.3	328.5	328.1	322.1
MS70	ATL	20.0	21.1	21.5	21.6	21.8	21.9	22.0	21.9	22.0	22.0
MS70	SYS	21.9	22.8	22.9	23.1	23.1	23.1	23.1	23.2	22.9	22.5
US140	ATL	128.6	157.2	159.7	160.7	162.5	164.1	161.7	159.8	162.8	161.7
US140	SYS	103.8	109.8	109.1	106.4	108.7	108.1	108.0	103.1	106.3	105.4
US2170	ATL	147.5	179.9	188.1	185.4	179.8	178.4	182.7	184.9	186.4	183.3
US2170	SYS	120.1	126.3	125.4	124.4	123.0	122.3	123.2	122.9	120.2	119.6
US2200	ATL	178.1	210.6	205.5	212.2	214.5	216.7	216.4	216.6	221.7	212.1
US2200	SYS	147.1	157.6	156.6	157.9	155.9	152.0	152.1	154.3	151.2	150.0

Table 7: System and ATLAS DGEMM comparison across platforms

		Matrix Order									
SYSTEM	LIB	100	200	300	400	500	600	700	800	900	1000
AS255	ATL	170.8	221.5	228.6	229.7	222.2	226.9	225.4	227.1	226.4	228.8
AS255	SYS	186.3	192.9	194.8	193.7	197.6	197.2	197.0	197.9	196.1	198.7
AS600	ATL	227.7	298.1	304.0	316.8	316.2	318.4	312.2	315.4	312.1	318.6
AS600	SYS	292.7	348.8	352.4	359.3	373.9	371.3	377.5	379.5	372.8	379.4
HP9K/715	ATL	50.0	37.1	55.7	58.2	59.0	59.6	59.6	60.3	60.0	60.2
HP9K/715	SYS	14.3	23.2	18.6	10.7	11.7	11.5	11.2	10.5	10.3	10.1
HP9K/735	ATL	—	145.5	142.1	143.8	144.5	145.9	143.2	146.9	145.7	146.4
HP9K/735	SYS	50.0	61.5	56.2	46.5	45.7	45.3	45.2	43.5	43.1	42.8
POWER2	ATL	—	—	450.0	412.9	431.0	419.4	420.9	424.9	430.1	426.5
POWER2	SYS	—	—	360.0	365.7	367.6	357.0	363.0	363.1	360.0	363.6
POWERPC	ATL	101.2	124.0	130.2	131.0	131.6	131.6	131.4	129.7	129.4	131.0
POWERPC	SYS	94.4	102.5	105.3	99.2	106.5	103.5	106.9	97.2	101.7	106.2
P5MMX	ATL	77.1	80.9	83.2	84.7	85.7	86.4	86.3	86.6	86.5	85.9
PPRO	ATL	137.8	154.7	157.3	159.4	159.8	161.2	160.5	161.7	163.0	162.5
PII266	ATL	153.8	200.1	209.0	210.6	213.8	212.8	214.3	215.4	214.9	215.8
PII300	ATL	167.0	215.1	227.8	231.1	234.4	230.6	237.0	237.6	237.0	238.1
R5000	ATL	176.7	207.8	215.8	215.9	209.0	215.5	213.0	215.8	212.4	214.2
R5000	SYS	105.3	114.9	119.3	115.7	116.6	118.1	116.3	116.7	118.1	117.1
R10Kip28	ATL	220.2	286.5	301.2	288.4	290.7	288.4	286.8	289.9	288.7	290.2
R10Kip28	SGI	269.9	285.5	300.5	279.0	298.3	302.2	300.9	298.9	300.3	300.7
R10Kip27	ATL	284.7	308.9	317.8	323.4	321.6	326.0	326.8	326.7	325.8	323.7
R10Kip27	SYS	321.5	339.4	345.4	347.8	340.4	343.7	340.7	348.5	349.3	349.2
MS70	ATL	21.4	22.9	23.0	23.3	23.3	23.5	23.5	23.6	23.6	23.6
MS70	SYS	23.8	24.6	24.8	25.0	24.9	25.0	24.9	25.0	24.9	25.0
US140	ATL	156.7	171.6	184.6	190.1	191.0	190.6	192.7	193.0	191.8	192.7
US140	SYS	126.1	128.6	132.0	130.1	130.9	130.0	130.7	131.0	128.9	128.8
US2170	ATL	189.3	218.3	227.6	224.3	223.3	223.6	223.2	226.9	225.9	226.2
US2170	SYS	139.6	150.5	151.9	150.6	150.8	150.2	149.7	148.2	147.8	147.7
US2200	ATL	233.2	273.8	280.4	278.4	280.9	285.1	282.7	283.9	287.6	285.1
US2200	SYS	171.9	182.5	187.2	187.7	185.4	183.4	182.7	180.9	178.2	178.8

