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.

Automatically Tuned Linear Algebra Software *

R. Clint Whaley † Jack J. Dongarra, ‡

December 20, 1998

Contents

1 Abstract **0**

2 Motivation **3**

3 ATLAS **5**

3.1 The ATLAS Approach 5
3.2 Building the General Matrix Multiply From the On-Chip Multiply 5
3.2.1 Choosing the Correct Looping Structure 9
3.2.2 Blocking for Higher Levels of Cache 9
3.3 Generation of the On-Chip Multiply 11
3.3.1 Instruction Cache Reuse 12
3.3.2 Floating Point Instruction Ordering 12
3.3.3 Reducing Loop Overhead 13
3.3.4 Exposing Parallelism 13
3.3.5 Finding the Correct Number of Cache Misses 13
3.3.6 Putting It All Together 14
3.3.7 Cleanup Code 15
3.4 Why Can't the Compiler Do This? 15
3.5 Requirements for Good Performance 15

*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, and Mathematical Sciences Section, ORNL, Oak Ridge, TN 37831, `dongarra@cs.utk.edu`

4 Results **16**
4.1 Results with Varying Timing Methods . . . 18
4.2 Square Matrix Multiply . . . 20
4.3 Matrix Multiply Results . . . 20
4.4 Preliminary Results with GEMM-Based Level 3 BLAS . . . 22
4.5 LU Timings . . . 24

5 Comparison to Other Work **26**

6 Downloading ATLAS **26**

7 Future Work **27**

8 Conclusions **27**

A BLAS and compiler details **29**

List of Tables

1 System Summary . . . 17
2 Cache flushing with large matrices . . . 19
3 Cache flushing with small matrices . . . 19
4 Theoretical and observed peak MFLOPS . . . 20
5 System and ATLAS DGEMM comparison across platforms (MFLOPS) . . . 21
6 500x500 System and GEMM-based BLAS comparison across platforms (MFLOPS) 23
7 Double precision LU timings on various platforms . . . 25
8 BLAS library and version . . . 29
9 Compiler and version . . . 30
10 Compiler flags . . . 31

List of Figures

1 ATLAS/vendor performance preview . . . 4
2 One step of matrix-matrix multiply . . . 7
3 General matrix multiplication with A as innermost matrix . . . 8
4 General matrix multiplication with B as innermost matrix . . . 8

2 Motivation

Today's microprocessors have peak execution rates exceeding 1 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 de facto 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 an "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 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.

