User contribution to ATLAS

R. Clint Whaley *

August 25, 2017

Abstract

This paper describes the method by which users can speed up ATLAS for themselves, as well as contribute any such speedup to the ATLAS project. It's written to get you started, in a highly informal (read sloppy) fashion. There's a lot of material that optimally should be covered in detail, which is only hinted at in this document. Since no real attempt has been made to make the document sheerly backward referential, it is recommended that the user at least skim the entire section before attempting to understand and/or apply information from a given subsection.

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

Contents

1 Introduction

This paper describes the method by which users can speed up ATLAS (Automatically Tuned Linear Algebra Software) for themselves, as well as contribute any such speedup to the ATLAS project.

ATLAS is an implementation of a new style of high performance software production/maintenance called Automated Empirical Optimization of Software (AEOS). In an AEOS-enabled library, many different ways of performing a given kernel operation are supplied, and timers are used to empirically determine which implementation is best for a given architectural platform. ATLAS uses two techniques for supplying different implementations of kernel operations: *multiple implementation* and *code generation*.

In code generation, a highly-parameterized program is written that can generate many different kernel implementations. The matrix multiply code generator is an example of this. The second method is multiple implementation, and this, as its name suggests, is simply supplying various hand-written implementations of the same kernel.

ATLAS provides a standard way for users to help with multiple implementation. ATLAS is designed such that several kernel routines supply performance for the entire library. The entire Level 3 BLAS may be speeded up by improving the one simple kernel, which we will refer to as *gemmK*, to distinguish it from a full GEMM routine. The Level 2 routines make similarly be sped up by providing GER and GEMV kernels (there are several of these, as discussed later). ATLAS has standard timers which can call user-programmed versions of these kernels, and automatically use them throughout the library when they are superior to the ATLAS-produced versions.

One thing to consider when getting started is to take the best ATLAS kernel found, and, for instance, add some prefetch instructions, and see if you can get noticeable improvements.

1.1 Contributing your improvement to ATLAS

If you produce a substantially better kernel than ATLAS presently has, we hope that you will contribute it to the ATLAS project. Code submissions and discussions of this type should take place on the `math-atlas-devel@lists.sourceforge.net` mailing list. Anyone can sign up for this list, and we hope that interested parties will correspond using it. In this way someone interested in supplying a particular enhancement to ATLAS can find if someone else is already working on it, find interested collaborators, get coding help from a wider pool than with the standard atlas mailing list, etc.

1.1.1 Signing up for the ATLAS mailing list

We have set up a mailing list that we hope contributers and interested parties will subscribe to. It's e-mail address is `math-atlas-devel@lists.sourceforge.net`. For details on signing up, see `http://math-atlas.sourceforge.net/faq.html#lists`.

1.1.2 License issues

ATLAS uses a BSD-style license for its distribution, and thus any code that you wish to contribute must have a compatible license. This effectively rules out GPL and even LGPL licenses, since if the ATLAS group redistributed any routines with these licenses, the

conditions of their licenses would effectively change the entire ATLAS library to GPL or LGPL.

Note that you, the author, retain the copyright, and thus you can relicense your code in any way you wish. So, anything released as part of the ATLAS tarfile will need a BSD-compatible license, but you could then issue the code yourself under a different license. Alternatively, you can provide your codes as "patches" to standard ATLAS, and thus avoid the license issue altogether, at the cost of not being part of the standard tarfile. Even if you decide not to license your contribution in a way that the ATLAS group can redistribute, submitting it to the mailing list is still recommended. Other people can see your work, and we can point to it if we think it is cool enough.

1.1.3 Due Credit

We have set in place some infrastructure in order to give contributers credit for their work. Firstly, if your code is included in the standard tarfile, your contribution will be noted in `ATLAS/doc/AtlasCredits.txt`. As the author, you retain the copyright, and thus your name is in the contributed code. We have also modified the installation logger to print the author name for user-provided routines (the exception is the Level 1 kernels, which are simply too numerous for logging). We will make a good faith effort to give due credit for work, but in general, we can't guarantee anything.

As an aside, if fame and fortune are your major motivations, you may want to consider contributing to something else anyway. As the founder of the ATLAS project, I'm still waiting for my first interview with CNN, NBC, etc., or indeed, anyone other then my relatives (and all those interviews go like, "You're in computers, can't you get those wordperfect guys to make it easier to use?").

1.1.4 Inclusion in the ATLAS tarfile

If you submit an improvement to `math-atlas-devel@lists.sourceforge.net`, we hope that other ATLAS users will try it out, and give comments. Ultimately, someone in the ATLAS group (currently, that would be me) will decide whether to include it in the standard tarfile or not. Such a decision will be based on how hard it is to get to work with our distribution (this pain should be minimal if the instructions in this note are followed), how much of an improvement it is, etc. Obviously, there are speed improvements that some users may utilize that the ATLAS group can't indulge in (examples include speedups that possibly lose accuracy, such as, for instance, using Strassen's algorithm for matrix multiply). If there appears to be a great demand for such lossy optimizations, we may consider ways of allowing users to select if they want to risk them.

Note that even if we are ungrateful morons who do not release your submission, you, and anyone who wants to use your kernel routines, can use the ATLAS infrastructure to get a complete, fast BLAS implementation.

1.2 Directory terminology

Using ATLAS's configure, you can build the library in any directory. We will call the path to your top-level directory where you choose to build ATLAS `BLDdir`. The ATLAS

tarfile produces the source directory, which we will indicate by either `SRCdir` or simply `ATLAS/.....`

1.3 Coding conventions for contributed code

The following rules are mandatory:

- All externally addressable symbols used in the code (eg, routine names, global variables, etc) will be prefaced by the prefix `ATLU_`.
- All user-contributed include files will be found in `ATLAS/include/contrib`.

The ATLAS team encourages:

- Using the standard BLAS names for operands (i.e., for GEMM, using the names `A`, `B`, and `C` for the input matrices, for GEMV, using the names `A`, `X`, and `Y`, etc.).

1.4 A note about ATLAS kernels

All of ATLAS's user-suppliable kernels are used to speed up a wide range of codes (i.e., GEMM speeds up all level 3 BLAS, etc), which means it is possible to write a good GEMM, for instance, that is still not a good GEMM *kernel*. The unmodified testers and timers described in this note time these kernels in their most-used states, so if you develop a kernel using these techniques, everything will likely be OK. However, if you first write a full-blown GEMV, for instance, and then attempt to adapt it, there is more opportunity for mismatch. At the end of each kernel section I give a few kernel notes to give you an idea of how ATLAS uses the kernel.

2 Speeding up the Level 3 BLAS

The performance kernel for the entire Level 3 BLAS is matrix multiply. Matrix multiply is written in terms of a lower-level building block that we call *gemmK*. *gemmK* is a special matrix multiply where the input dimensions are fixed at $M = N = K = N_B$, where the blocking factor N_B is chosen in order to maximize L1 cache reuse, for a loose enough definition of L1 cache (typically, we use it to mean the first level of cache accessible by the FPU, which may be the L2 cache on some systems).

ATLAS actually has two different classes of GEMM kernels: one for copied matrices (*gemmK*), and one that operates directly on the user's matrices without a copy. For matrices of sufficient size, ATLAS copies the input matrix into *block-major* storage. In block-major storage, the $N_B \times N_B$ blocks operated on by the *gemmK* are actually contiguous. This optimization prevents unnecessary cache misses, cache conflicts, and TLB problems. However, for sufficiently small matrices, the cost of this data copy is prohibitively expensive, and thus ATLAS has kernels that operate on non-copied data. However, without the copy to simplify the process, there are multiple non-copy kernels (differing kernels for differing transpose settings, for instance). Since the non-copy kernels are typically only used for very small problems, and they are much more complex, ATLAS presently accepts contributed code only for the copy matmul kernel. For most problems, well over 98% of ATLAS time is spent in the copy matmul kernel, so this should not be much of a problem.

2.1 Building the General Matrix Multiply From *gemmK*

This section describes the code necessary to build the BLAS's general matrix-matrix multiply using an L1 cache-contained matmul (hereafter referred to as *gemmK*).

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

When the user calls GEMM, ATLAS must decide whether the problem is large enough to tolerate copying the input matrices A and B. If the matrices are large enough to support this $O(N^2)$ overhead, ATLAS will copy A and B into block-major format. ATLAS's block-major format breaks up the input matrices into contiguous blocks of a fixed size N_B, where N_B is chosen in order to maximize L1 cache reuse. Once in block-major format, the blocks are contiguous, which eliminates TLB problems, minimizes cache thrashing and maximizes cache line use. It also allows ATLAS to apply alpha (if alpha is not already one) to the smaller of A or B, thus minimizing this cost as well. Finally, the package can use the copy to transform the problem to a particular transpose setting, which for load and indexing optimization, is set so A is copied to transposed form, and B is in normal (non-transposed) form. This means our L1-cache contained code is of the form $C \leftarrow A^T B$, $C \leftarrow A^T B + C$, and $C \leftarrow A^T B + \beta C$, *where all dimensions, including the non-contiguous stride, are known to be* N_B. Knowing all of the dimensions of the loops allows for arbitrary unrollings (i.e., if the instruction cache could support it, ATLAS could unroll all loops completely, so that the L1 cache-contained multiply had no loops at all). Further, when the code generator knows leading dimension of the matrices (i.e., the row stride), all indexing can be done up front, without the need for expensive integer or pointer computations.

If the matrices are too small, the $O(N^2)$ data copy cost can actually dominate the algorithm cost, even though the computation cost is $O(N^3)$. For these matrices, ATLAS will call an gemm kernel which operates on non-copied matrices (i.e. directly on the user's operands). The non-copy matmul kernels will generally not be as efficient as the even the generated copy *gemmK*; at this problem size the main drawback is the additional pointer arithmetic required in order to support the user-supplied leading dimension and its affect on the cost of the memory load (which varies according to transpose settings, as well as architectural features).

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

Figure 1 shows the necessary steps for computing a $N_B \times N_B$ section of C using *gemmK*.

More formally, 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 *gemmK* of the correct form based on user-defined β (eg. if $\beta == 0$, use $C \leftarrow AB$)

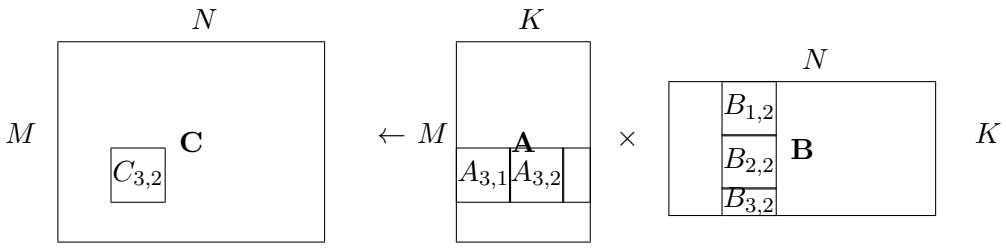

Figure 1: One step of matrix-matrix multiply

to multiply block 0 of the row panel i of A with block 0 of the column panel j of B.

2. Call *gemmK* 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, $\forall k, 1 \leq k < \lceil K/N_B \rceil$.

As this example demonstrates, if a given dimension is not a multiple of the L1 blocking factor N_B, partial blocks results. ATLAS has special routines that handle cases where one or more dimension is less than N_B; these routines are referred to as *cleanup* codes.

2.2 The Main GEMM Kernel, *gemmK*

So, there are actually three *gemmK* kernels (corresponding to different β values), and perform the operations: $C \leftarrow A^T B$, $C \leftarrow A^T B + C$, $C \leftarrow A^T B + \beta C$. All input arrays (A, B, C) are column-major (they are still used as performance kernels for row-major BLAS as well, so don't worry). Additionally, A^T and B are in block-major format, such that $lda = ldb = M = N = K = N_B$.

2.2.1 *gemmK* macro definitions

In order to make writing a *gemmK* easier, ATLAS defines several cpp macros for programmer use. Examples in subsequent sections should illustrate the use of these macros, so we merely define them here.

First, ATLAS defines the macro `ATL_USERMM` to the appropriate ATLAS internal kernel name. Second, it defines one of `SREAL`, `DREAL`, `SCPLX`, `DCPLX`, according to the data type being compiled (single precision real, double precision real, single precision complex, double precision complex, respectively).

Similarly, ATLAS defines a macro indicating the β case being compiled, `BETA1` (β should be assumed to be 1.0), `BETA0` (β should be assumed to be 0.0), and `BETAX` (β neither zero or one, and should be handled as an input parameter).

Finally, the fixed blocking factors for each dimension are defined `MB`, `NB`, `KB`. Note that for our *gemmK*, `MB` = `NB` = `KB` = N_B; they are separated out for support of the cleanup codes, where they can be different. ATLAS also defines the macros `MB2`, `NB2`, `KB2`, which are simply two times the appropriate blocking factor.

2.2.2 *gemmK* API

The *gemmK* API may be summarized as:

```
#if defined(SREAL) || defined(SCPLX)
```

void ATL_USERMM (const int M, const int N, const int K, const float alpha, const float *A, const int lda, const float *B, const int ldb, const float beta, float *C, const int ldc)

```
#elif defined(DREAL) || defined(DCPLX)
```

void ATL_USERMM (const int M, const int N, const int K, const double alpha, const double *A, const int lda, const double *B, const int ldb, const double beta, double *C, const int ldc)

```
#endif
```

2.2.3 *gemmK* description file

In the install process, ATLAS first searches through the *gemmK* implementations provided by the ATLAS matmul generator. When the best generated code is found, the user contributed codes are timed to see if they can beat the generated code. The `gemmK` multiple implementation search script opens a description file for each precision (`scases.dsc`, `dcases.dsc`, `ccases.dsc`, `zcases.dsc`) in the `BLDdir/tune/blas/gemm/` directory, to see what user-contributed codes are available. This master index file is actually generated based on several user-supplied files from `ATLAS/tune/blas/gemm/CASES` (see Section 2.2.4 for the names and definitions of these files). The format for all these files is the same, and is described in the following paragraphs.