Table 8: System and ATLAS SGEMM comparison across platforms

4.3 LU timings

In order to demonstrate that these routines provide good performance in practice, we timed LAPACK's LU factorization linking to both the vendor supplied DGEMM, and the one produced by ATLAS. The other BLAS used by LU were the fastest available, usually the ones supplied by the vendor.

The blocking factor for the factorization to use was determined by timing all blocking factors between 1 and 64, and choosing the one that performed best for the LU factorization of a matrix of order 500. As with previous timings, caches were flushed before the start of the algorithm.

Table 9 shows the LU performance on several platforms. The LIB column is overloaded to convey both the DGEMM used (A for ATLAS, S for system), and the blocking factor chosen (for instance, A(40) in this column indicates a run using ATLAS's DGEMM, using a blocking factor of 40).

		Matrix Order									
SYSTEM	LIB	100	200	300	400	500	600	700	800	900	1000
AS255	A(44)	49.2	70.0	85.0	94.9	98.6	102.4	105.0	108.2	110.0	111.0
AS255	S(32)	60.6	83.4	97.1	101.7	101.9	103.7	104.4	105.5	105.4	105.2
AS600	A(56)	89.4	126.7	145.9	157.5	167.7	175.0	181.1	178.1	182.0	180.8
AS600	S(32)	108.8	156.1	186.7	198.7	217.8	217.0	231.3	217.2	213.6	204.8
POWERPC	A(20)	45.5	52.3	57.1	54.0	53.8	55.2	54.6	55.6	55.2	57.4
POWERPC	S(22)	47.8	51.3	50.0	49.6	51.8	52.2	52.2	51.4	53.6	53.7
POWER2	A(60)	170.1	230.6	225.0	213.3	203.3	211.8	211.7	218.8	228.2	233.9
POWER2	S(31)	203.3	224.6	236.8	224.6	213.7	200.0	198.8	191.8	200.0	202.0
R10Kip28	A(64)	110.0	146.0	171.1	166.8	160.7	163.2	159.3	156.3	159.8	160.1
R10Kip28	S(56)	129.4	167.7	190.4	185.7	165.7	167.6	162.3	151.4	166.8	163.4
R10Kip27	A(40)	146.7	181.0	206.1	214.2	226.0	226.4	232.2	227.4	226.9	221.7
R10Kip27	S(38)	156.4	199.4	226.5	236.9	248.6	250.8	256.2	252.7	249.7	243.5
PII	A(35)	65.3	92.1	102.5	105.1	109.1	112.7	116.4	119.2	122.1	124.3
US2200	A(36)	103.6	132.5	140.8	147.9	147.5	151.4	155.6	157.9	161.0	160.3
US2200	S(32)	107.4	126.6	133.3	134.2	127.2	130.9	133.6	132.5	132.8	125.7

Table 9: Double precision LU timings on various platforms

The reader may notice that the LU times are not in general as good as the GEMM timings. The first reason is that we have made sure to include the platforms where either the vendor was faster, or our win was marginal, so we can see any adverse effects in detail. In the similar vein, it can be noted that the LU timings for the platform R10Kip28 are worse for ATLAS than using the vendor DGEMM, even though ATLAS was faster for large matrix multiplication. This is due to the fact that LU does not perform DGEMM on the entire matrix, but rather uses a rank-K update. To get a matrix multiply of size equal to the square matrix multiply of order 200 (the first case where ATLAS beat the vender supplied BLAS), one must run a 625 size LU. Even then, subsequent DGEMM calls will be on smaller matrices. This is why we see these discrepancies in DGEMM and LU timings.