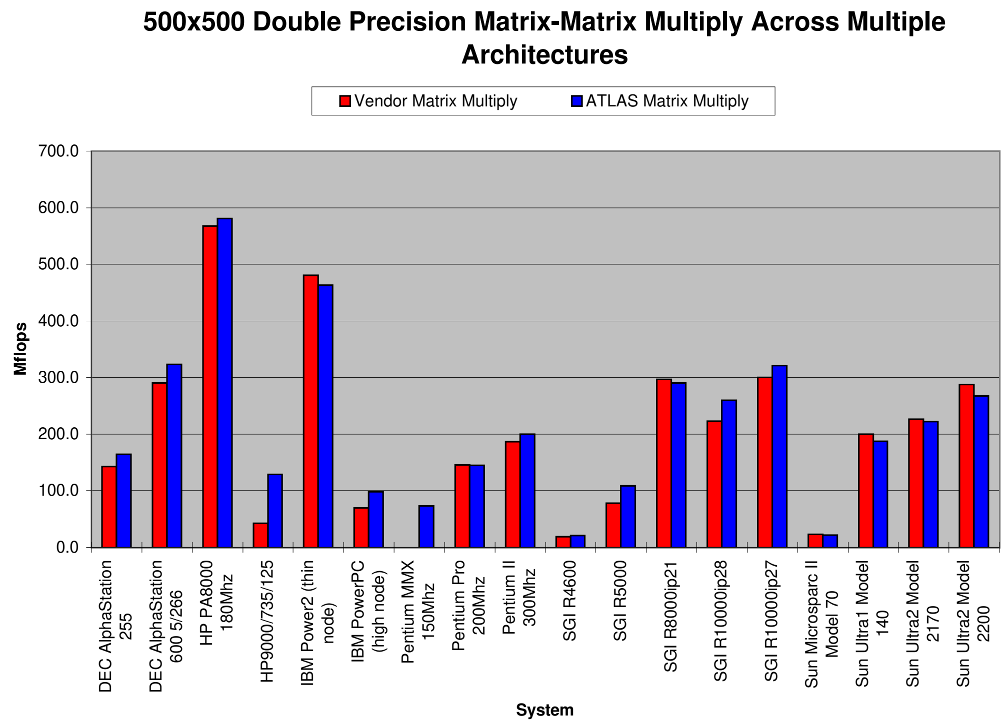

Figure 1: ATLAS/vendor performance preview

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 is 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, cache contained, (i.e., in Level 1 (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 (section 3.3) describes the on-chip multiply and its code generator in detail. For this section, it is enough to know that we have efficient on-chip matrix-matrix multiplies of the forms $C \leftarrow A^TB$, $C \leftarrow A^TB + C$, $C \leftarrow A^TB - C$, and $C \leftarrow A^TB + \beta C$, 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 have the option of writing the result of the on-chip multiply 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., we can ensure during the malloc that we begin on a cache-line boundary)

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

The disadvantage of using $\hat{C}$ is that an additional write to C is required after the on-chip operations have completed. This cost is minimal if we make many calls to the on-chip multiply (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. Writing to $\hat{C}$ essentially doubles the write cost, which is unacceptable. The routines therefore employ a heuristic to determine if the number of times the on-chip multiply 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, the 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 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 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

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, 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.

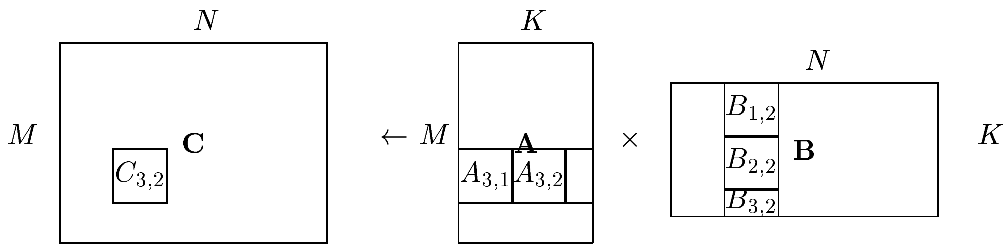

Figure 2: One step of matrix-matrix multiply

If we are 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 $0 \leq i < \lceil M/N_B \rceil$, $0 \leq j < \lceil N/N_B \rceil$:

1. Call on-chip multiply 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 on-chip multiply 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, $\forall k, 1 \leq k < \lceil K/N_B \rceil$. 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. $\hat{C}$ now holds the product of the row panel of A with the column panel of B, so we now perform the block operation $C_{i,j} \leftarrow \hat{C}_{i,j} + \beta C_{i,j}$.

If we are writing directly to C, this action becomes:

1. Call on-chip multiply 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 on-chip multiply 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 on this inner loop, we have the loop orderings giving us our two algorithms for off-chip matrix multiplication. Figures 3 and 4 give the pseudo-code for these two algorithms, assuming we are writing directly to C. 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 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 3: 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 4: General matrix multiplication with B as innermost matrix

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 the entire inner matrix)
- If neither matrix fits completely into L2 cache, put the one with the largest panel that 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 (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 Blocking for Higher Levels of Cache

Note that we define the Level 1 (L1) cache is 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, and are often non-associative and write-through. Higher levels of cache or more often non-write-through, with varying degrees of associativity and differing replacement polices.

Because of the wide variance in high level cache behaviors, our present cache detection algorithm is not sophisticated enough to reliably detect multiple levels of cache. Therefore, by default the only cache size that is known to ATLAS is that of the L1 cache. This means that we cannot automatically determine when it is appropriate to perform blocking for these higher levels of cache, as we do for L1.

At installation time, the installer may supply ATLAS the cache size of one of the higher level caches to ATLAS (this quantity is called `CacheEdge`), and ATLAS will then perform explicit cache blocking, when appropriate.