The first line of each file is a comment line, and is ignored. The next line indicates the number of user-contributed codes to search, and each subsequent line supplies information about a given user-supplied *gemmK*. The form of these lines is:

`<ID> <flag> <mb> <nb> <kb> <muladd> <lat> <mu> <nu> <ku> <rout> "<author>"`

`<rout>` and `<author>"` are strings, and the rest of the parameters are signed integers. The meaning of these parameters are:

- `ID`: Strictly positive integer which uniquely identifies this descriptor line. ID must by unique only within a precision.
- `<flag>`: flag indicating special conditions. See table below.
- `<mb>`, `<nb>`, `<kb>`: Used to indicate restriction on the input parameter M (N, K resp.), and its associated blocking `MB` (`NB`, `KB`, resp.). If the value is zero, the internal routine handles any M; i.e. the loop-limit is a runtime variable. If the value is negative, then M = `MB` = `-<mb>` (i.e., the blocking factor cannot be varied using a macro). If the value is positive, the blocking factor can be varied by setting the appropriate macro (`MB` `NB`, `KB`, resp.), but the blocking factor must be a multiple of the value. Therefore, setting `<mb>` = 4, indicates that `MB` must be a multiple of 4, while setting it to 1 indicates that `MB` is an arbitrary compile-time constant.

- `<muladd>`: Set to zero if you are using separate multiply and add instructions, 1 otherwise. If you don't know the answer, put 1.
- `<lat>`: Set to the latency you use between floating point instructions. If you don't know the answer, put 1.
- `<mu>`: Unrolling you are using for the M loop.
- `<nu>`: Unrolling you are using for the N loop.
- `<ku>`: Unrolling you are using for the K loop.
- `<rout>`: The filename of the user-contributed routine, relative to the path `ATLAS/tune/blas/gemm/CASES`. Maximum length 64 chars.
- `<author>`: The name of the author or authors, enclosed in quotes. Maximum length 64 chars.

Table 1 summarizes the presently defined `flag` values.

FLAG	MEANING
0	Normal
8	Do not consider this kernel for cleanup
16	Consider this kernel for cleanup *only*
32	lda and ldb are not restricted to `KB`
64	`mb` provides run-time constraint, not compile-time
128	`nb` provides run-time constraint, not compile-time
256	`kb` provides run-time constraint, not compile-time
512	This kernel needs $4N_b \leq cacheelts$

Table 1: Matmul index routine flag variables

Here's an example:

```
<ID> <flag> <mb> <nb> <kb> <muladd> <lat> <mu> <nu> <ku> <rout> "<Contributer>"
3
 1 0 0 0 0 1 1 1 1 1 ATL_mm1x1x1.c "R. Clint Whaley"
 2 0 1 1 1 1 1 1 1 1 ATL_mm1x1x1b.c "R. Clint Whaley"
 3 0 1 1 8 1 1 1 1 4 ATL_mm2.c "R. Clint Whaley"
```

So, we have 3 user-supplied routines, all written by me. The first loops over M, N, and K, but the following two routines loop over the cpp macros `MB`, `NB`, `KB`. The third routine insists that `KB` be a multiple of 8. The first two routines don't unroll any of the loops, while the third unrolls the K loop to a depth of 4. They all use a combined muladd style of programming, and don't worry about latency.

2.2.4 Index filenames

As previously mentioned, ATLAS builds a system and type dependent index file from user-supplied files in `ATLAS/tune/blas/gemm/CASES`. This is done so that the all routines do not need to be run on all machines (i.e., no need to waste time trying to run SSE-enabled assembly routines when on a Dec ev56). Here is a list of description files presently queried by ATLAS when building the full search index:

1. **[s,d,c,z]cases.0**: Any user-contributed kernel which is system independent (i.e. doesn't require a particular compiler, etc)
   - Convention is to choose IDs in range: $0 < ID < 100$.
2. **[s,d,c,z]cases.flg**: Any user-contributed kernel requiring specific compiler and/or flags
   - Convention is to choose IDs in range: $300 \leq ID < 400$.
3. **[s,c]cases.3DN**: Kernels requiring 3DNow! to run.
   - Convention is to choose IDs in range: $100 \leq ID < 200$.
4. **[s,c]cases.SSE**: Kernels requiring SSE1 to run.
   - Convention is to choose IDs in range: $200 \leq ID < 300$.

2.3 Putting it together with some examples

Let's say we decide to cover the basics, the classical 3 do loop implementation of matmul would be:

```
void ATL_USERMM
   (const int M, const int N, const int K, const double alpha,
    const double *A, const int lda, const double *B, const int ldb,
    const double beta, double *C, const int ldc)
{
   int i, j, k;
   register double c00;

   for (j=0; j < N; j++)
   {
      for (i=0; i < M; i++)
      {
         #ifdef BETA0
            c00 = 0.0;
         #elif defined(BETA1)
            c00 = C[i+j*ldc];
         #else
            c00 = C[i+j*ldc] * beta;
         #endif
```

```
         for (k=0; k < K; k++) c00 += A[k+i*lda] * B[k+j*ldb];
         C[i+j*ldc] = c00;
      }
   }
}
```

We then save this paragon of performance to `ATLAS/tune/blas/gemm/CASES/ATL_mm1x1x1.c`. From `BLDdir/tune/blas/gemm/`, we can test that it gets the right answer by:

```
make mmutstcase pre=d nb=40 mmrout=CASES/ATL_mm1x1x1.c beta=0
make mmutstcase pre=d nb=40 mmrout=CASES/ATL_mm1x1x1.c beta=1
make mmutstcase pre=d nb=40 mmrout=CASES/ATL_mm1x1x1.c beta=7
```

We pass four arguments to mmutstcase, a precision specifier (`d` : double precision real; `s` : single precision real; `z` : double precision complex; `c` : single precision complex), the size of the blocking parameter N_B, the beta value to test (0, 1, and other), and finally, the filename to test.

If these messages pass the test, we can then see what kind of performance we get by (this is the actual output on my 266Mhz PII):

```
make ummcase pre=d nb=40 mmrout=CASES/ATL_mm1x1x1.c beta=1
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=1.820000, mflop=53.731868
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=1.810000, mflop=54.028729
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=1.830000, mflop=53.438251
```

This is the same timing repeated three times (this just tries to ensure timings are repeatable), and the only output of real interest is the MFLOP rate at the end. The values the timer prints (mu, nu, ku, lat) are all defaults because we didn't specify them; specifying them has no effect when the timer is used in this way, so don't worry about them.

Now we can trivially improve the implementation by using the macro constants in order to let the compiler unroll the loops:

```
void ATL_USERMM
   (const int M, const int N, const int K, const double alpha,
    const double *A, const int lda, const double *B, const int ldb,
    const double beta, double *C, const int ldc)
{
   int i, j, k;
   register double c00;

   for (j=0; j < NB; j++)
   {
      for (i=0; i < MB; i++)
      {
         #ifdef BETA0
            c00 = 0.0;
         #elif defined(BETA1)
            c00 = C[i+j*ldc];
         #else
```

```
            c00 = C[i+j*ldc] * beta;
         #endif
         for (k=0; k < KB; k++) c00 += A[k+i*KB] * B[k+j*KB];
         C[i+j*ldc] = c00;
      }
   }
}
```

We save this to `ATL_mm1x1x1b.c`, and then time:

```
make ummcase pre=d nb=40 mmrout=CASES/ATL_mm1x1x1b.c beta=1
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=1.670000, mflop=58.558084
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=1.660000, mflop=58.910843
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=1.670000, mflop=58.558084
```

OK, maybe a little explicit loop unrolling will make things work better:

```
void ATL_USERMM
   (const int M, const int N, const int K, const double alpha, const double *A, const int lda
{
   int i, j, k;
   register double c00, c10, b0;
   const double *pA0, *pB=B;

#if ( (KB / 8)*8 != KB ) || (MB / 2)*2 != MB
   create syntax error!$@@&
#endif
   for (j=0; j < NB; j++, pB += KB)
   {
      pA0 = A;
      for (i=0; i < MB; i += 2, pA0 += KB2)
      {
         #ifdef BETA0
            c00 = c10 = 0.0;
         #elif defined(BETA1)
            c00 = C[i+j*ldc];
            c10 = C[i+1+j*ldc];
         #else
            c00 = beta*C[i+j*ldc];
            c10 = beta*C[i+1+j*ldc];
         #endif
         for (k=0; k < KB; k += 8)
         {
            b0 = pB[k];
            c00 += pA0[k] * b0;
            c10 += pA0[KB+k] * b0;
            b0 = pB[k+1];
            c00 += pA0[k+1] * b0;
```

```
            c10 += pA0[KB+k+1] * b0;
            b0 = pB[k+2];
            c00 += pA0[k+2] * b0;
            c10 += pA0[KB+k+2] * b0;
            b0   =  pB[k+3];
            c00 += pA0[k+3] * b0;
            c10 += pA0[KB+k+3] * b0;
            b0   =  pB[k+4];
            c00 += pA0[k+4] * b0;
            c10 += pA0[KB+k+4] * b0;
            b0   =  pB[k+5];
            c00 += pA0[k+5] * b0;
            c10 += pA0[KB+k+5] * b0;
            b0   =  pB[k+6];
            c00 += pA0[k+6] * b0;
            c10 += pA0[KB+k+6] * b0;
            b0   =  pB[k+7];
            c00 += pA0[k+7] * b0;
            c10 += pA0[KB+k+7] * b0;
         }
         C[i+j*ldc] = c00;
         C[i+1+j*ldc] = c10;
      }
   }
}
```

And with this ode to beauty and elegance we get (after checking that it still gets the right answer, of course):

```
make ummcase pre=d nb=40 mmrout=CASES/ATL_mm2x1x8.c beta=1
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=0.720000, mflop=135.822222
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=0.710000, mflop=137.735211
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=0.710000, mflop=137.735211
```

Its interesting to see the effects of differing β on the code:

```
make ummcase pre=d nb=40 mmrout=CASES/ATL_mm2x1x8a.c beta=0
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=0.700000, mflop=139.702857
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=0.700000, mflop=139.702857
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=0.700000, mflop=139.702857

make ummcase pre=d nb=40 mmrout=CASES/ATL_mm2x1x8a.c beta=7
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=0.720000, mflop=135.822222
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=0.730000, mflop=133.961644
dNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=0.720000, mflop=135.822222
```

As well as differing blocking factors:

```
make ummcase pre=d mmrout=CASES/ATL_mm2x1x8a.c beta=1 nb=16
dNB=16, ldc=16, mu=4, nu=4, ku=1, lat=4: time=0.850000, mflop=115.112056
dNB=16, ldc=16, mu=4, nu=4, ku=1, lat=4: time=0.860000, mflop=113.773544
dNB=16, ldc=16, mu=4, nu=4, ku=1, lat=4: time=0.850000, mflop=115.112056

make ummcase pre=d mmrout=CASES/ATL_mm2x1x8a.c beta=1 nb=32
dNB=32, ldc=32, mu=4, nu=4, ku=1, lat=4: time=0.730000, mflop=134.034586
dNB=32, ldc=32, mu=4, nu=4, ku=1, lat=4: time=0.740000, mflop=132.223308
dNB=32, ldc=32, mu=4, nu=4, ku=1, lat=4: time=0.730000, mflop=134.034586

make ummcase pre=d mmrout=CASES/ATL_mm2x1x8a.c beta=1 nb=48
dNB=48, ldc=48, mu=4, nu=4, ku=1, lat=4: time=0.820000, mflop=119.223571
dNB=48, ldc=48, mu=4, nu=4, ku=1, lat=4: time=0.820000, mflop=119.223571
dNB=48, ldc=48, mu=4, nu=4, ku=1, lat=4: time=0.820000, mflop=119.223571
```

If we wanted to have ATLAS try these crappy implementations during the ATLAS search, we would have the following `ATLAS/tune/blas/gemm/CASES/dcases.dsc`:

```
<ID> <flag> <mb> <nb> <kb> <muladd> <lat> <mu> <nu> <ku> <rout> "<Contributer>"
3
1 0 0 0 0 1 1 1 1 1 ATL_mm1x1x1.c    "R. Clint Whaley"
2 0 1 1 1 1 1 1 1 1 ATL_mm1x1x1b.c   "R. Clint Whaley"
3 0 2 1 8 1 1 2 1 8 ATL_mm2x1x8a.c   "R. Clint Whaley"
```

2.4 More timing info

So maybe you wonder how our big hand-tuned guy stacks up against the ATLAS code generator? When ATLAS completed its search on my PII, it stored its best case in `ATLAS/tune/blas/gemm/LINUX_PII/res/dMMRES`:

```
speedy. cat res/dMMRES
MULADD  LAT  NB  MU  NU  KU  FFTCH  IFTCH  NFTCH    MFLOP
     0    5  40   2   1  40      0      2      1   197.94
16
```

We generate and time this case by:

```
make mmcase muladd=0 lat=5 nb=40 mu=2 nu=1 ku=40 beta=1
dNB=40, ldc=40, mu=2, nu=1, ku=40, lat=5: time=0.490000, mflop=199.575510
dNB=40, ldc=40, mu=2, nu=1, ku=40, lat=5: time=0.500000, mflop=195.584000
dNB=40, ldc=40, mu=2, nu=1, ku=40, lat=5: time=0.490000, mflop=199.575510
```

We test that the generator isn't out of its mind by:

```
make mmtstcase muladd=0 lat=5 nb=40 mu=2 nu=1 ku=40 beta=1
```

Note that when timing/testing the generator, varying the parameters such as mu, nu, ku, beta, etc., generates different codes, and thus different performance numbers:

```
make mmcase muladd=0 lat=4 nb=40 mu=2 nu=1 ku=4 beta=1
dNB=40, ldc=40, mu=2, nu=1, ku=4, lat=4: time=0.760000, mflop=128.673684
dNB=40, ldc=40, mu=2, nu=1, ku=4, lat=4: time=0.770000, mflop=127.002597
dNB=40, ldc=40, mu=2, nu=1, ku=4, lat=4: time=0.770000, mflop=127.002597

make mmcase muladd=0 lat=4 nb=40 mu=2 nu=1 ku=8 beta=1
dNB=40, ldc=40, mu=2, nu=1, ku=8, lat=4: time=0.640000, mflop=152.800000
dNB=40, ldc=40, mu=2, nu=1, ku=8, lat=4: time=0.630000, mflop=155.225397
dNB=40, ldc=40, mu=2, nu=1, ku=8, lat=4: time=0.630000, mflop=155.225397

make mmcase muladd=0 lat=4 nb=40 mu=2 nu=2 ku=40 beta=1
dNB=40, ldc=40, mu=2, nu=2, ku=40, lat=4: time=2.550000, mflop=38.349804
dNB=40, ldc=40, mu=2, nu=2, ku=40, lat=4: time=2.560000, mflop=38.200000
dNB=40, ldc=40, mu=2, nu=2, ku=40, lat=4: time=2.550000, mflop=38.349804
```

2.5 Complex *gemmK*

Vainly hoping we were done, eh? Nope, complex codes are special. ATLAS actually uses the real matrix multiply generator in order to do complex multiplication. It needs some tricks to do this, obviously. The first thing to note is that if X_r denotes the real elements of X, and X_i indicates the imaginary components, then complex matrix-matrix multiply of the form $C \leftarrow AB + \beta C$ may be accomplished by the following four real matrix multiplies:

1. $C_r \leftarrow A_i B_i - \beta C_r$
2. $C_i \leftarrow A_i B_r + \beta C_i$
3. $C_r \leftarrow A_r B_r - C_r$
4. $C_i \leftarrow A_r B_i + C_i$

This works fine assuming β is real, otherwise β must be applied explicitly as a complex scalar, and then set $\beta = 1$ in the above outline.