In order to demonstrate that matrix size is the primary reason for this difference, we show performance for very large LU factorizations on two select platforms in table 10

		Matrix Order									
SYSTEM	LIB	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
POWER2	A(60)	239.2	246.7	252.1	256.6	261.6	261.6	270.2	276.1	276.3	278.5
POWER2	S(31)	204.5	205.9	212.2	208.4	215.1	215.0	213.7	217.3	217.2	221.2
R10Kip28	A(64)	161.2	161.1	162.8	163.2	163.9	162.4	165.9	163.5	167.0	166.9
R10Kip28	S(56)	165.0	161.7	161.7	159.9	164.4	142.9	155.8	157.8	155.7	147.9

Table 10: Asymptotic double precision LU performance

4.4 Threaded GEMM timings

Several of the platforms we have surveyed have multiple processors accessible through threading. We have implemented a simple parallel matrix multiply using pthreads. Table 11 shows the threaded timings for these platforms. Of these architectures, only SUN provided a vendor-supplied threaded multiply, so we cannot report system numbers for other platforms. Further, the POWERPC unrepeatably produced incorrect results; this problem has not been tracked down further.

In the table, the column LIB has been overloaded to provide the number of processors available. A(2) indicates ATLAS run on two processors; S(8) indicates the system or vendor supplied DGEMM run on eight processors.

		Matrix Order									
SYSTEM	LIB	100	200	300	400	500	600	700	800	900	1000
POWERPC	A(8)	140.1	247.9	307.1	340.6	320.6	343.7	381.9	381.9	414.6	423.9
PPRO	A(2)	147.2	190.3	210.4	231.8	240.1	239.3	239.6	245.4	246.8	210.9
PII300	A(2)	165.4	292.3	313.7	319.3	350.4	346.6	349.1	351.4	341.6	348.2
US2200	A(2)	235.4	412.2	421.0	413.6	435.3	404.1	431.5	434.1	415.9	445.4
US2200	S(2)	264.9	306.6	305.9	320.4	303.8	305.7	302.7	298.3	296.4	297.2

Table 11: Threaded DGEMM timings across various platforms

5 Comparison to Other Work

There are other efforts to produce optimal codes through code generation. The closest parallel to ATLAS is seen in the PHiPAC [3] effort. PHiPAC also deals with using a code generator for BLAS work. Since PhiPAC predates ATLAS by several years, it is natural to ask what the differences between the packages are, and perhaps why the ATLAS project was begun.

ATLAS was started because we needed an optimized DGEMM for Pentium's running Linux. The authors of PhiPAC reported disappointing performance for PHiPAC on the Linux/Intel platform (this is no longer the case). When we examined the issue of creating an efficient DGEMM for this platform, it was readily apparent that it would require only a little more effort to make the work portable.

If this answers the question of why ATLAS was begun in the first place, it does not tell how it is different from PHiPAC. The main difference is in the complexity of the approach. ATLAS puts all system-specific code in one square on-chip multiply. It then uses the off-chip code to coerce all problems to this format. ATLAS further counts on a level 1 cache being accessible by the floating point unit, in order to be able to make the simplifying step of writing the on-chip multiply. This means we need generate/time only one routine for each new platform. This has resulted in a code generator that finishes in a relatively short time (generally, 1-2 hours), even though the operations being timed are artificially inflated in order to ensure repeatability.

PHiPAC, on the other hand, chose the more comprehensive approach of directly optimizing each individual operation. This means different code will be generated for each transpose combination, for instance. This results in a lengthy installation process (usually, a matter of days), as multiple cases for every routine must be generated and timed.