Explicit cache blocking is required only 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 setting `CacheEdge`. This is expressed formally below; Notice that conditions 1 and 2 do not require explicit cache blocking, so the user gets this result even if `CacheEdge` is not set.

The explicit cache blocking strategy discussed in 4 below assumes that we have a case where the panels of A and B overflow a particular level of cache. In this case, we can easily partition the K dimension of the input matrices so 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.

More concretely, ATLAS allows an installer to supply the value `CacheEdge`. This value is set to the size of the level of cache the user wants explicit blocking support for. 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 we solve this equation for K, we have the maximal K (call this quantity K_m) which will, assuming we copy all of the inner matrix up front, allow us to reuse the outer matrix panel N/N_B times. We now translate our original matrix multiply into $\lceil K/K_m \rceil$ rank-K_m updates.

Assuming 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) reads 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 is 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 we copy entire inner matrix)
   OR ($3KN_B + {N_B}^2 \leq S_L$ AND we copy a panel of the inner matrix in the inner loop, thus doubling the inner panel's footprint in the cache))
   - $NK + \frac{MNK}{N_B}$ 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 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 we do not set `CacheEdge`, and thus do no explicit level L blocking, the memory fetch becomes:
   - $\frac{2MNK}{N_B}$ 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 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 fetch from that given in 3 to:
   - $NK + \frac{MNK}{N_B}$ 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. It is used as an alternative to case 3. Note that, assuming we have a non-write-through cache, we reduce one of the reads from $O(n^3)$ to $O(n^2)$ at the cost of raising the number writes from $O(n^2)$ to $O(n^3)$. At first glance this may appear to be a poor bargain indeed, since writes are generally more expensive than reads. There are 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 $K_m < N_B$, so we know that the number of writes is at most the same as the number of reads we saved. For many problems $K_m \gg N_B$, so the number of writes is much less. Finally, 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 fact alone can make writing faster than reading on some systems.

In practice, 4 has been shown to be at least 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 Alpha- based systems (which have three layers of cache).

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 we cannot 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 you have 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; there may be implicit blocking going on, as in cases 1 and 2 above.

Another interesting idea for performing explicit cache blocking is where multiple matrix panels will fit into level L 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 level L 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 (case 2 and 1, respectively). Within this restriction, we reduce the number of times we must copy A.

This idea was promising enough that it was implemented. However, the range between X of 1 and the entire matrix was typically not large enough to make the speedup worthwhile, and so, for the sake of simplicity, we have chosen not to support this kind of blocking.

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 transpose case is required, which we have chosen as $C \leftarrow A^T B + C$ (we support differing values of β as well, but this is merely a convenience to avoid scaling). 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 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. 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 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. 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 thus 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 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.

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 down 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 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), and thus dictate the maximum amount of M and/or N loop unrolling to perform.

The first step of the timing figures the size of the L1 cache, if the user has not supplied it. 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, we check only powers of 2. This means that a 48K cache would probably be detected as a 32K cache, for instance. We have not found this problem severe enough problem to justify the additional installation time it would take to remedy it.

Next, we attempt to determine something about the floating point units of the platform. We need to understand whether we have a combined muladd unit, or whether we need to allow for independent multiply and add pipes. To do this, we generate simple register-to-register code which performs the required multiply-add using a combined muladd and separate multiply and add pipes. We try both variants using code which implies various pipeline lengths. Therefore, once we finish timing this simple register-register code, we have a good empirical idea of the kinds of instructions to issue (muladd or separate multiply and add), the pipeline depth, and even some idea if we have multiple floating point units or not. With this data in hand, we are ready to begin actual on-chip multiply timings.

In general, K loop unrollings of 1 or K have tended to produce the best results. Thus we time 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 we try to ensure we have not left out an opportunity to gain greater performance by trying a wide range of K loop unrolling factors with the best case code generated for the unrollings factors of 1 or K.

At the beginning of the search a number of possible blocking factors are tried using set amount of M and N loop unrolling (at the present, 2x2), from which an initial blocking factor is chosen.

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, we again try all blocking factors, and various latency and K-loop unrolling 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.