Therefore, in order to use this trick, upon copying A and B to block-major storage, ATLAS also splits the arrays into real and imaginary components. The only matrix not expressed as two real matrices is then C, and to fix this problem, ATLAS demands that the complex *gemmK* stride C by 2. An example will solidify the confusion.

A simple 3-loop implementation of an ATLAS complex *gemmK* is:

```
void ATL_USERMM
   (const int M, const int N, const int K, const double alpha,
    const double *A, const int lda, const double *B, const int ldb,
    const double beta, double *C, const int ldc)
{
   int i, j, k;
   register double c00;

   for (j=0; j < N; j++)
   {
```

```
      for (i=0; i < M; i++)
      {
         #ifdef BETA0
            c00 = 0.0;
         #else
            c00 = C[2*(i+j*ldc)];
            #ifdef BETAX
               c00 *= beta;
            #endif
         #endif
         for (k=0; k < K; k++) c00 += A[k+i*lda] * B[k+j*ldb];
         C[2*(i+j*ldc)] = c00;
      }
   }
}
```

First we test that it produces the right answer:

```
make cmmutstcase pre=z nb=40 mmrout=CASES/ATL_cmm1x1x1.c beta=0
make cmmutstcase pre=z nb=40 mmrout=CASES/ATL_cmm1x1x1.c beta=1
make cmmutstcase pre=z nb=40 mmrout=CASES/ATL_cmm1x1x1.c beta=8
```

Then we scope the awesome performance:

```
make cmmucase pre=z nb=40 mmrout=CASES/ATL_cmm1x1x1.c beta=1
zNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=1.830000, mflop=53.438251
zNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=1.830000, mflop=53.438251
zNB=40, ldc=40, mu=4, nu=4, ku=1, lat=4: time=1.830000, mflop=53.438251
```

Now, since we are clearly gluttons for punishment, we compare our masterwork to ATLAS's generated kernel:

```
speedy. cat res/zMMRES
MULADD  LAT  NB  MU  NU  KU  FFTCH  IFTCH  NFTCH    MFLOP
     0    5  36   2   1  36      0      2      1   186.42
     10

speedy. make mmcase muladd=0 lat=5 nb=36 mu=2 nu=1 ku=36 beta=1
dNB=36, ldc=36, mu=2, nu=1, ku=36, lat=5: time=0.500000, mflop=195.581952
dNB=36, ldc=36, mu=2, nu=1, ku=36, lat=5: time=0.490000, mflop=199.573420
dNB=36, ldc=36, mu=2, nu=1, ku=36, lat=5: time=0.490000, mflop=199.573420
```

2.6 What to do if you are writing in assembler

If your kernel is written in gas assembler, you can tell the tester and timer that by setting the appropriate compiler and flag macro on the command line. For single precision types, these macros for *gemmK* are called `SMC` and `SMCFLAGS`, respectively, and they are `DMC` and `DMCFLAGS` for double precision routines. For instance, to test a `DGEMM` code written in assembler, requiring a 16 blocking factor, you'd issue:

```
make ummcase pre=d mmrout=CASES/myassembler.c nb=16 \
     DMC=gcc DMCFLAGS="-x assembler-with-cpp"
```

2.7 Providing ATLAS with kernel cleanup code

As mentioned in Section 2.1, when any problem dimension (eg., `M`, `N`, or `K`) is not a multiple of N_B, ATLAS must call cleanup code to handle the remainder. When the user-contributed kernel is only modestly faster than ATLAS's generated kernel, letting the generated code handle cleanup will probably be an adequate solution. However, when the user-contributed kernel is much faster than the generated code, using the generated cleanup may represent a significant performance drop for many problem sizes (see Section 2.7.5 for an analysis of the cost of cleanup), and thus it becomes necessary for the user to supply ATLAS with cleanup code as well. In order to understand how this is done, it is necessary to discuss how ATLAS does cleanup.

2.7.1 ATLAS and cleanup

As we have seen, ATLAS's main performance kernel is an L1 cache contained matmul with fixed dimension N_B, and when a given problem dimension of the general matmul is not a multiple of N_B, cleanup code must be called. It should be apparent that generating codes with all dimensions fixed at compile time, as we do with the full kernel, is not a good idea for cleanup, since it would result in roughly ${N_B}^3$ cleanup routines. Not only would this make the average executable huge, but it would also probably result in performance degredation due to constant instruction load.

ATLAS therefore normally generates a variable number of cleanup cases, with the number of generated codes minimally being 7, and the maximum number being $6 + N_B$. The number of generated codes can vary because the K cleanup routines are special, sometimes requiring N_B different codes to handle efficiently, as we will see below.

ATLAS splits the generated cleanup into these categories

1. **M-cleanup** $M < N_B$ && $N = K = N_B$: 3 routines, corresponding to `BETA` = 0, 1 and arbitrary
2. **N-cleanup** $N < N_B$ && $M = K = N_B$: 3 routines, corresponding to `BETA` = 0, 1 and arbitrary
3. **K-cleanup** $K <= N_B$ && $M \leq N_B$ && $N \leq N_B$: Only one `BETA` case (arbitrary), but may compile special case for each possible K value, resulting in at least 1, and at most N_B K-cleanup routines

So we see that K-cleanup is special in several ways. First, it is the most general cleanup routine, since it can handle multiple dimensions not being less than N_B, whereas the M- and N-cleanup routines can only have their respective dimensions less then N_B. The second thing to note is that we compile only the most general `BETA` case for K-cleanup; this is due to the fact that we may need N_B different routines to handle K-cleanup efficiently, and multiplying this number of routines by three seems counterproductive.

The final difference in the K-cleanup is the fact that it potentially requires N_B different routines to support. This is due to several factors. Firstly, in ATLAS, the innermost loop

in gemm is the K-loop, making it very important for performance. On systems without good loop handling, such as the x86, heavy K unrollings are critical. Secondly, the leading dimensions of the A and B matrices are fixed to `KB` due to the data copy, which allows for more efficient indexing of these matrices. If a routine takes run-time K (rather than compile-time, as when the dimension is fixed to `KB`), it must also take run-time `lda` and `ldb`, and this extra indexing is too costly on many architectures.

2.7.2 User supplied cleanup

Users can supply cleanup code for the following three cases only, all of which come in the three `BETA` variants:

1. **M-cleanup** $M < N_B$ && $N = K = N_B$
2. **N-cleanup** $N < N_B$ && $M = K = N_B$
3. **K-cleanup** $K < N_B$ && $M = N = N_B$

The generated code handles all cleanup where more than one dimension is less than the blocking factor. This simplification allows ATLAS to avoid having to test ${N_B}^3$ cases when selecting user cleanup. Once the matrices in question are larger than N_B, cleanup with more than one dimension less than N_B rapidly stops being a performance factor. Small matrices where this cleanup is a factor are almost certainly going to be handled by ATLAS's small-case code anyway, so it seems unlikely that this simplification will hurt performance in practice. Section 2.7.5 shows this in a more formal way.

Users need to be very careful when supplying cleanup, because if the user indicates that a dimension must be a compile-time variable, rather than a runtime variable, ATLAS will generate up to N_B routines to handle user cleanup, and since user routines are compiled with all `BETA` variants, it is possible to generate $9N_B$ cleanup cases, in addition to ATLAS's generated cases. It is therefore recommended that the user supply cleanup that uses run-time arguments whenever possible, and indicate kernels taking compile-time dimensions as not to be used for cleanup.

2.7.3 Indicating cleanup in the index file

Any routine that does not throw the flag value of 8 will be evaluated by the install as a cleanup possibility. Flag values are very important for indicating opportunities for cleanup. Here is an example from the release:

```
1 480 4 4 1 1 1 4 4 2 ATL_mm4x4x2US.c "V. Nguyen & P. Strazdins"
```

OK, as always, we can read this to see that `MB` and `NB` must be multiples of 4, and that `KB` can be any value. With no flag modifiers, if we wanted to use the routine for K cleanup, we would have to compile it into N_B different routines, since loop dimensions are compile-time parameters by default. However, this routine is modified by a flag value of 480. What does this mean? Consulting table 1, we see that $32 + 64 + 128 + 256 = 480$, which means lda and ldb are not restricted to `KB` (i.e., they are run-time parameters to the routine), the M-loop is controlled by a run-time variable, the N-loop is controlled by a run-time variable, and the K-loop is controlled by a run-time variable. We therefore know

that we can use this routine for all cleanups (M-, N-, and K-cleanup), and we need only one routine to do so (i.e., we do not have to compile N_B routines to handle all cases). However, it can only be used for M- and N- cleanup cases where the respective dimension is a multiple of 4. Therefore, assuming this kernel is superior to the generated code, it will be used for all K cleanup routines. However, for M and N cleanup, there will be something corresponding to the following pseudocode:

```
if (M % 4 == 0) call ATL_mm4x4x2US
else call generated M cleanup
```

It is clear that without overloading the flag value to an even more ludicrous degree, that cleanup will eventually need to have it's own index file. For instance, it would be nice to be able to insist that a particular K-cleanup code be used only when $K > 3$, for instance, in addition to insisting it be a multiple of a particular value. The fact that cleanup does not already have such a seperate file simply represents a design failure on my part; it was not until I had already produced the system working as it does now that I saw its shortcomings, and then it was too late to change for the release. Subsequent developer releases will probably address this shortcoming.

2.7.4 Testing and timing cleanup

Cleanup is tested in the same way as the normal kernel, but you need to supply additional parameters. `M`, `N`, and `K` are the problem dimensions, and `MB`, `NB`, `KB` are the blocking factors. If the blocking factors are set to zero, that means they are run-time parameters to the routine. `lda`, `ldb`, `ldc` are the leading dimensions of the operand arrays, and they default to `KB`, `KB`, and zero, respectively.

Here is an example of testing an M-cleanup routine, insisting that M is a run-time argument:

```
make mmutstcase mmrout=CASES/ATL_mm1x1x1.c pre=d M=17 N=40 K=40 \
     mb=0 nb=40 kb=40
```

Here is timing the same routine, but insisting that the M-loop is fixed at compile time:

```
make ummcase mmrout=CASES/ATL_mm1x1x1.c pre=d M=17 N=40 K=40 \
     mb=17 nb=40 kb=40
```

Here's testing a K-cleanup routine, taking run-time K and leading dimensions:

```
make mmutstcase mmrout=CASES/ATL_mm1x1x1.c pre=d M=40 N=40 K=27 \
     mb=40 nb=40 kb=0 lda=0 ldb=0
```

The same test taking compile-time K and leading dimensions:

```
make mmutstcase mmrout=CASES/ATL_mm1x1x1.c pre=d M=40 N=40 K=27 \
     mb=40 nb=40 kb=27 lda=27 ldb=27
```

2.7.5 Importance of cleanup

In analyzing the importance of good cleanup for performance, it is necessary to recognize the various types that can occur. The cleanup that user's can supply to ATLAS is *one dimensional cleanup*, i.e., only one of the three possible dimensions is less than N_B. There is also 2 and 3 dimensional cleanup. To give an idea of the relative importance of various catagories of computation, it is roughly true that the matmul kernel is a cubic cost, the one dimensional cleanup is a square cost, the two dimensional cleanup is a linear cost, and the three dimensional cleanup is $O(1)$.

This is shown more formally below. Define $M_r = M \bmod N_B$, let m, n, k be the dimensional arguments to the *gemmK* and/or cleanup, and remember that matrix multiplication takes $2MNK$ flops, and we see that the flop count for each catagory is:

- $\mathbf{m = n = k = N_B}$: $\lfloor \frac{M}{N_B} \rfloor \lfloor \frac{N}{N_B} \rfloor \lfloor \frac{K}{N_B} \rfloor 2{N_B}^3 \Longrightarrow \lfloor \frac{N}{N_B} \rfloor^3 \ 2{N_B}^3$
- $\mathbf{m < N_B, n = k = N_B}$: $\lfloor \frac{N}{N_B} \rfloor \lfloor \frac{K}{N_B} \rfloor 2M_r{N_B}^2 \Longrightarrow \lfloor \frac{N}{N_B} \rfloor^2 \ 2N_r{N_B}^2$
- $\mathbf{n < N_B, m = k = N_B}$: $\lfloor \frac{M}{N_B} \rfloor \lfloor \frac{K}{N_B} \rfloor 2N_r{N_B}^2 \Longrightarrow \lfloor \frac{N}{N_B} \rfloor^2 \ 2N_r{N_B}^2$
- $\mathbf{k < N_B, m = n = N_B}$: $\lfloor \frac{M}{N_B} \rfloor \lfloor \frac{N}{N_B} \rfloor 2K_r{N_B}^2 \Longrightarrow \lfloor \frac{N}{N_B} \rfloor^2 \ 2N_r{N_B}^2$
- $\mathbf{m < N_B, n < N_B, k = N_B}$: $\lfloor \frac{K}{N_B} \rfloor 2M_rN_rN_B \Longrightarrow \lfloor \frac{N}{N_B} \rfloor 2{N_r}^2N_B$
- $\mathbf{m < N_B, k < N_B, n = N_B}$: $\lfloor \frac{N}{N_B} \rfloor 2M_rK_rN_B \Longrightarrow \lfloor \frac{N}{N_B} \rfloor 2{N_r}^2N_B$
- $\mathbf{n < N_B, k < N_B, m = N_B}$: $\lfloor \frac{M}{N_B} \rfloor 2N_rK_rN_B \Longrightarrow \lfloor \frac{N}{N_B} \rfloor 2{N_r}^2N_B$
- $\mathbf{m < N_B, n < N_B, k < N_B}$: $2M_rN_rK_r \Longrightarrow 2{N_r}^3$

Note that the simplified equations to the right of $\Longrightarrow$ assume the square case, i.e. $M = K = N$. The above analysis can now be grouped into the catagories of interest as in:

- **kernel** : $\lfloor \frac{N}{N_B} \rfloor^3 \ 2{N_B}^3$
- **1D cleanup**: $3\lfloor \frac{N}{N_B} \rfloor^2 \ 2N_r{N_B}^2 \quad < \quad 3\lfloor \frac{N}{N_B} \rfloor^2 \ 2{N_B}^3$
- **2D cleanup**: $\lfloor \frac{N}{N_B} \rfloor 2{N_r}^2N_B \quad < \quad 3\lfloor \frac{N}{N_B} \rfloor \ 2{N_B}^3$
- **3D cleanup**: $2{N_r}^3 \quad < \quad 2{N_B}^3$

The simplified equations to the right of the $<$ above provide a safe upper bound on cleanup cost by setting $N_r = N_B$ (in reality, $0 < N_r < N_B$, of course).