Neither of these approaches are "better" than the other. The approach used by PHiPAC will probably yield better performance for very small problems (since they may avoid any unnecessary data copies), or on machines with no L1 cache. The same methods of code generation used in the level 3 BLAS should work pretty much unchanged for level 1 and 2. However, the cost of this increased generality is seen in the longer installation time, and in performance which may be more sensitive to various factors such as poorly chosen leading dimensions (ATLAS is somewhat shielded from such factors by its data copy), etc.

The best way to determine which of these packages a user should use is to time them with the specific application. If the user wishes to compare raw performance as reported in the publications, it should be mentioned that the PHiPAC timing method is not the same as used in this paper. Current PHiPAC timings as reported in [3] use timing method 2 discussed in section 4.1. This means that their performance numbers do not in general include the costs of bringing operands into cache. Section 4.1 can give the reader an idea of the effects of this.

6 Downloading ATLAS

The alpha release of ATLAS can be found at `www.netlib.org/utk/projects/atlas`. Installation instructions are provided in the supplied `README` file.

7 Future Work

Currently the code generator in ATLAS works only for matrix-multiplication, which is the basic operation underlying other BLAS. We will extend this generator to other Level 1, 2 and 3 BLAS. For the other Level 3 BLAS, we will consider using the GEMM-based Level 3 BLAS [7], which implement all the other Level 3 BLAS (solving triangular systems for many right hand sides, etc.) in terms of matrix-multiplication. The next most important operation is matrix-vector multiplication. We will develop a version of ATLAS that can produce a full set of BLAS, run it on a variety of architectures of interest, and make it publicly available for others to perform their own optimizations on architectures, problem shapes, sizes and alignments of their choice.

We are planning to study other architectures of interest, including development of cost models, prototyping hand-written BLAS, and developing algorithmic generators appropriate to these architectures. Future RISC processors with vector instructions will require loop lengths that match the optimal vector lengths. SMPs will require load balancing while avoiding "false sharing" of cache lines by different processors. Different ways of thread management will also have to be considered.

We are planning to develop and refine algorithms that exploit sparse BLAS. Sparse matrix-vector multiplication is a essential kernel in most iterative algorithms for large matrix problems. Optimizing its performance requires a number of both architecture and matrix dependent transformations [8]. We will study how to extend ATLAS to optimize sparse-matrix vector multiplication, where the optimizations may depend on the sparsity structure.

8 Conclusions

We have demonstrated the ability to produce highly optimized matrix multiply for a wide range of architectures based on a code generator that probes and searches the system for an optimum set of parameters. This avoids the tedious task of generating by hand routines optimized for a specific architecture. We believe these ideas can be expanded to cover not only the Level 3 BLAS, but Level 2 BLAS as well. In addition there is scope for additional operations beyond the BLAS, such as sparse matrix vector multiplication, and FFTs.

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, 1995. 324 pages.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997.

[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 International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC.

[4] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[5] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

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

[7] B. Kågström, P. Ling, and C. Van Loan. Portable High Performance GEMM-based Level 3 BLAS. In R. F. Sincovec et al., editor, *Parallel Processing for Scientific Computing*, pages 339–346, Philadelphia, 1993. SIAM.

[8] S. Toledo. Improving instruction-level parallelism in sparse matrix-vector multiplication using reordering, blocking, and prefetching. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1997.

Appendix: BLAS and compiler details

This section exists to provide further details regarding the compilers and BLAS used in our timings. Table 12 shows the system BLAS that were used for the timings. Tables 13 and 14 show the compiler version and flags used in compiling the on-chip matrix multiply.