Because of the number of routines required in this cleanup strategy, we generate only the $\beta = 1$ case (eg., the operation done by the cleanup is always $C \leftarrow A^T B + C$). Other β choices are accommodated by scaling.

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 for achieving good performance:

1. There is a cache from which the floating point unit can fetch operands cheaply.

2. The platform possesses an adequate C compiler.

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, register blocking, etc.). Some compilers ignore the `register` keyword, and/or reorder the loops even though they are already optimal. Also, if the compiler has limits on its use of the hardware (for instance, it only emits code for a subset of the actual physical registers), there is nothing ATLAS can do about it. 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).

There are two systems where we attempted an ATLAS installation, and had unacceptable performance. The platforms, and the reason for the performance loss is summarized below:

- *Intel i860*: inadequate C compiler
- *Cray T3E*: inadequate C compiler/lack of L3 cache

On Intel i860 we were unable to achieve a DGEMM speed exceeding 12MFLOPS. The system supplied DGEMM executed at rates above 40Mflop. We were only able to approach this speed by rewriting the on-chip multiply in terms of fortran77. Since changing languages appears to have such a salutary effect, it is easy to believe that the compiler is at fault. Further, the case that got the best performance involved 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 complexity inherit in outer loop unrolling. There is a `gcc` installation for the i860, and it would be interesting to see if results are better with this alternative compiler.

The nodes of the SGI/CRAY T3E we had access to are DEC Alpha 21164 RISC processors, running at 450MHz. Our top MFLOPS on this system was on the order of 400MFLOPS, far below both the theoretical peak and that enjoyed by the vendor-supplied BLAS. ATLAS performs well on other machines built around this chip, but since SGI/CRAY removed the Level 3 cache supplied by DEC, we are not able to determine if the problem is the differing compilers or hardware.

As hinted at above, 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.

4 Results

In this section we present double precision (64-bit floating point arithmetic) timings across various platforms. ATLAS also supplies a single precision version; The single precision results are not markedly different than the double, and so we omit them in the interest of brevity.

The timings presented here 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.

Table 1 shows the configurations of the various platforms which we have installed and timed the package on.

Appendix A has several tables providing further details. Table 8 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 for the HP9K/735, and so had to compare against their vector library (which describes itself as optimized for the 9000 series) instead. Similarly, Intel does not officially supply a BLAS for their chips running Linux. However, it is possible to get a unofficial version of the library which works under Linux. This is what we compare against in our timings below.

Tables 9 and 10 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
DCG533	DCG LX Series 21164a-533	533	8	8	96KB & 2MB
HP9K/735	HP 9000/735/125	125	256	256	NONE
HPPA8K	HP PA8000	180	1Mb	UNKNOWN	NONE
POWER2	IBM Power2 (thin node)	135	128	32	NONE
POWERPC	IBM PowerPC 604 (high node)	112	16	16	1MB
P5MMX	Pentium with MMX	150	16	16	256KB
PPRO	Pentium Pro	200	8	8	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
US2170	Sun Ultra2 Model 2170	167	16	16	512KB
US2200	Sun Ultra2 Model 2200	200	16	16	1MB

Table 1: 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 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 highest cache level 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 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 may cause less 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 sets $LDA = M$, runs the problem 5 times, and chooses 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 high level caches, table 2 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 3 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.

SYSTEM	TIMING METHOD	Matrix Order 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 2: Cache flushing with large matrices

SYSTEM	TIMING METHOD	Matrix Order 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

Table 3: Cache flushing with small matrices

4.2 Square Matrix Multiply

Table 4 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 table 5, the difference is usually due to using a multiple of the blocking factor.