With this analysis, we can easily see why it is not important for the user to be able to contribute 2D and 3D cleanup cases: remember that all of these kernels are for ATLAS's *large-case* gemm. ATLAS has a seperate small-case gemm, which is invoked when the problem is so small that the $2N^2$ copy cost is significant compared to the $2N^3$ computational costs. So, in the cases where the $O(N)$ 2D cleanup or $O(1)$ 3D cleanup costs are prohibitive, this large-case gemm will probably not be used anyway.

2.8 *gemmK* usage notes

The assumptions behind this kernel are that the input operands are loaded to L1 only one time (i.e., the blocking guarantees that all of the matrix accessed in the inner loop plus the active panel of the matrix in the outer loop fits in L1). For very large caches, all three operands may fit into cache, but this is typically not the case. Because this *gemmK* is called by routines that place K as the innermost loop, the output operand C will typically come from the L2 cache (except, obviously, on the first of the $\frac{K}{N_B}$ such calls). ATLAS uses the JIK loop variant of on-chip multiply, and thus all of A fits in cache, with nu columns of B. To take an example, say you are using mu = nu = 4, with $N_B = 40$, then the idea is that the 40×40 piece of A, along with the 40×4 piece of B (the active panel of B), and the 4×4 section of C all fit into cache at once, with enough room for the load of the next step, and any junk the algorithm might have in L1. That panel of B is applied to all of A, and then a new panel is loaded. Since the panel has been applied to all A, it will never be reloaded, and thus we see that B is loaded to L1 only one time. Since all of A fits in L1, and we keep it there across all panels of B, it is also loaded to L1 only one time.

If written appropriately, loading all of B with a few rows of A should theoretically be just as efficient (i.e., the IJK variant of matmul). However, the variants where K is not the innermost loop are unlikely to work well in ATLAS, if for no other reason than the transpose settings we have chosen militate against it.

Note that the $\beta = 0$ case must not read C, since the memory may legally be unitialized.

2.9 Getting ATLAS to use your kernel

OK, so let's say you've got a kernel that is faster than what ATLAS is presently using, how do you get ATLAS to use it? First, of course, you put the source in the CASES directory, and update the appropriate `<pre>cases` index file. Then, you take different steps depending on how you wish to do the install, as discussed in the following sections. In all of these discussions, `<pre>` is replaced by your type/precision modifier (one of `s`, `d`, `c`, `z`).

2.9.1 Putting it in by hand

In an already-installed ATLAS, you can make ATLAS reinstall just the kernel. From your `BLDdir/tune/blas/gemm/` directory, issue these commands:

```
rm res/<pre>u*
rm res/<pre>MMRES
./xmmsearch -p <pre>
make <pre>install
```

2.9.2 With a fresh install

First, run config as usual. Then, tail the created `Make.inc` file, and see if the macro `INSTFLAGS` includes `-a 1`. If so, ATLAS has some architectural defaults for your architecture (though perhaps not for your compiler, if you have forced the use of a non-default compiler), which won't include your shiny new kernel. So, you will need to remove the files indicating the default `gemmK` kernel for your precision. To do this, scope your `ARCH` setting in your `Make.inc`. For the purposes of this discussion, let us say it set to `Core2Duo64SSE3`

(i.e., in the below example, substitute the definition of ARCH for Core2Duo64SSE3). Go to ATLAS/CONFIG/ARCHS, and issue the following commands:

```
gunzip -c Core2Duo64SSE3.tgz | tar xvf -
rm Core2Duo64SSE3/gemm/gcc/<pre>u*
rm Core2Duo64SSE3/gemm/gcc/<pre>MMRES
rm Core2Duo64SSE3.tgz
tar cvf Core2Duo64SSE3.tar Core2Duo64SSE3
gzip Core2Duo64SSE3.tar
mv Core2Duo64SSE3.tar.gz Core2Duo64SSE3.tgz
```

Now, continue install as normal, and your kernel should be used if it beats what ATLAS is presently using. Note that this assumes you are using gcc as the gemmK compiler, which is the default on most systems. If you are using a different compiler, you would substitute its name instead of gcc in the above lines. If there is no subdirectory with the name of your compiler in the tarfile, ATLAS has no architectural defaults for that compiler, and thus you need to make no changes to the tarfile.

2.9.3 With an old install, but using full install command

When rebuilding an old install, the main trap is to avoid having architectural defaults make it so you don't time your new kernel. Follow the instructions given in Section 2.9.2, but additionally make sure you delete any prexisting directory that matches your ARCH definiton. Therefore, in the above example, in the ATLAS/CONFIG/ARCHS directory, you would additionally issue:

```
rm -rf Core2Duo64SSE3
```

If such a subdirectory existed.

From your BLDdir directory, then issue:

```
rm bin/INSTALL_LOG/*
rm tune/blas/gemm/res/<pre>u*
rm tune/blas/gemm/res/<pre>MMRES
make build
```

2.10 Contributing a complete GEMM implementation

This feature has been temporarily disabled in 3.8, though it may be re-enabled in the 3.9 series if there is user demand. This section is therefore kept around solely historical purposes, and will need to be updated if the feature is added back in.

Contributing an L1 kernel is the prefered method of user contribution for Level 3 BLAS speedup, but it is not the only one supported by ATLAS. ATLAS also allows a user to contribute a complete system-specific GEMM implementation. This method of contribution is far less desirable than kernel contribution, and thus the standards of acceptance are correspondingly higher.

When only a kernel is contributed, it is only used when timings indicate it is superior to the best ATLAS-supplied routine for a given architecture. Because kernel routines are

called in known ways by the ATLAS infrastructure, the timer can be made to accurately reflect typical usage. A full GEMM, which is to all intents called directly by the user, has no "typical" usage, and the timer is thus not able to ensure that the user's full GEMM is superior to that supplied by ATLAS in a system-independent way, even if the additional installation time required to choose amoung full GEMM implementations were allowed. Thus, full GEMM implementations will be used only when ATLAS's configuration detects a known architecture where the ATLAS team has certified the full GEMM to be significantly better than ATLAS's native GEMM, across the entire spectrum of problem shapes and sizes (with the exception of those shapes and sizes handled by ATLAS's non-copy code, as explained below).

As explained in Section 2.1, ATLAS has both a small-case matmul, which does not copy the user's input operands, and a large-case code that does. The user contributed GEMM replaces ATLAS's large-case GEMM, and then timings are used as normal to determine the crossover points at which the contributed GEMM outperforms ATLAS's small-case code.

2.10.1 Supplying ATLAS with what it needs

ATLAS expects that the contributed GEMM will have its own architecture- specific subdirectory, just as with all other ATLAS source directories. That directory is indicated to ATLAS by the `UMMdir` macro set in `Make.inc`. For instance, on the alpha platform, Mr. Goto's GEMM is used by ATLAS, and UMMdir is therefore set to: `$(TOPdir)/src/blas/gemm/GOTO/$(ARCH)`.

In this directory, there must be a master makefile, called `Makefile`, which minimally contains the following targets:

- `susermm` : builds the single precision real gemm with all its dependencies
- `dusermm` : builds the double precision real gemm with all its dependencies
- `cusermm` : builds the single precision complex gemm with all its dependencies
- `zusermm` : builds the double precision complex gemm with all its dependencies
- `sclean` : deletes all non-library files created by `susermm`
- `dclean` : deletes all non-library files created by `dusermm`
- `cclean` : deletes all non-library files created by `cusermm`
- `zclean` : deletes all non-library files created by `zusermm`

For each precision, ATLAS calls the user's GEMM using this API:

```
int ATL_U<pre>usergemm(const enum ATLAS_TRANS TA, const enum ATLAS_TRANS TB,
                        const int M, const int N, const int K,
                        const SCALAR alpha, const TYPE *A, const int lda,
                        const TYPE *B, const int ldb, const SCALAR beta,
                        TYPE *C, const int ldc)
```

where,

<pre> :	s	d	c	z
SCALAR	float	double	float*	float*
TYPE	float	double	float	float

This routine should return 0 upon successful invocation, and -1 if unable to malloc enough memory. Other errors may be signaled by returning a value of 2. On error in this routine, ATLAS will call the no-copy code to get the answer, so a return value of 1 indicates that ATLAS should do this. If a fatal error occurs, or if an error occurs after operands have been modified (i.e., calling the no-copy code will no longer produce the correct answer), then execution should be halted.

ATLAS's interface routines have already done all required error checking, so the user need not check the input arguments in this routine, or any of the lower-level user contributed routines.

2.10.2 What to do if you don't supply all precisions

Remember that what ATLAS is doing is substituting your GEMM for its own large-case GEMM. However, ATLAS's large-case GEMM is still compiled in the library, it is just not being used. The following code will call ATLAS's large case code for all precisions:

```
#include "atlas_misc.h"
#include "atlas_lvl3.h"

int Mjoin(Mjoin(ATL_U,PRE),usergemm)
   (const enum ATLAS_TRANS TA, const enum ATLAS_TRANS TB,
    const int M, const int N, const int K, const SCALAR alpha,
    const TYPE *A, const int lda, const TYPE *B, const int ldb,
    const SCALAR beta, TYPE *C, const int ldc)
{
   int ierr;

   if (N >= M)
   {
      if ( ierr = Mjoin(PATL,mmJIK)(TA, TB, M, N, K, alpha, A, lda, B, ldb,
                                    beta, C, ldc) )
         ierr = Mjoin(PATL,mmIJK)(TA, TB, M, N, K, alpha, A, lda, B, ldb,
                                  beta, C, ldc);
   }
   else
   {
      if ( ierr = Mjoin(PATL,mmIJK)(TA, TB, M, N, K, alpha, A, lda, B, ldb,
                                    beta, C, ldc) )
         ierr = Mjoin(PATL,mmJIK)(TA, TB, M, N, K, alpha, A, lda, B, ldb,
                                  beta, C, ldc);
   }
```

```
   return (ierr);
}
```

So, you can use the above code to have ATLAS call it's normal routines for those precisions you do not supply, and call your own otherwise. In order to help understand this process, ATLAS includes a "user-supplied" GEMM that simply calls ATLAS's own GEMM for all precisions, in `ATLAS/src/blas/gemm/UMMEXAMPLE`. If, for instance, you have a single precision real GEMM but nothing else, you can take this directory as a starting point, adding your own commands for single precision real in the Makefile, etc, and leaving everything else alone.

In order to do this, you should do the following:

1. In `ATLAS/src/blas/gemm/UMMEXAMPLE`:
   - `mkdir <arch>`
   - `cp Makefile <arch>/.`
   - `ln -s ../../../../../Make.<arch> Make.inc`
   - Modify `<arch>/Makefile` to compile your single precision routine
   - Follow the instructions given in Section 2.10.3, which discusses how to make ATLAS use a contributed GEMM that config has not setup for you

2.10.3 Forcing ATLAS to use your GEMM

If ATLAS detects you are on a platform where a contributed full GEMM is superior to ATLAS's large-case GEMM, ATLAS will automatically handle the details of making ATLAS call the user-contributed routine. If, however, you wish to force ATLAS to use your GEMM (for instance, you are testing your code before contribution, or just want to utilize ATLAS for complete BLAS coverage with your GEMM), you should take the following steps, after creating the appropriate subdirectory and API as previously described:

1. Edit your `Make.inc` file:
   - Change `UMMdir` to point to your full GEMM's architecture subdirectory
   - Add `-DUSERGEMM` to the `CDEFS` macro.
2. In `ATLAS/src/blas/gemm`, touch `ATL_gemmXX.c` and `ATL_AgemmXX.c` to force recompilation
3. In `BLDdir/src/blas/gemm/` type `make lib`
4. In `BLDdir/include/`, issue
   - `rm ?Xover.h atlas_cacheedge.h`
   - `touch altas_cacheedge.h sXover.h dXover.h cXover.h zXover.h`
5. In `BLDdir/tune/blas/gemm/`, issue:
   - `rm res/?Xover.h res/atlas_cacheedge.h`
   - `make res/atlas_cacheedge.h`

- `make res/sXover.h pre=s`
- `make res/dXover.h pre=d`
- `make res/cXover.h pre=c`
- `make res/zXover.h pre=z`

3 Speeding up the Level 2 BLAS

ATLAS presently empirically tunes four kernels to optimize the various L2BLAS, as shown in Table 2. This table shows matvec used to tune SYMV and HEMV, and this is mostly true. For essentially any modern x86 or ARM, GEMV will be used speed up HEMV and SYMV, but on other systems, ATLAS will simply use the reference implementation, which may be faster. The problem is that to support HEMV/SYMV optimally, we need a kernel which we have not yet empirically tuned. You can build SYMV and HEMV out of GEMV, but in doing so you bring the matrix A into registers twice (once from memory, and once from cache). Since the load of A is the dominant cost in these operations, that is not good news. However, for places where vector instructions speed up memory access (modern x86) and where the compiler can't do a great job (ARM), using a tuned GEMV in this fashion is still faster than just calling a reference version. So, for most systems, speeding up the GEMV kernels will cause a corresponding speedup in SYMV and GEMV, but these operations are the least well-tuned BLAS that ATLAS supports (as a percentage of achievable peak, not raw MFLOP, of course). All other L2BLAS operations should be well optimized, particularly for large problems.

Mneum	operations	Used to support
`mvn_k`	$y \leftarrow Ax$, $y \leftarrow Ax + y$	GEMV, TRMV, TRSV, HEMV, SYMV
`mvt_k`	$y \leftarrow A^T x$, $y \leftarrow A^T x + y$	GEMV, TRMV, TRSV, HEMV SYMV
`ger_k`	$A \leftarrow xy^T + A$	GER, GERU, GERC, SYR, HER
`ger2_k`	$A \leftarrow xy^T + wz^T + A$,	GER2, GER2U, GER2C, SYR2, HER2

Table 2: Level-2 BLAS kernels, and the L2BLAS routines they improve

In this section, we use some macros that are automatically defined by the ATLAS build system. The first is `ATL_CINT` which is presently an alias for `const int`. The macros `SCALAR` and TYPE are defined according the precision being being compiled:

<pre> :	s	d	c	z
SCALAR	float	double	float*	float*
TYPE	float	double	float	float

3.1 Testing and timing `mvn_k`

The API of this routine is given by:

```
void ATL_UGEMV(ATL_CINT M, ATL_CINT N, const TYPE *A, ATL_CINT lda,
               const TYPE *X, TYPE *Y)
```

If the routine is compiled with the macro BETA0 defined, then it should perform the operation $y \leftarrow Ax$; if this macro is not defined, then it should perform $y \leftarrow Ax + y$. A is a column-major $lda \times N$ contiguous array where $lda \geq M$. M specifies both the number of rows of A and the length of the vector X. N provides the number of columns in A, and the length of the vector Y.