System	link	version
AS255	-ldxml	DXML V3.3a
AS600	-ldxml	DXML V3.2
POWER2	-lesslp2	essl 2.2.2.2
POWERPC	-lessl	essl 2.2.2.2
HP9K/715	-lvec	Revision 73.4
HP9K/735	-lvec	Revision 73.14
R4600	-lblas	Standard Execution Environment (Fortran 77, 4.0.2)
R5000	-lblas	Standard Execution Environment (Fortran 77, 7.1)
R10Kip27	-lblas	Standard Execution Environment (Fortran 77, 7.1)
R10Kip28	-lblas	Standard Execution Environment (Fortran 77, 6.2)
MS70	-xlic_lib=sunperf	Sun Performance Library 1.2
US140	-xlic_lib=sunperf	Sun Performance Library 1.2
US2170	-xlic_lib=sunperf	Sun Performance Library 1.2
US2200	-xlic_lib=sunperf	Sun Performance Library 1.2

Table 12: BLAS library and version

System	Compiler version
AS255	cc Digital UNIX Compiler Driver 3.11 DEC C V5.2-033 on Digital UNIX V4.0 (Rev. 564)
AS600	cc Digital UNIX Compiler Driver 3.11 DEC C V5.2-033 on Digital UNIX V4.0 (Rev. 564)
POWER2	xlC.C V3.1.4.0
POWERPC	xlC.c V3.1.4.0
HP9K/715	HP92453-01 A.09.75 HP C Compiler (CXREF A.09.75)
HP9K/735	
P5	gcc version 2.7.2.1
P5MMX	gcc version 2.7.2.1
PPRO	gcc version 2.7.2.2.f.2
PII266	gcc version 2.7.2.1.f.1
PII300	gcc version 2.7.2.1
R4600	Base Compiler Development Environment, 5.3
R5000	Base Compiler Development Environment, 7.0
R10Kip27	Compiler Development Environment, 7.1
R10Kip28	Base Compiler Development Environment, 7.0
MS70	cc: WorkShop Compilers 4.2 30 Oct 1996 C 4.2
US140	cc: WorkShop Compilers 4.2 30 Oct 1996 C 4.2
US2170	cc: WorkShop Compilers 4.2 30 Oct 1996 C 4.2
US2200	cc: WorkShop Compilers 4.2 30 Oct 1996 C 4.2

Table 13: Compiler and version

System	Compiler & flags
AS255	`cc -arch host -tune host -std -assume aligned_objects -O5`
AS600	`cc -arch host -tune host -std -assume aligned_objects -O5`
POWER2	`xlc -qarch=pwr2 -qtune=pwr2 -qmaxmem=-1 -qfloat=hssngl -qansialias -qfold -O`
POWERPC	`xlc -qarch=ppc -qtune=604 -qmaxmem=-1 -qfloat=hssngl -qansialias -qfold -O`
HP9K/735	`-Aa +O2`
HP9K/715	`-Aa +O2`
P5	`gcc -fomit-frame-pointer -O`
P5MMX	`gcc -fomit-frame-pointer -O`
PPRO	`gcc -fomit-frame-pointer -O`
PII266	`gcc -fomit-frame-pointer -O`
PII300	`gcc -fomit-frame-pointer -O`
R4600	`cc -O2 -mips2 -Olimit 15000`
R5000	`cc  -n32 -mips4 -r5000 -OPT:Olimit=15000 -TARG:platform=ip32_5k -TARG:processor=r5000 -LOPT:alias=typed -LNO:blocking=OFF -O2`
R10Kip27	`cc  -64 -mips4 -r10000 -OPT:Olimit=15000 -TARG:platform=ip27 -LOPT:alias=typed -LNO:blocking=OFF -O2`
R10Kip28	`cc  -64 -mips4 -r10000 -OPT:Olimit=15000 -TARG:platform=ip28 -LOPT:alias=typed -LNO:blocking=OFF -O2`
MS70	`cc -xchip=micro2 -xarch=v8 -dalign -fsingle -fsimple=1 -xsafe=mem`
US140	`cc -dalign -fsingle -xtarget=ultra1/140 -xO5 -fsimple=1 -xsafe=mem`
US2170	`cc -dalign -fsingle -xtarget=ultra2/2170 -xO5 -fsimple=1 -xsafe=mem`
US2200	`cc -native -dalign -fsingle -xO5 -fsimple=1 -xsafe=mem`

Table 14: Compiler flags