Abbr.	Clock	Theoretical	DGEMM (MFLOPS)	
Name	Rate (MHz)	Peak	VENDOR	ATLAS
AS255	300	300	141.5	175.5
AS600	266	532	299.7	282.0
DCG533	533	1066	–	595.3
HPPA8K	180	720	582.7	605.6
HP9K/735	125	250	59.3	119.6
POWER2	135	540	481.9	464.3
POWERPC	112	224	70.1	100.0
P5MMX	150	150	–	74.5
PPRO	200	200	–	145.4
PII300	300	300	–	193.7
R4600	100	33.3	18.9	21.0
R5000	180	360	78.8	111.2
R8Kip21	90	360	261.3	286.1
R10Kip28	195	390	238.7	258.3
R10Kip27	195	390	328.5	306.2
MS70	70	23.33	23.2	22.0
US2170	167	334	131.6	188.1
US2200	200	400	309.3	265.4

Table 4: Theoretical and observed peak MFLOPS

Table 5 shows the times for the vendor-supplied dgemm and ATLAS dgemm 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

4.3 Matrix Multiply Results

		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	227.7	273.2	310.8	315.3	323.4	321.7	318.6	311.2	309.9	304.0
AS600	SYS	227.7	256.1	283.7	285.7	290.7	289.7	292.7	289.9	284.8	288.3
DCG533	ATL	409.6	528.5	588.3	577.4	585.8	587.5	594.3	592.1	595.3	593.1
HPPA8K	ATL	–	–	599.7	556.1	581.0	591.4	596.1	598.4	604.5	605.6
HPPA8K	SYS	–	–	539.5	556.0	567.6	568.1	566.4	571.7	566.7	582.7
HP9K/735	ATL	100.0	123.1	128.6	127.7	128.9	131.3	130.2	131.6	134.0	132.6
HP9K/735	SYS	50.0	59.3	54.0	43.4	42.6	41.8	41.1	41.5	40.7	41.2
POWER2	ATL	—	—	—	474.1	471.7	464.5	451.3	471.9	473.4	464.0
POWER2	SYS	—	—	—	492.3	480.8	480.0	479.7	478.5	484.4	478.5
POWERPC	ATL	80.9	95.1	95.6	97.2	98.7	97.9	98.0	98.6	100.2	99.3
POWERPC	SYS	66.9	70.3	71.3	70.8	70.5	69.8	69.8	69.1	68.7	68.6
P5MMX	ATL	64.2	68.8	69.7	71.6	71.9	72.5	72.6	72.6	73.0	70.1
PPRO	ATL	120.1	137.3	140.2	142.2	145.2	145.8	147.0	147.6	142.7	143.0
PPRO	SYS	111.7	142.8	138.6	148.4	146.0	154.7	150.9	157.1	156.0	155.9
PII300	ATL	156.0	193.4	195.8	202.1	201.3	204.4	206.2	205.5	204.8	208.0
PII300	SYS	155.4	187.0	175.8	191.6	184.3	193.2	187.3	198.2	194.7	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	98.0	105.1	102.9	104.0	106.5	107.2	107.9	108.0	107.2	108.0
R5000	SYS	85.3	73.2	75.2	74.3	68.1	72.7	69.5	66.7	70.6	68.1
R8Kip21	ATL	220.5	268.9	282.5	293.9	291.5	292.1	289.5	287.8	283.4	287.8
R8Kip21	SYS	243.7	292.5	289.8	293.1	297.1	296.7	295.4	294.0	295.0	295.2
R10Kip27	ATL	253.1	302.6	316.7	323.0	322.1	323.1	324.0	323.1	323.6	324.2
R10Kip27	SYS	283.5	314.4	318.0	309.5	312.7	318.3	323.0	318.0	317.4	318.9
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
US2170	ATL	201.6	218.6	217.2	220.4	222.1	224.0	226.2	223.6	226.0	227.5
US2170	SYS	220.2	235.8	233.7	228.9	226.2	221.6	219.3	216.5	212.6	212.5
US2200	ATL	224.7	251.2	264.9	270.4	266.0	268.1	269.1	269.2	270.1	264.1
US2200	SYS	284.5	297.5	300.5	294.9	290.3	287.8	282.7	281.7	279.3	276.7

Table 5: System and ATLAS DGEMM comparison across platforms (MFLOPS)

4.4 Preliminary Results with GEMM-Based Level 3 BLAS

We present here some preliminary results for the Superscalar GEMM-based level 3 BLAS where ATLAS supplies the optimized GEMM. This package was developed by Bo Kagstrom and Per Ling. We have not had the time to do a exhaustive set of timings for these codes as yet. During the installation of ATLAS on the several platforms detailed in this paper, we did some preliminary timings of this package to determine whether it would supply an adequate level 3 BLAS. We reproduce these preliminary results here.