Since this is a $O(MN)$ operation, and A is $M \times N$ in size, this algorithm is dominated by the load of A. No reuse of A is possible, so the best we can do is reuse the vectors (through operations like register and cache blocking).

Each element of Y can obviously be computed by doing a dot product of the corresponding row of A with X. However, this accesses the matrix along the non-contiguous dimension, which leads to terrible memory heirarchy usage for any reasonably sized matrix. Therefore, mvn_k cannot be written in this fashion.

In order to optimize the access of A (dominant algorithmic cost) we must access it in a column-major fashion. Therefore, instead of writing it as a series of dot products (ddots), we can write it as a series of axpys (recall that axpy does the operation $y \leftarrow \alpha x + y$).

Therefore, instead of being composed of a series of ddots, mvn_k should be implemented as a series of axpys. In this formulation, we access columns of A (the contiguous dimension) at a time, and we write update all elements of Y

A simple implementation of this kernel (for real precisions) is:

```
#include "atlas_misc.h"  /* define TYPE macros */
void ATL_UGEMV(ATL_CINT M, ATL_CINT N, const TYPE *A, ATL_CINT lda,
               const TYPE *X, TYPE *Y)
{
   register int j, i;

/*
 * Peel 1st iteration of N-loop if BETA0 is defined
 */
   #ifdef BETA0
   {
      const register TYPE x0 = *X;
      for (i=0; i != M; i++)
         Y[i] = A[i] * x0;
      j=1;
      A += lda;   /* done with this column of A */
   }
   #else
      j=0;
   #endif
   for (; j != N; j++)
   {
      const TYPE register x0 = X[j];
      register TYPE y0 = Y[i];

      for (i=0; i != M; i++)
         Y[i] += A[i] * x0;
```

```
      A += lda;   /* done with this column of A */
   }
}
```

To time and test `mvn_k` kernel, its implementation must be stored in the `OBJdir/tune/blas/gemv/MVNCASES` directory. Assuming I saved the above implementation to `mvn.c` in the above directory, I can test the kernel from the `OBJdir/tune/blas/gemv` directory with the command: `make <pre>mvnktest`, where `<pre>` specifies the type/precision and is one of : `s`, `d`, `c`, or `z`.

This target uses the following `make` variables which you can change to vary the type of testing done; number in parens are the default values that will be used if no command-line override is given:

- `mu` (1): Unrolling on M dimension
- `nu` (1): Unrolling on N dimension
- `mvnrout`: the filename of your kernel implementation
- `Mt` (297): Number of rows of A (elements of y)
- `Nt` (177): Number of columns of A (elements of x)
- `ldat` (`Mt`): leading dimension for the matrix A
- `align` (`-Fx 16 -Fy 16 -Fa 16`): alignments that matrix and vectors must adhere to.
- `<pre>MVCC` (ATLAS default compiler) : compiler required to compile your kernel
- `<pre>MVCFLAGS` (ATLAS default flags) : compiler flags required to compile your kernel
- `beta` (1) : 0 or 1 (scale of y)

Therefore, to test the $\beta = 1.0$ case of double precision real:

```
>make dmvnktest mu=1 nu=1 mvnrout=mvn.c
.... bunch of compilation, etc ....
   TEST M=997, N=177, lda=1111, STARTED
   TEST M=997, N=177, be=0.00, lda=1111, incXY=1,1 PASSED
```

To test single precision $\beta = 0.0$ with a different shape:

```
>make smvnktest mu=1 nu=1 mvnrout=mvn.c beta=0 Mt=801 Nt=55 ldat=1000
.... bunch of compilation, etc ....
   TEST M=801, N=55, lda=1000, STARTED
   TEST M=801, N=55, be=0.00, lda=1000, incXY=1,1 PASSED
```

Now we are ready to time this masterpiece. We have two choices for timers. The first such timer simply calls your newly-written kernel directly. It does no cache flushing at all: it initializes the operands (bringing them into the fastest level of heirarchy into which they fit), and times the operations. This is the timer to use when you want to preload the

problem to a given level of cache and see how fast your kernel is in isolation. The make target for this methodology is `<pre>mvnktime`.

The second make target calls ATLAS's GEMV driver that builds the full GEMV from the kernels. This timer does cache flushing, and is what you should use to estimate how fast the complete GEMV will be. This make target is `<pre>mvntime`.

Both of these targets take largely the same make macros:

- `mu` (1): Unrolling on M dimension
- `nu` (1): Unrolling on N dimension
- `mvnrout`: the filename of your kernel implementation
- `M` (1000): Number of rows of A (elements of y)
- `N` (1000): Number of columns of A (elements of x)
- `lda` (`M`): leading dimension for the matrix A
- `align` (`-Fx 16 -Fy 16 -Fa 16`): alignments that matrix and vectors must adhere to.
- `<pre>MVCC` (ATLAS default compiler) : compiler required to compile your kernel
- `<pre>MVCFLAGS` (ATLAS default flags) : compiler flags required to compile your kernel; pass "-x assembler-with-cpp" (along with any architecture-specific flags) if your kernel is written in assembly (assuming your compiler is `gcc`).
- `beta` (1) : 0 or 1 (scale of y)
- `flushKB` : for `mvntime`, kilobytes of memory to flush.

Therefore, to time the probable speed of a complete GEMV of size 1000x32 while flushing 16MB of memory, I would issue:

```
>make dmvntime mu=1 nu=1 mvnrout=mvn.c M=1000 N=32 flushKB=16384
GEMV: M=1000, N=32, lda=1008, AF=[16,16,16], AM=[0,0,0], beta=1.000000e+00, alpha=1.000000e+0
   M=1000, N=32, lda=1008, nreps=57, time=2.011856e-05, mflop=3230.85
   M=1000, N=32, lda=1008, nreps=57, time=2.011042e-05, mflop=3232.15
   M=1000, N=32, lda=1008, nreps=57, time=2.006722e-05, mflop=3239.11
NREPS=3, MAX=3239.11, MIN=3230.85, AVG=3234.04, MED=3232.15
```

3.2 Testing and timing `mvt_k`

The API of this routine is given by:

```
void ATL_UGEMV(ATL_CINT M, ATL_CINT N, const TYPE *A, ATL_CINT lda,
               const TYPE *X, TYPE *Y)
```

If the routine is compiled with the macro `BETA0` defined, then it should perform the operation $y \leftarrow A^T x$; if this macro is not defined, then it should perform $y \leftarrow A^T x + y$. A is a column-major $lda \times N$ contiguous array where $lda \geq M$. M specifies both the number of rows of

A and the length of the vector X. N provides the number of columns in A, and the length of the vector Y.

Since this is a $O(MN)$ operation, and A is $M \times N$ in size, this algorithm is dominated by the load of A. No reuse of A is possible, so the best we can do is reuse the vectors (through operations like register and cache blocking).

Each element of Y can obviously be computed by doing a dot product of the corresponding row of A with X, and this gives us the simplist implementation possible:

```
#include "atlas_misc.h"  /* define TYPE macros */
void ATL_UGEMV(ATL_CINT M, ATL_CINT N, const TYPE *A, ATL_CINT lda,
               const TYPE *X, TYPE *Y)
{
   register int j, i;

   for (j=0; j < N; j++)
   {
      #ifdef BETA0
         register TYPE y0 = 0.0;
      #else
         register TYPE y0 = Y[j];
      #endif
      for (i=0; i < M; i++)
         y0 += A[i] * X[i];
      Y[j] = y0;
      A += lda;  /* done with this column */
   }
}
```

To time and test a `mvt_k` kernel, its implementation must be stored in the `OBJdir/tune/blas/gemv/MVTCASES` directory. Assuming I saved the above implementation to `mvt.c` in the above directory, I can test the kernel from the `OBJdir/tune/blas/gemv` directory with the command: `make <pre>mvtktest`, where `<pre>` specifies the type/precision and is one of : `s`, `d`, `c`, or `z`.

This target uses the following `make` variables which you can change to vary the type of testing done; number in parens are the default values that will be used if no command-line override is given:

- `mu` (1): Unrolling on M dimension
- `nu` (1): Unrolling on N dimension
- `mvtrout`: the filename of your kernel implementation
- `Mt` (297): Number of rows of A (elements of y)
- `Nt` (177): Number of columns of A (elements of x)
- `ldat` (`Mt`): leading dimension for the matrix A
- `align` (`-Fx 16 -Fy 16 -Fa 16`): alignments that matrix and vectors must adhere to.

- <pre>MVCC (ATLAS default compiler) : compiler required to compile your kernel
- <pre>MVCFLAGS (ATLAS default flags) : compiler flags required to compile your kernel
- beta (1) : 0 or 1 (scale of y)

Therefore, to test the $\beta = 1.0$ case of double precision real:

```
>make dmvtktest mu=1 nu=1 mvtrout=mvt.c
.... bunch of compilation, etc ....
   TEST M=997, N=177, lda=1111, STARTED
   TEST M=997, N=177, be=1.00, lda=1111, incXY=1,1 PASSED
```

We have two choices for timers. The first such timer simply calls your newly-written kernel directly. It does no cache flushing at all: it initializes the operands (bringing them into the fastest level of heirarchy into which they fit), and times the operations. This is the timer to use when you want to preload the problem to a given level of cache and see how fast your kernel is in isolation. The make target for this methodology is <pre>mvtktime.

The second make target calls ATLAS's GEMV driver that builds the full GEMV from the kernels. This timer does cache flushing, and is what you should use to estimate how fast the complete GEMV will be. This make target is <pre>mvttime.

Both of these targets take largely the same make macros:

- mu (1): Unrolling on M dimension
- nu (1): Unrolling on N dimension
- mvtrout: the filename of your kernel implementation
- M (1000): Number of rows of A (elements of y)
- N (1000): Number of columns of A (elements of x)
- lda (M): leading dimension for the matrix A
- align (-Fx 16 -Fy 16 -Fa 16): alignments that matrix and vectors must adhere to.
- <pre>MVCC (ATLAS default compiler) : compiler required to compile your kernel
- <pre>MVCFLAGS (ATLAS default flags) : compiler flags required to compile your kernel; pass "-x assembler-with-cpp" (along with any architecture-specific flags) if your kernel is written in assembly (assuming your compiler is gcc).
- beta (1) : 0 or 1 (scale of y)
- flushKB : for mvntime, kilobytes of memory to flush.

Therefore, to time the probable speed of a complete GEMV of size 1000x32 while flushing 16MB of memory, I would issue:

```
>make dmvttime mu=1 nu=1 mvtrout=mvt.c M=1000 N=32 flushKB=16384
GEMV: M=1000, N=32, lda=1008, AF=[16,16,16], AM=[0,0,0], beta=1.000000e+00, alpha=1.000000e+0
   M=1000, N=32, lda=1008, nreps=57, time=2.809835e-05, mflop=2313.30
   M=1000, N=32, lda=1008, nreps=57, time=2.812956e-05, mflop=2310.74
   M=1000, N=32, lda=1008, nreps=57, time=2.807517e-05, mflop=2315.21
NREPS=3, MAX=2315.21, MIN=2310.74, AVG=2313.08, MED=2313.30
```

3.3 Testing and timing ger_k

The API of ger_k is given by:

```
void ATL_UGERK(ATL_CINT M, ATL_CINT N, const TYPE *X, const TYPE *Y,
               TYPE *A, ATL_CINT lda)
```

This kernel performs the outer product operation $A \leftarrow xy^T + A$. A is a column-major $lda \times N$ contiguous array where $lda \geq M$. M specifies both the number of rows of A and the length of the vector X. N provides the number of columns in A, and the length of the vector Y.

Since this is a $O(MN)$ operation, and A is $M \times N$ in size, this algorithm is dominated by the load of A. No reuse of A is possible, so the best we can do is reuse the vectors (through operations like register and cache blocking).

The simplist implementation of this operation is done by doing a axpy operation for each column of the matrix:

```
#include "atlas_misc.h"  /* define TYPE macros */
void ATL_UGERK(ATL_CINT M, ATL_CINT N, const TYPE *X, const TYPE *Y,
               TYPE *A, ATL_CINT lda)
{
   register int j, i;

   for (j=0; j < N; j++)
   {
      const register TYPE y0 = Y[j];
      for (i=0; i < M; i++)
         A[i] += X[i] * y0;
      A += lda;  /* done with this column */
   }
}
```

To time and test a ger_k kernel, its implementation must be stored in the OBJdir/tune/blas/gemv/R1CASES directory. Assuming I saved the above implementation to ger.c in the above directory, I can test the kernel from the OBJdir/tune/blas/ger directory with the command: make <pre>r1ktest, where <pre> specifies the type/precision and is one of : s, d, c, or z.

This target uses the following make variables which you can change to vary the type of testing done; number in parens are the default values that will be used if no command-line override is given:

- mu (1): Unrolling on M dimension
- nu (1): Unrolling on N dimension
- r1rout: the filename of your kernel implementation
- Mt (297): Number of rows of A (elements of x)
- Nt (177): Number of columns of A (elements of y)

- `ldat` (`Mt`): leading dimension for the matrix A
- `align` (`-Fx 16 -Fy 16 -Fa 16`): alignments that matrix and vectors must adhere to.
- `<pre>R1CC` (ATLAS default compiler) : compiler required to compile your kernel
- `<pre>R1CFLAGS` (ATLAS default flags) : compiler flags required to compile your kernel

To test this kernel with a 511x220 matrix, I would issue:

```
>make dr1ktest mu=1 nu=1 r1rout=ger.c Mt=511 Nt=220
.... bunch of compilation, etc ....
   TEST CONJ=0, M=511, N=220, lda=511, incY=1, STARTED
   TEST CONJ=0, M=511, N=220, lda=511, incY=1, PASSED
```

We have two choices for timers. The first such timer simply calls your newly-written kernel directly. It does no cache flushing at all: it initializes the operands (bringing them into the fastest level of heirarchy into which they fit), and times the operations. This is the timer to use when you want to preload the problem to a given level of cache and see how fast your kernel is in isolation. The make target for this methodology is `<pre>r1ktime`.

The second make target calls ATLAS's GER driver that builds the full GER from the kernels. This timer does cache flushing, and is what you should use to estimate how fast the complete GER will be. This make target is `<pre>r1time`.

Both of these targets take largely the same make macros:

- `mu` (1): Unrolling on M dimension
- `nu` (1): Unrolling on N dimension
- `r1rout`: the filename of your kernel implementation
- `M` (1000): Number of rows of A (elements of x)
- `N` (1000): Number of columns of A (elements of y)
- `lda` (`M`): leading dimension for the matrix A
- `align` (`-Fx 16 -Fy 16 -Fa 16`): alignments that matrix and vectors must adhere to.
- `<pre>R1CC` (ATLAS default compiler) : compiler required to compile your kernel
- `<pre>R1CFLAGS` (ATLAS default flags) : compiler flags required to compile your kernel; pass "-x assembler-with-cpp" (along with any architecture-specific flags) if your kernel is written in assembly (assuming your compiler is `gcc`).
- `beta` (1) : 0 or 1 (scale of y)
- `flushKB` : for `mvntime`, kilobytes of memory to flush.

Therefore, to time the probable speed of a complete GER of size 1000x220 while flushing 16MB of memory, I would issue:

```
>make dr1time mu=1 nu=1 r1rout=ger.c M=800 Nt=220 flushKB=16384
.... bunch of compilation, etc ....
GER1: M=800, N=1000, lda=800, AF=[16,16,16], AM=[0,0,0], alpha=1.000000e+00:
   M=800, N=1000, lda=800, nreps=3, time=8.589835e-04, mflop=1863.60
   M=800, N=1000, lda=800, nreps=3, time=8.340690e-04, mflop=1919.27
   M=800, N=1000, lda=800, nreps=3, time=8.746753e-04, mflop=1830.16
NREPS=3, MAX=1919.27, MIN=1830.16, AVG=1871.01, MED=1863.60
```

To time the kernel alone without cache flushing:

```
>make dr1ktime mu=1 nu=1 r1rout=ger.c M=800 Nt=220
.... bunch of compilation, etc ....
GER1: M=800, N=1000, lda=800, AF=[16,16,16], AM=[0,0,0], alpha=1.000000e+00:
   M=800, N=1000, lda=800, nreps=3, time=6.657494e-04, mflop=2404.51
   M=800, N=1000, lda=800, nreps=3, time=6.664957e-04, mflop=2401.82
   M=800, N=1000, lda=800, nreps=3, time=6.842208e-04, mflop=2339.60
NREPS=3, MAX=2404.51, MIN=2339.60, AVG=2381.97, MED=2401.82
```

3.4 Testing and timing ger2_k

The rank-2 update kernel `ger2_k` performs the operation $A \leftarrow xy^T + wz^T + A$, and has the API:

```
void ATL_UGER2K
   (ATL_CINT M, ATL_CINT N, const TYPE *X, const TYPE *Y,
    const TYPE *W, const TYPE *Z, TYPE *A, ATL_CINT lda)
```

The rank-2 update kernel `ger2_k` uses the exact same testing and timing methodology as described for `ger_k` in the previous section, except the kernel must be stored in the `R2CASES/` subdirectory, and you substute "r2" for "r1" in the testing and timing commands, and "R2" for "R1" in the compiler and flag macros.

A simple GER2 implememtation would be:

```
#include "atlas_misc.h"  /* define TYPE macros */
void ATL_UGER2K
   (ATL_CINT M, ATL_CINT N, const TYPE *X, const TYPE *Y,
    const TYPE *W, const TYPE *Z, TYPE *A, ATL_CINT lda)
{
   register ATL_INT i, j;

   for (j=0; j < N; j++)
   {
      const register TYPE y0=Y[j], z0=Z[j];
      for (i=0; i < M; i++)
         A[i] += X[i]*y0 + W[i]*z0;
      A += lda;  /* finished with this column */
   }
}
```

Assuming I save the above file to `R2CASES/r2k.c`, I would test:

```
>make sr2ktest mu=1 nu=1 r2rout=r2k.c
.... bunch of compilation, etc ....
   TEST CONJ=0, M=297, N=177, lda=297, incY=1, STARTED
   TEST CONJ=0, M=297, N=177, lda=297, incY=1, PASSED
```

And time the single precision real kernel without cache flushing with:

```
>make sr2ktime mu=1 nu=1 r2rout=r2k.c
.... bunch of compilation, etc ....
GER2: M=1000, N=1000, lda=1000, AF=[16,16,16], AM=[0,0,0], alpha=1.000000e+00:
   M=1000, N=1000, lda=1000, nreps=1, time=9.489282e-04, mflop=4217.39
   M=1000, N=1000, lda=1000, nreps=1, time=9.714776e-04, mflop=4119.50
   M=1000, N=1000, lda=1000, nreps=1, time=9.486141e-04, mflop=4218.79
NREPS=3, MAX=4218.79, MIN=4119.50, AVG=4185.22, MED=4217.39
```

4 Speeding up the Level 1 BLAS

4.1 General comments for Level 1 optimization

All Level 1 optimizations are carried on in the `BLDdir/tune/blas/level1` directory and its subdirectories. Under this directory are subdirs with names corresponding to the generic name of the routine in question (eg. AXPY, IAMAX, DOT, etc). It is in these subdirs that the user should place the routines to test and time.

A great deal of the performance win to be had on the Level 1 BLAS, particularly for long vectors, comes from using data prefetch. ATLAS now includes a prefetch header file (described in Section 6.1), which makes prefetch instructions for various systems available for C programmers.

4.1.1 No general kernels here

The Level 1 BLAS are in general too basic to be written in terms of simpler kernels. Therefore, each Level 1 routine must be pretty much optimized individually. The only real reuse of kernels comes from either complex-to-real reuse, or one BLAS routine simplifying to another.

For an example of complex-to-real reuse, consider ZNRM2 which, when called with `incX = 1`, can be simply implemented as a call to DNRM2 with `2*N`. An example of a routine simplifying to another would be calling DSCAL with `alpha = 0.0`, which devolves to a call to the primitive `ATL_dset`.

Therefore, if you are planning to optimize a particular case, be sure to read the appopriate section below to make sure that the case you want to optimize is not implemented by a call to another routine.

As with other ATLAS optimizations, each routine has its own kernel index file, one for each precision (eg., `AXPY/dcases.dsc` indexes the various DAXPY implementations that should be tested and timed during the install process). All of these index files follow the format below, though they leave out unneeded parameters (eg., SCAL, which operates on only one vector, will not have an entry for `incY` or `BETA`).

4.1.2 General index file description

The general form of the index files are:

```
<Number of kernels>
<ID> <alpha> <beta> <incX> <incY> <source file> <author name>
 .
 .
<ID> <alpha> <beta> <incX> <incY> <source file> <author name>
```

Here's an explanation of these values:

- **ID**: Strictly positive (> 0) integer which must be unique in the file (eg., two lines should not begin with the same ID). This ID is used to distinguish between kernels, so reusing one will result in confusion.
- **ialpha**: Integer flag describing special conditions of the scalar `alpha`. Possible values are:
  - **1**: alpha = 1.0 (for complex, (1.0, 0.0))
  - **-1**: alpha = -1.0 (for complex, (-1.0, 0.0))
  - **0**: For complex only, this flag means that the imaginary component is zero (i.e., scalar is actually real, not complex).
  - **2**: or anything else, is assumed to mean that nothing special is known about the scalar.
- **ibeta**: Same as `ialpha`, but for second scalar.
- **incX**: Stride on the X vector your kernel is specialized for (eg., `incX=1` means the kernel works only for X vectors with unit stride).
- **incY**: Vector stride for Y vector.
- **source file**: The name of the source file appearing in the BLAS subdir (eg, `AXPY/myaxpy.c` would mean an entry of `myaxpy.c`).
- **author name**: Name of the author, enclosed in double quotes.

For example, here is an example `AXPY/dcases.dsc`:

```
3
1  2  0  0 axpy1_x0y0.c       "R. Clint Whaley"
2  2  1  1 axpy1_x1y1.c       "R. Clint Whaley"
2 -1  1  1 axpy1_an1x1y1.c    "R. Clint Whaley"
<ID> <ialpha> <incX> <incY> <rout> <author> [\
```

So, to make this work, all three filenames mentioned above must appear in your AXPY/ subdir. The first kernel routine is the general case: general incX, incY, and alpha. The second specializes both vector increments to unit stride, but takes any alpha. Finally, the last routine specializes even further, with unit strides and requiring alpha be negative one.

Note that all index files must contain at least one routine that handles the most general cases, and passes the tester.

If you need a particular compiler and flag combo to compile your kernel you end the line with a \, and put the compiler on the following line, and the flags on a line after that. So, if `axpy_x1y1.c` needed `gcc`, with `-O1 -fschedule-insns -DGOODPERFORMANCE` for flags, the above file would be changed to:

```
3
1  2  0  0 axpy1_x0y0.c       "R. Clint Whaley"
2  2  1  1 axpy1_x1y1.c       "R. Clint Whaley" \
gcc
-O1 -fschedule-insns -DGOODPERFORMANCE
2 -1  1  1 axpy1_an1x1y1.c    "R. Clint Whaley"
<ID> <ialpha> <incX> <incY> <rout> <author> [\
```

4.1.3 Things you can assume, and/or need to know when writing kernels

First, ATLAS will ensure that `N > 0` for all routines. Vector strides will never be zero, and for those routines taking only one vector, the vector stride will always be positive.

For routines taking more than one vector, this is not so simple. The first thing that needs to be clear is that internally, ATLAS does not use the pointer conventions that the BLAS do when it comes to handling vectors with negative increments. In the BLAS, if you want to operate on a vector with a negative increment, you pass in the `end` of the negatively incremented vector (i.e, you always pass the part of the vector closest to 0 in memory, regardless of the increment). ATLAS will instead pass the beginning of the vector regardless of sign, so that the following sample code would correctly step through a vector regardless of the sign:

```
for (i=0; i < N; i++, X += incX, Y += incY) ...
```

.

ATLAS also fiddles with the vector increments so that:

1. Both increments are never negative
2. `incY` will be negative only if |`incX`| == 1 **and** |`incY`| != 1.

There may be additional constraints, on an individual routine basis.

4.2 Testing a kernel

Testing is straightforward. If your BLAS routine is `<blas>`, and your file in the `/<blas>` subdir is `<rout>`, then you compile and test it by:

```
make <pre><blas>test urout=<rout>
```

On the makefile line, you can also pass `N=<N>` to test a particular length vector. Each tester has various flags that can be passed to them, and these can be passed to make via the `opt="flags"` macro.

To find out what flags a tester takes, compile it, and run the resulting executable with `-help` (or just look in the source, you slacker).

Alright, here's a specific example. Let's say I have just written a new AXPY implementation and put it in `AXPY/myaxpy.c`. I can then test it with:

```
make daxpytest urout=myaxpy.c
```

When I fire this off, it does a bunch of compilation, and then spits out something like:

```
  ITST         N     alpha  incX  incY    TEST
======  ========  ========  ====  ====  ======
     0       777      1.00     1     1  PASSED
     1       777     -1.00     1     1  PASSED
     2       777      0.90     1     1  PASSED
ALL AXPY SANITY TESTS PASSED.
```

Now, let's say I've written a complex routine (cmyaxpy), that can handle multiple increments. To test various increments, I do:

```
speedy. make zaxpytest urout=cmyaxpy.c opt="-X 4 -1 1 -2 3 -Y 4 1 -1 2 -3"
  ITST         N    ralpha   ialpha  incX  incY    TEST
======  ========  ======== ========  ====  ====  ======
     0       777      1.00     0.00    -1     1  PASSED
     1       777     -1.00     0.00    -1     1  PASSED
     2       777      1.30     0.00    -1     1  PASSED
     3       777      0.90     1.10    -1     1  PASSED
     .....
     .....
    61       777     -1.00     0.00     3    -3  PASSED
    62       777      1.30     0.00     3    -3  PASSED
    63       777      0.90     1.10     3    -3  PASSED
ALL AXPY SANITY TESTS PASSED.
```

4.3 Timing a kernel

IMPORTANT NOTE: At present, all of ATLAS Level 1 timing is completely inaccurate for short vectors. Both the level 1 timing in `ATLAS/bin`, and the kernel timers described here screw this up. Essentially, our portable cache flushing mechanisms are not complete enough to get things completely out of cache, so you see that short vectors appear to get better performance than long vectors, a patent impossibility if all caches were correctly flushed. The only "solution" we have at the moment is to time vectors that themselves overflow the cache (i.e., N=1000000).

Timing works a lot like testing. If your BLAS routine is `<blas>`, and your file in the `/<blas>` subdir is `<rout>`, then you compile and time it by:

```
make <pre><blas>case urout=<rout>
```

So, to time the previously meantioned `myaxpy.c` with a million length vector, you'd have the following:

```
speedy. make daxpycase urout=myaxpy.c N=1000000
      N=1000000, tim=5.000000e-02
      N=1000000, tim=5.000000e-02
      N=1000000, tim=5.000000e-02
   N=1000000, time=5.000000e-02, mflop=40.000000
N=1000000, incX=1, incY=1, mflop = 40.000000
```

So, we got 40Mflop out of that implementation. All the timers take the flag `-C <cache flush size in bytes>`, which you can use to get the L1 (or L2) contained performance. Eg, the timer is usually doing it's best to flush the caches, but I want to see what performance I get when in the L1 cache. What I do is:

```
make daxpycase urout=myaxpy.c N=500 opt="-C 512"
      N=500, tim=4.296302e-06
      N=500, tim=3.938276e-06
      N=500, tim=4.296302e-06
   N=500, time=4.176960e-06, mflop=239.408571
N=500, incX=1, incY=1, mflop = 239.408571
```

Flushing 512 bytes is not going to do anything, and N=500 will not overflow cache, so we see that L1-contained operations come in at 240Mflop . . .

When it comes to timing codes with varying increments, the timer is not as flexible as the tester. It can time only one given increment value at a time. So, to time axpy with incX=3 and incY=4, we'd do:

```
make daxpycase urout=myaxpy N=500 opt="-X 3 -Y 4"
```

4.4 What to do if you are writing in assembler

If your kernel is written in gas assembler, you can tell the tester and timer that by setting the `<pre>UCC` and `<pre>UCCFLAGS` appropriately. For instance, to test the `DDOT` code given in Section A.1.3, you would issue:

```
make ddottest dUCC=gcc dUCCFLAGS="-x assembler-with-cpp" urout=ddot.c
```

4.5 Ramblings on special cases

When presented with these options, you may be tempted to ask what cases you should optimize. On the other hand, you might also validly ask: isn't it obvious that it is always `incX = incY = 1`, and why did you bother building in this special case flexibility?