We used the BLAS timer from netlib, and report only the 500x500 result for each Level 3 BLAS call, always taking the 'Notranspose', 'Notranspose', 'Left', 'Upper' variant of each routine. These results are shown in table 6. For some of the systems, we timed the reference Fortran77 BLAS available from netlib as well.

SYSTEM	LIB	DGEMM	DSYMM	DSYRK	DSYR2K	DTRMM	DTRSM
AS255	ATLAS	159.2	159.2	128.6	155.7	152.3	150.3
AS255	SYS	140.4	141.1	68.6	68.7	127.4	127.1
AS600	ATLAS	315.1	298.2	245.8	317.8	291.7	287.8
AS600	SYS	289.4	294.8	202.3	192.7	266.8	269.6
AS600	F77	85.5	124.7	83.9	101.6	86.2	38.8
HPPA8K	ATLAS	594.8	520.6	480.4	489.9	446.2	480.5
HPPA8K	SYS	543.1	520.5	403.0	531.6	430.8	430.8
HPPA8K	F77	38.7	89.2	49.2	61.3	51.2	45.3
POWER2	ATLAS	446.4	446.4	446.4	438.6	431.0	446.4
POWER2	SYS	471.7	471.7	446.4	431.0	446.4	463.0
POWER2	F77	153.4	189.4	131.6	130.9	133.0	96.9
PPRO	ATLAS	146.2	146.2	100.8	141.2	101.6	120.2
PPRO	SYS	145.4	142.9	126.3	150.6	120.2	120.2
PPRO	F77	18.3	34.8	18.6	21.8	18.0	19.3
PII300	ATLAS	206.6	200.0	147.1	193.8	178.6	173.6
PII300	SYS	186.6	181.2	156.3	192.3	158.2	158.2
PII300	F77	21.6	42.0	22.0	26.7	22.0	21.8
R5000	ATLAS	106.3	104.8	82.4	100.2	95.1	95.0
R5000	SYS	66.4	63.8	87.8	62.2	63.3	62.8
R5000	F77	14.2	29.1	14.8	14.9	15.0	14.3
R8Kip21	ATLAS	291.3	281.9	226.2	282.0	264.2	259.1
R8Kip21	SYS	294.6	296.1	245.2	284.3	270.7	258.7
R10Kip27	ATLAS	319.4	312.8	288.6	314.7	301.0	295.9
R10Kip27	SYS	288.4	303.3	286.4	263.2	295.1	259.6
R10Kip27	F77	98.8	151.3	99.0	121.6	102.3	99.1
US2170	ATLAS	217.5	215.6	214.4	212.6	221.5	222.5
US2170	SYS	232.0	178.5	216.4	231.1	105.7	222.6
US2200	ATLAS	256.2	247.4	257.5	252.0	262.5	261.2
US2200	SYS	276.8	203.6	253.4	272.1	122.9	255.6

Table 6: 500x500 System and GEMM-based BLAS comparison across platforms (MFLOPS)

4.5 LU Timings

In order to demonstrate that these routines provide good performance in practice, we timed LAPACK's LU factorization. For each platform, we show three LU results:

1. LU factorization time linking to ATLAS DGEMM, Superscalar Level 3 GEMM-based BLAS, and reference Fortran77 Level 1 and 2 BLAS (all free software)