First, it must be acknowledged that on most systems, bandwidth contraints will make any non-unit optimization tough. Doesn't mean it can't be done, though, at least to some degree.

For an example of why you want some flexibility, consider the COPY routine. Of course, the most optimizable routine is `incX = incY = 1`. However, one big use of the routine I make myself is to copy from noncontiguous storage to unit stride, so that more efficient access is possible. This suggests that the ability to make `incX` arbitrary and `incY = 1` might be useful in this routine.

For an example of a non-unit fixed stride, think of doing conjugation on a complex vector. That is essentially a real SCAL, with `incX = 2` and `alpha = -1.0`, and if this was important to you, you could optimize that exact case.

Ultimately, one can never foresee when flexibility will be needed, anyway. With the present case, a user that knew he was accessing a vector with increments of 50, 100, 500 all the time, could create special cases for them . . .

All that said, incX=incY=1 is often the only case where optimization will have a noticable effect, so it's where I'd concentrate my efforts as long as there's not some reason to do otherwise.

4.6 Notes for ROTG, ROTMG, ROT, ROTM

1. ROTG is essentially a scalar routine, so I didn't feel it was worth optimizing beyond just not writing it in a non-braindead way in the first place.

2. As for the rest of the routines, OK, so I haven't supported speeding up any of them yet, sue me. I myself don't use'em, and I couldn't write the testers without scoping the code, so they got a pass. Let me know if you call these guys, maybe I'll get off my duff . . .

4.7 Notes for ROT

1. Applies plane rotation.

2. An index file input line looks like:

   ```
   <ID> <C> <S> <incX> <incY> <source file> <author name>
   ```

3. Will never be called with $C = 1, S = 0$ (ROT is no-op for this case)

4. For complex, if both vectors have unit stride, the real routine is called with `2*N`.

4.8 Notes for ASUM

1. Performs $asum \leftarrow ||re(x)||_1 + ||im(x)||_1$

2. An index file input line looks like:

   ```
   <ID> <incX> <source file> <author name>
   ```

3. For complex, if both vectors have unit stride, the real routine is called with `2*N`.

4.9 Notes for AXPBY

1. Performs $y \leftarrow \alpha x + \beta y$

2. An index file input line looks like:

   ```
   <ID> <alpha> <beta> <incX> <incY> <source file> <author name>
   ```

3. The kernel will never be called with `beta == 1.0` (AXPY is called instead).
4. The kernel will never be called with `beta == 0.0` (CPSC is called instead).
5. The kernel will never be called with `alpha == 0.0` (SCAL is called instead).
6. For complex, if both vector strides are 1, and the imaginary components of alpha and beta are both zero, the real kernel is called with `2*N`.

4.10 Notes for AXPY

1. Performs $y \leftarrow \alpha x + y$
2. An index file input line looks like:

```
<ID> <alpha> <incX> <incY> <source file> <author name>
```

3. The kernel will never be called with `alpha == 0.0`.
4. For complex, if both vector strides are 1, and the imaginary component of alpha is zero, the real kernel is called with 2*N.

4.11 Notes for COPY

1. Performs $y \leftarrow x$
2. An index file input line looks like:

```
<ID> <incX> <incY> <source file> <author name>
```

3. For complex with both vector strides 1, the real kernel is called with 2*N.
4. `incX` arbitrary `incY=1` is interesting since it represents copying to contiguous storage.

4.12 Notes for CPSC

1. Performs $y \leftarrow \alpha x$
2. An index file input line looks like:

```
<ID> <alpha> <incX> <incY> <source file> <author name>
```

3. Never called with α of zero (SET is called instead).
4. Never called with α of one (COPY is called instead).
5. For complex, if both vector strides are 1, and the imaginary component of α is zero, the real kernel is called with 2*N.
6. `incX` arbitrary `incY=1` is interesting since it represents copying to contiguous storage.

4.13 Notes for DOT

1. Performs $dot \leftarrow x^T y$ or $dot \leftarrow x^H y$
2. An index file input line looks like:

   ```
   <ID> <incX> <incY> <source file> <author name>
   ```

3. For complex, `DOTU`'s index file is `[c,z]cases.dsc`, while `DOTC`'s is `[c,z]casesc.dsc`.
4. During compilation of the DOTC kernels, ATLAS throws the compilation flag `-DConj_` (this is so both nonconjugate and conjugate cases can come from the same file).

4.14 Notes for IAMAX

1. Performs $amax \leftarrow 1^{st} k \ni |re(x_k)| + |im(x_k)|$
2. An index file input line looks like:

   ```
   <ID> <incX> <source file> <author name>
   ```

4.15 Notes for NRM2

1. Performs $nrm2 \leftarrow ||x||_2$
2. An index file input line looks like:

   ```
   <ID> <incX> <source file> <author name>
   ```

3. For complex, if both vectors have unit stride, the real routine is called with `2*N`.
4. The user should be aware that this routine was originally added to the BLAS mostly for *accuracy* concerns, not for optimization. The simple implementation of this routine:

   ```
   for (nrm2=0.0, i=0; i < N; i++, X += incX) nrm2 += *X * *X;
   return(sqrt(nrm2));
   ```

   will needlessly overflow for half of the exponent range. Thus, a good implementation must either use extended precision or a scaling algorithm (such as the sum of squares used in the reference BLAS) to prevent overflow in the squaring process.

4.16 Notes for SCAL

1. Performs $x \leftarrow \alpha x$
2. An index file input line looks like:

   ```
   <ID> <alpha> <incX> <source file> <author name>
   ```

3. If `alpha` is zero, the ATLAS internal routine `ATL_set` is called.
4. For complex, if `incX == 1`, and the imaginary component of `alpha` is zero, the real routine is called with `N*2`.
5. The `incX == 2`, `alpha == -1.0` real cases can be used for complex conjugation.

4.17 Notes for SET

1. Performs $x \leftarrow \alpha$
2. An index file input line looks like:

   ```
   <ID> <alpha> <incX> <source file> <author name>
   ```

3. For complex, if `incX == 1`, and the imaginary component of `alpha` is the same as the real component, the real routine is called with `N*2`.
4. `alpha == 0.0` is heavily used in some codes.

4.18 Notes for SWAP

1. Performs $x \leftrightarrow y$
2. An index file input line looks like:

   ```
   <ID> <incX> <incY> <source file> <author name>
   ```

3. For unit stride case, the complex routine calls the real equivalent with `N*2`.
4. The main use of this routine that I am aware of is row-swapping in factorizations, so this is one routine that it would make sense to optimize the arbitrary increment cases, if it were possible.

4.19 Getting your new kernel used

Once you have a kernel that beats the present offerings, and you have updated the appropriate index file(s), nothing could be easier. If you are starting from a fresh install without Level 1 arch defaults, you need do nothing further: the install process will find your kernel.

Things are almost as simple if you have already installed, and need to force a redo. First, get rid of old search results by issuing `rm BLDdir/tune/blas/level1/res/<pre><blas>_SUMM`. Then, go to `BLDdir/src/blas/level1/`, and type :

```
rm Make_<pre><blas>
make Make_<pre><blas>
make <pre>lib
```

and that should do it.

So, to do that for the omnipresent DAXPY, on my PIII, I'd do:

```
cd BLDdir
rm tune/blas/level1/res/dAXPY_SUMM.
cd ATLAS/src/blas/level1/
rm Make_daxpy
make Make_daxpy
make dlib
```

4.19.1 Details I doubt you care about

The executable that tests and times all input kernels and chooses a winner to be included in ATLAS is `x<blas>srch`. So, for instance, the search executable for AXPY can be built by typing `make xaxpysrch` in `BLDdir/tune/blas/level1/`.

Each BLAS's search routine creates a file `res/<pre><blas>_SUMM`. All summation file have the same format, which is:

```
<# of specific cases>
<ID> <ialpha> <ibeta> <incX> <incY> <rout> <auth>
....
<ID> <ialpha> <ibeta> <incX> <incY> <rout> <auth>
```

For instance, if you want to see what went into making your DAXPY, you'd

```
speedy. cat res/dAXPY_SUMM
2
  3   2   2   1   1 axpy32_x1y1.c "R. Clint Whaley"
  1   2   2   0   0 axpy1_x0y0.c "R. Clint Whaley"
```

So, we see that the search has found two routines to build DAXPY out of. For `incX = incY = 1`, it will call `axpy32_x1y1.c`, and for all other cases, it will call `axpy1_x0y0.c`

5 Using special compilers and flags for kernel compilation

5.1 Specifying all new compilers and flags

For all BLAS kernels supported by ATLAS, you can specify a particular compiler and flags to be used in compiling your kernel. Normally, the default compiler specified by config is used. To override this behavior, you simply end a primitive line in the particular description file with the backslash character '\', and then the next two lines are assumed to contain your compiler, and the flags to use, respectively. For instance, Let us say you start with a simple gemm description file like:

```
2
300 480    4    4    1 1 1 4 4 2 ATL_mm4x4x2US.c "V. Nguyen & P. Strazdins"
301   8    4    4    2 1 1 4 4 2 ATL_mm4x4x2US_NB.c "V. Nguyen & P. Strazdins"
```

You then decide you want the first routine compiled with gcc, and some ultrasparc-specific flags. This file would change to:

```
2
300 480    4    4    1 1 1 4 4 2 ATL_mm4x4x2US.c "V. Nguyen & P. Strazdins" \
gcc
-O -mcpu=ultrasparc -mtune=ultrasparc -fno-schedule-insns -fno-schedule-insns2
301   8    4    4    2 1 1 4 4 2 ATL_mm4x4x2US_NB.c "V. Nguyen & P. Strazdins"
```

5.2 Specifying additional flags for the default compiler

At present, only the Level 2 kernels have this option. It's the same as when you specify a new compiler and flags, but you specify the compiler as '+', and the flags you give on the following flagline are *added* to its default flags. This can be useful for compiling the same routine multiple times with differing macros (eg., prefetch distance, etc.). Let us say you had this simple gemv description file:

```
2
  1  8  0  0 ATL_gemvN_mm.c     "R. Clint Whaley"
  2  0  1  1 ATL_gemvN_1x1_1.c  "R. Clint Whaley"
2
101 8  0  0 ATL_gemvT_mm.c      "R. Clint Whaley"
102 0  2  8 ATL_gemvT_2x8_0.c   "R. Clint Whaley"
```

You now want to compile `ATL_gemvT_2x8_0.c` two different ways: one with a default prefetch distance, and once with a prefetch distance of 16. This would be simply:

```
2
  1  8  0  0 ATL_gemvN_mm.c     "R. Clint Whaley"
  2  0  1  1 ATL_gemvN_1x1_1.c  "R. Clint Whaley"
2
101 8  0  0 ATL_gemvT_mm.c      "R. Clint Whaley"
102 0  2  8 ATL_gemvT_2x8_0.c   "R. Clint Whaley"
103 0  2  8 ATL_gemvT_2x8_0.c   "R. Clint Whaley" \
+
-DPREFETCH_DIST=16
```

5.3 Using a binary kernel

All the ATLAS machinery assumes you are using a C source, or something that can be made to look like one (i.e. assembler done as in-line assembler in C). Not all ways of programming can be made to work this way, so ATLAS has the ability to take an object-only kernel. The level 1 kernels do not have this option, but the Level 2 and 3 do.

This ability is an ugly hack on the pre-existing machinery, so don't expect either elegance or ease of use with this option: it is a last-resort kind of thing at best.

ATLAS has a routine that it builds called `./xccobj`, which is a "compiler" for object files. Essentially, what it does is fake compilation, but instead it simply moves a pre-compiled .o into place during the install/tuning process. You can see the source for this routine in `ATLAS/bin/ccobj.c`.

This routine reads in some special "compiler flags" that you pass it in order to figure out what to do. The most important of these are:

- `-o <outfile>`: Same as with a compiler.
- `--objdir <objdir>` The directory where the object file to be copied resides. The default is `ATLAS/blas/gemm/CASES/objs`.

- `--name <name>` The base name of the object file to be copied. This name is suffixed by the appropriate beta suffix, as determined by the usual BETA macro definition. Name has no usuable default, so it must be specified.
- `-DBETA` or `-DATL_BETA`. These are standard macros used to specify which BETA case a particular kernel can handle. According to this definition, a beta suffix will be added to the `--name` basename to find the object file to copy (this is to simulate all the beta cases coming from the same C code, as in the normal case). The values, along with their suffixes are:
  - `-DATL_BETA0` ⇒ `_b0`
  - `-DATL_BETA1` ⇒ `_b1`
  - `-DATL_BETAX` ⇒ `_bX`
  - `-DATL_BETAXI0` ⇒ `_bXi0`

  The default is `ATL_BETA1`.
- `-DConj_`: If this macro definition is found, the name of the routine to be copied is actually `<name>c_<beta>`, and that routine should supply the conjugate version of the kernel (simulates recompiling the same source in order to get both normal and conjugate versions of routines). The default is no conjugation. Note that this macro is set automatically by the build process, so the user would only set it manually when testing and timing by hand.

5.3.1 Using a binary kernel from the command line

OK, let us say you have a kernel object file, called `new_b1.o` which you have compiled/assembled/created using the magnetic field of the earth. You want to test and time it on the command line. For the DGEMM kernel, this would be simply:

```
make ummcase mmrout=CASES/ATL_objdummy.c DMC=./xccobj \
     DMCFLAGS="--name new" nb=30 beta=1
make mmutstcase mmrout=CASES/ATL_objdummy.c DMC=./xccobj \
     DMCFLAGS="--name new" nb=30 beta=1
```

Note that `ATL_objdummy.c` could be any file that exists in the indicated directory (does not need to be a compilable file), and that we do not specify the beta case (done using the tester/timer machinery) or the `objdir` (because we placed the kernel in the default directory).

5.3.2 Using a binary kernel from input file

Here's an example of an object kernel primitive line from a real gemm kernel descriptor file:

```
315   8 -30 -30 -30 0 4 30 30 30 ATL_objdummy.c "Julian Ruhe" \
./xccobj
--name julian2
```

This guy assumes that in the directory `ATLAS/tune/blas/gemm/CASES/objs`, you have these three files:

1. julian2_b0.o
2. julian2_b1.o
3. julian2_bX.o

6 A quick reference to ATLAS programming resources

ATLAS has a quite a bit of programming infrastructure which can be used. The routines in `ATLAS/src/auxil` represent low-level routines which can be called from anywhere in the ATLAS code (other than the Level 1 kernel routines), and are prototyped by including either `atlas_misc.h` or `atlas_aux.h`.