2. LU factorization time linking to vendor supplied BLAS

3. LU factorization time linking only to reference Fortran77 BLAS

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 7 shows the LU performance on several platforms. The LIB column is overloaded to convey both the BLAS used (A for ATLAS/Superscaler, S for system, F for Fortran77), 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(28)	77.5	106.1	122.0	126.2	125.7	127.1	127.1	127.0	129.2	129.4
AS255	S(32)	60.6	83.4	97.4	102.4	102.1	104.4	105.0	105.7	106.0	106.5
AS255	F(61)	48.8	43.0	42.9	39.6	38.8	38.8	37.8	37.0	36.3	35.8
AS600	A(56)	125.4	174.5	206.7	223.0	241.9	244.7	253.0	236.3	236.8	224.0
AS600	S(32)	110.5	157.8	188.2	202.9	220.1	224.9	233.6	220.5	221.0	209.9
AS600	F(32)	82.1	94.0	100.2	103.1	104.3	104.2	101.8	95.2	91.2	86.4
DCG533	A(28)	159.2	223.0	278.4	300.3	316.0	326.2	338.4	325.1	346.3	345.1
DCG533	F(16)	67.9	67.1	67.6	62.8	64.3	63.2	61.8	59.1	60.0	57.2
HPPA8K	A(20)	245.1	371.0	428.6	474.1	520.8	464.5	466.7	449.1	462.9	459.8
HPPA8K	S(52)	208.3	328.2	391.3	426.7	463.0	436.4	423.5	411.2	405.0	409.0
HPPA8K	F(20)	115.7	149.7	153.6	170.7	177.3	161.8	155.6	147.8	145.5	141.5
POWERPC	A(36)	40.8	51.7	56.6	60.9	63.6	66.0	67.8	69.3	72.0	72.1
POWERPC	S(34)	38.9	45.6	50.3	50.2	51.4	52.2	54.2	54.6	55.5	55.9
POWER2	A(32)	213.7	224.6	214.3	224.6	208.3	208.7	219.9	227.6	231.4	246.9
POWER2	S(32)	213.7	230.7	230.8	213.3	208.3	211.8	217.8	227.6	239.4	244.2
POWER2	F(25)	136.6	147.1	152.5	152.4	143.7	141.1	136.9	133.3	133.8	128.9
R5000	A(32)	54.0	68.4	72.7	73.4	77.3	78.9	80.8	80.9	83.4	84.1
R5000	S(30)	67.4	70.8	71.7	68.4	68.6	70.8	70.2	70.5	72.8	69.6
R5000	F(8)	30.4	31.1	29.9	28.4	27.6	26.0	24.4	24.1	23.6	22.4
R8Kip21	A(40)	101.6	164.6	194.8	219.0	231.3	241.1	240.3	233.1	223.2	212.9
R8Kip21	S(40)	107.0	169.2	205.5	227.7	243.1	252.4	258.2	248.8	237.3	221.8
R8Kip21	F(40)	58.1	75.7	84.5	98.8	93.3	95.8	97.2	96.4	94.8	92.6
R10Kip27	A(28)	156.6	208.4	236.5	245.4	256.1	255.1	259.9	254.1	249.7	242.1
R10Kip27	S(41)	154.0	198.1	224.0	235.9	247.7	251.5	256.1	253.1	253.2	244.5
PPRO	A(32)	55.6	80.0	90.4	94.3	97.8	101.7	104.0	106.2	108.9	110.9
PPRO	S(32)	63.6	90.7	101.4	109.4	110.5	113.7	115.8	120.6	122.2	123.5
PPRO	F(30)	31.0	34.1	34.7	34.3	34.2	33.2	33.2	32.3	31.3	30.6
PII300	A(30)	82.0	111.9	122.6	124.9	130.7	134.8	137.2	140.3	143.7	145.4
PII300	S(32)	93.1	128.0	137.7	143.5	145.5	138.4	150.9	156.2	158.3	160.8
PII300	F(25)	42.4	43.6	43.7	43.0	42.0	40.9	40.9	40.0	38.1	37.3
US2170	A(44)	103.2	144.7	157.9	161.9	164.2	168.9	172.9	167.8	176.6	178.8
US2170	S(32)	128.0	174.5	178.3	177.2	175.6	175.9	181.9	172.3	186.0	185.3
US2170	F(48)	74.0	85.0	86.0	84.7	83.6	82.2	82.6	78.0	81.1	80.3
US2200	A(44)	123.8	173.1	195.5	202.0	195.2	205.8	208.5	204.4	210.4	208.8
US2200	S(32)	142.3	207.7	233.0	227.6	217.5	219.1	218.6	218.6	225.1	22.0
US2200	F(48)	88.2	101.0	105.3	101.8	100.5	99.4	98.6	99.3	98.5	97.9

Table 7: Double precision LU timings on 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 Pentiums 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 experiment with them in a 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/atlas`. Installation instructions are provided in the supplied `README` file.

7 Future Work

The ATLAS package presently available on netlib contains the real version of matrix-matrix multiplication. Matrix multiplication is the building block of all of the level 3 BLAS, however. Initial research using the publicly available level 3 gemm-based BLAS [7] suggests that this provides a perfectly acceptable level 3 BLAS.

As time allows, we can avoid some of the $O(N^2)$ costs associated with using the gemm-based BLAS by supporting the level 3 BLAS directly in ATLAS. We also plan on providing the software for complex data types.

We have preliminary results for the most important level 2 BLAS routine (matrix-vector multiply) as well. This is of particular importance, because matrix vector operations, which have $O(N^2)$ operations and $O(N^2)$ data, demand a significantly different code generation approach than that required for matrix-matrix operations, where the data is $O(N^2)$, but the operation count is $O(N^3)$. Initial results suggest that ATLAS will achieve comparable success with optimizing the level 2 BLAS as has been achieved for level 3 (this means that our timings compared to the vendor will be comparable; obviously, unless your architecture supports many pipes to memory, a level 2 operation will not be as efficient as the corresponding level 3 operation).

Another avenue of research involves sparse algorithms. The fundamental building block of iterative methods is the sparse matrix times dense vector multiply. This work should leverage the present research (in particular, make use of the dense matrix-vector multiply). The present work uses compile-time adaptation of software. Since matrix vector multiply may be called literally thousands of times during the course of an iterative method, we plan to investigate run-time adaptation as well. These run-time adaptations could include matrix dependent transformations [8], as well as specific code generation.

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 optimal 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.

A BLAS and compiler details

This section exists to provide further details regarding the compilers and BLAS used in our timings. Table 8 shows the system BLAS that were used for the timings. Tables 9 and 10 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
HP9K/735	-lvec	Revision 73.14
POWER2	-lesslp2	essl 2.2.2.4
POWERPC	-lessl	essl 2.2.2.2
R4600	-lblas	Standard Execution Environment (Fortran 77, 4.0.2)
R5000	-lblas	Standard Execution Environment (Fortran 77, 7.1)
R8Kip21	-lblas	Standard Execution Environment (Fortran 77, 7.2)
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
US2170	-xlic_lib=sunperf	Sun Performance Library 1.2
US2200	-xlic_lib=sunperf	Sun Performance Library 1.2

Table 8: 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	gcc 2.7.2.3, DEC C V5.2-033 on Digital UNIX V4.0 (Rev. 564)
DCG533	gcc version 2.7.2.3, Red Hat Linux 5.0
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	
P5MMX	gcc version 2.7.2.1, Red Hat Linux 5.0
PPRO	gcc version 2.7.2.3, Red Hat Linux 5.0
PII300	gcc version 2.7.2.3, Red Hat Linux 5.0
R4600	Base Compiler Development Environment, 5.3
R5000	Base Compiler Development Environment, 7.0
R8Kip21	MIPSpro Compilers: Version 7.20
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
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 9: Compiler and version

System	Compiler & flags
AS255	`cc -arch host -tune host -std -assume aligned_objects -O5`
AS600	`-O1 -fschedule-insns -fschedule-insns2 -fno-expensive-optimizations`
DCG533	`-O1 -fschedule-insns -fschedule-insns2 -fno-expensive-optimizations`
HP9K/735	`-Aa +O2`
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`
P5MMX	`gcc -fomit-frame-pointer -O`
PPRO	`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`
R8Kip21	`cc -64 -mips4 -r8000 -OPT:Olimit=15000 -TARG:platform=ip21 -LOPT:alias=typed -LNO:blocking=OFF -O2`
R10Kip27	`cc -64 -mips4 -r10000 -OPT:Olimit=15000 -TARG:platform=ip27 -LOPT:alias=typed -LNO:blocking=OFF -O3`
R10Kip28	`cc -64 -mips4 -r10000 -OPT:Olimit=15000 -TARG:platform=ip28 -LOPT:alias=typed -LNO:blocking=OFF -O3`
MS70	`cc -xchip=micro2 -xarch=v8 -dalign -fsingle -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` double precision only, `arch=v8plusa`

Table 10: Compiler flags