ATLAS makes fairly heavy use of macros in order to achieve something approaching precision (and in some cases type) independent code. Any routine can include the general macro file `atlas_misc.h` in order to get access to these macros. Low level routines are compiled multiple times with differing makefile-controlled cpp macros (call these *index macros*) in order to produce differing implementations. The index macros used in ATLAS presently include:

- Type/precision selection macros, choices are:
  1. `SREAL`: single precision real is expected
  2. `DREAL`: double precision real is expected
  3. `SCPLX`: single precision complex is expected
  4. `DCPLX`: double precision complex is expected
- Scalar alpha case selection macros, choices are:
  1. `ALPHA1`: scalar α should be assumed to be 1
  2. `ALPHAXI0`: only valid for complex types; real component is unknown, but imaginary component is zero
  3. `ALPHAX`: scalar alpha is unknown, and must be applied as a variable
- Scalar β case selection macros, choices are:
  1. `BETA0`: scalar beta should be assumed to be 0
  2. `BETA1`: scalar beta should be assumed to be 1
  3. `BETAXI0`: only valid for complex types; real component is unknown, but imaginary component is zero
  4. `BETAX`: scalar beta is unknown, and must be applied as a variable
- For complex types only, ATLAS defines the index variable `Conj_` when the conjugation of the transpose setting is needed

Each of these index macros define a number of helper macros that go with them. `atlas_misc.h` should be examined for full details. For instance, if `SREAL` or `DREAL` are defined, the type-determinant macro `TREAL` is defined. Similarly, if `SCPLX` or `DCPLX` are defined, the type-determinant macro `TCPLX` is defined. For a simpler example, the precision macro defines `ATL_rone`, which corresponds to `1.0f` in single precision, and `1.0` in double. A great many of these helper macros are designed to be used to help in resolving names independant of type. In order to use these naming macros, we use the macro-joining function `Mjoin(Mac1, Mac2)`, which results in the joining of the `Mac1` and `Mac2`.

You can examine the GEMV kernel implementations for examples of how this works. In particular, ATLAS uses the same implementation for single and double precision by using these macros, and recompiling the same source with differing index macros. Just to give a quick example, the index macro controlling β defines a helper macro `BNM` which corresponds to the correct beta name for the gemv and ger kernels. Using this trick, we can reduce:

```
#ifdef BETA0
   #ifdef Conj_
      void ATL_dgemvNc_a1_x1_b0_y1
   #else
      void ATL_dgemvN_a1_x1_b0_y1
   #endif
#elif defined (BETA1)
   #ifdef Conj_
      void ATL_dgemvNc_a1_x1_b1_y1
   #else
      void ATL_dgemvN_a1_x1_b1_y1
   #endif
#elif defined (BETAXI0)
   #ifdef Conj_
      void ATL_dgemvNc_a1_x1_bXi0_y1
   #else
      void ATL_dgemvN_a1_x1_bXi0_y1
   #endif
#else
   #ifdef Conj_
      void ATL_dgemvNc_a1_x1_bX_y1
   #else
      void ATL_dgemvN_a1_x1_bX_y1
   #endif
#endif
```

to:

```
#include "atlas_misc.h"

#ifdef Conj_
   void Mjoin(Mjoin(ATL_dgemvNc_a1_x1,BNM),_y1)
#else
   void Mjoin(Mjoin(ATL_dgemvN_a1_x1,BNM),_y1)
#endif
```

6.1 ATLAS's prefetch header file

ATLAS has an include file (`ATLAS/include/atlas_prefetch.h`), which defines the following macros:

- `ATL_GOT_L1PREFETCH`: This macro is only defined if ATLAS has a way of doing Level 1 cache prefetch for your architecture.

- `ATL_pfl1R(mem)` This macro prefetches the address pointed at by `mem`. It prefetches one cacheline into L1. The cacheline length will vary by platform, and the include file does not define it at this time (so sue me). This prefetch is optimized for read access. If your system does not possess level 1 prefetch, this macro will define to nothing (i.e., it will not be an error to call it).
- `ATL_pfl1W(mem)` This macro prefetches the address pointed at by `mem`. It prefetches one cacheline into L1. The cacheline length will vary by platform, and the include file does not define it at this time (so sue me). This prefetch is optimized for write access. If your system does not possess level 1 prefetch, this macro will define to nothing (i.e., it will not be an error to call it).

These prefetch instructions are not the same as user read, and do not cause access errors (i.e., you can prefetch the NULL pointer).

7 Conclusion

As you have seen, this note and the protocols it describes have plenty of room for improvement. Now, as the end-user of this function, you may have a naturally strong and negative reaction to these crude mechanisms, tempting you to send messages decrying my lack of humanity, decency, and legal parentage to the atlas or developer mailing lists. While this is quite understandable, keep in mind that I'm not a very understanding guy, and am also a gigantic baby that pouts when my tender feelings are hurt. So, the proper bitch format involves

- *First* thanking me for spending time in hell getting things to their present crude state
- *Then*, supplying your constructive ideas

For a higher overview of things, scope out the paper `ATLAS/doc/atlas_over.ps`, which describes the ideas behind ATLAS, including several that are touched upon only lightly in this note.

A Some notes on using assembly

This appendix contains useful information for people wanting to supply assembly kernels. Presently, ATLAS config detects seven styles of assembler, all using gnu's gcc/gas:

1. **x86-32**: Classic x86 assembler (AT&T syntax)
2. **x86-64**: Assembler for 64-bit mode of x86-64 (AT&T syntax).
3. **Sparc**: Sparc assembler.
4. **OS X PowerPC**: PowerPC assembly using OS X's ABI.
5. **Linux PowerPC**: PowerPC assembly using Linux's ABI.
6. **HP-UX PA-RISC**: PA-RISC assembly using HP-UX's commands.
7. **Linux PA-RISC**: PA-RISC assembly using Linux's commands.

The following sections give some hints that may be helpful in writing assembly for each of these variants. One trick to keep in mind, when you are unsure of syntax or how to do something, is to write the code in C, and use gcc to convert your C implementation to assembler (`gcc -S`), and see how gcc does it.

A.1 Some notes on x86-32 assembler

All assembly kernels presently in ATLAS utilize gcc for compilation. This is because gcc is required by the architectural defaults for all x86 archs, and is freely and widely available. Gcc/gas uses an AT&T style assembler syntax, which may be a bit confusing for people used to the MASM/NASM style. People tend to talk like this is a big deal, but I learned x86 assembler from a MASM-style book, and was working in AT&T immediately. The main difference is that MASM has a style of `DEST, SOURCE`, while AT&T uses `SOURCE, DEST`. For concise description of MASM/AT&T differences, scope:

`http://www.gnu.org/manual/gas-2.9.1/html_mono/as.html#SEC198`

There are numerous examples of CPP-augmented assembler in ATLAS's kernels. Look in the relavant description file (eg., `ATLAS/tune/blas/gemm/dcases.SSE`, `ATLAS/tune/blas/level1/IAMAX/dcases.dsc`, etc.), and any kernel that mandates `gcc` as the compiler, with the flags of `-x assembler-with-cpp` is an example.

A x86-32 assembler kernel should contain the following CPP lines at the beginning of the file:

```
#ifndef ATL_GAS_x8632
   #error "This kernel requires gas x86-32 assembler!"
#endif
```

This quick error exit keeps a non-x86-32 assembler from generating hundreds or thousands of spurious error messages during install.

A.1.1 Register usage

x86-32 provides a grand total of eight integer registers, a full 7 of which are usable as integer registers!! Whatever will we do with such bounty. Table 3 summarizes these registers. Note

REGISTER	USAGE	CALLEE SAVE
%eax	integer return val	NO
%edx	dividend reg	NO
%ecx	count reg	NO
%ebp	optional frame pointer	YES
%ebx	local reg	YES
%esi	local reg	YES
%edi	local reg	YES
%esp	Stack pointer	YES
%st	fp return val, aliased with mmx	NO
%st1-7	scratch regs, aliased with mmx	NO
%mmx0-7	scratch MMX regs, aliased with fp stack	NO
%xmm0-7	scratch SSE/SSE2 regs	NO

Table 3: Possible x86-32 registers

that the MMX registers are only available on architectures possing the MMX ISA extensions, and that the `xmm` registers are only available on architectures implementing SSE or SSE2. It is also worth mentioning that all of the integer registers with the exception of the stack pointer (`%esp`) may be used as general purpose registers, in addition to their specialized usages. Finally, please note that the MMX registers and floating point are aliased, so they cannot be referenced at the same time.

Registers marked as CALLEE SAVE = YES must be explicitly saved by any functions which write them. Registers marked NO are scratch registers whose values need not be saved. EAX is used by intergral functions to return the value, and the floating point top-of-stack, `%st` returns floating point values. EBP is documented as base pointer, but I think most sane people just do all referencing from ESP (the stack pointer), and utilize EBP as an additional integer register.

As for floating point registers, the x86-32 has eight of them arranged in a pseudo-stack. A document this brief cannot explain this travesty to you if you don't already grok it. Go read the x87 FPU description in an assembler book, and come back when you are through crying.

A.1.2 The calling sequence and stack frame

When your function first gets control, the stack frame looks like:

	Caller's frame
	last arg
	⋮
%esp+4	1st arg
%esp	return address

To get an idea of how this can be used, let us say we have the following function:

```
int isum(int N, int *v);
```

So, we want to sum up the vector `v`, and return the sum. Let us say that we are going to use all 7 integer registers, and put `N` into `eax` and `v` into `edx`. Our function prologue would then look like:

```
#   int isum(int N, int *v)
.global isum
        .type   isum,@function
isum:
#
#       Save non-scratch registers
#
        subl    $16, %esp
        movl    %ebx, (%esp)
        movl    %ebp, 4(%esp)
        movl    %esi, 8(%esp)
        movl    %edi, 12(%esp)
#
#       Load N and v
#
        movl    20(%esp), %eax
        movl    24(%esp), %edx
```

Assuming we accumulated our integer sum into `ecx`, the function epilogue would then consist of:

```
#
#       Set return value
#
        movl    %ecx, %eax
#
#       Restore registers
#
        movl    (%esp), %ebx
        movl    4(%esp), %ebp
        movl    8(%esp), %esi
        movl    12(%esp), %edi
        addl    $16, %esp
        ret
```

A.1.3 A simple example

Here is the complete code for the simplist assembler implementation of a `DDOT` primitive (we are implementing the case where `incX` and `incY` are known to be `1`):

```
#
#  These macros show integer register usage
#
#define N       %eax
#define X       %edx
#define Y       %ecx

#
#double ATL_UDOT(const int N, const double *X, const int incX,
#                              const double *Y, const int incY)
.global ATL_UDOT
        .type   ATL_UDOT,@function
ATL_UDOT:
#
#       Load parameters
#
        movl    4(%esp), N
        movl    8(%esp), X
        movl    16(%esp), Y
#
#       Dot product starts at 0
#
        fldz
LOOP:
        fldl    (X)
        fldl    (Y)
        fmulp   %st, %st(1)
        addl    $8, X
        addl    $8, Y
        faddp   %st, %st(1)
        dec     N
        jnz     LOOP

        ret
```

Notice that because we are able to confine ourselves to the three scratch registers, we have an empty function prologue and epilogue (we do not save any registers or move the stack pointer).

A.2 Some notes on x86-64 assembler

x86-64 assembler is AMD's extension of the classic IA32 into 64 bits. It's expanded register set and associated calling sequence improvements make it a good deal easier to code.

A x86-64 assembler kernel should contain the following CPP lines at the beginning of the file:

```
#ifndef ATL_GAS_x8664
   #error "This kernel requires gas x86-64 assembler!"
#endif
```

This quick error exit keeps a non-x86-64 assembler from generating hundreds or thousands of spurious error messages during install.

A.2.1 Register usage

x86-64 has a much less claustrophobic 16 integer registers, as well as 16 SSE/SSE2 registers. It has the usual complement of 8 x87 registers, aliased with 8 MMX registers. The available x86-64 registers is summarized in Table 4.

REGISTER	USAGE	CALLEE SAVE
%rsp	Stack pointer	YES
%rbx	optional base pointer	YES
%rbp	optional frame pointer	YES
%rax	integer return val	NO
%rdi	1st int arg	NO
%rsi	2nd int arg	NO
%rdx	3rd int arg	NO
%rcx	4th int arg	NO
%r8	5th int arg	NO
%r9	6th int arg	NO
%r10	used to pass static chain pointer	NO
%r11	scratch reg	NO
%r12-15	callee-saved regs	YES
%xmm0-1	pass & return fp args	NO
%xmm2-7	pass fp args	NO
%xmm8-15	scratch regs	NO
%mmx0-7	scratch regs, aliased to fp stack	NO
%st	returns `long double` args aliased with mmx regs	NO
%st1-7	scratch regs, aliased with mmx	NO

Table 4: x86-64 register summary

A.2.2 The calling sequence and stack frame

When your function first gets control, the stack frame looks like:

	Caller's frame
	last overflow arg
	⋮
8(%rsp)	1st overflow arg
0(%rsp)	return address
-8(%rsp)	begin red zone (16-byte aligned)
-128(%rsp)	end of red zone

The "red zone" is an reserved area that is not modified by signal or interrupt handlers, and so may be used as the temporary area for leaf functions.

When arguments are passed, they are first passed in registers as summarized in Table 4. Only when all registers of a given type are used up are they passed on the stack (the "overflow" args above). Note that the 7th and later integral arguments will overflow, as will the 9th and later floating point arguments. All argument lengths are rounded up to 8 bytes (i.e., a 4-byte integer is passed in the `%edi` portion of the `%rdi` register, for instance), both in register passing and in stack passing.

A.3 Some notes on WOW64 assembler

This is Window's bizarre ABI for x86-64.

A x86-64 assembler kernel should contain the following CPP lines at the beginning of the file:

```
#ifndef ATL_GAS_WOW64
   #error "This kernel requires gas WOW64 assembler!"
#endif
```

This quick error exit keeps a non-WOW64 assembler from generating hundreds or thousands of spurious error messages during install.

A.3.1 Register usage

The available WOW64 registers are summarized in Table 5.

REGISTER	USAGE	CALLEE SAVE
%rsp	Stack pointer	YES
%rbp	optional frame pointer	YES
%rax	integer return val	NO
%rcx	1st int arg	NO
%rdx	2nd int arg	NO
%r8	3rd int arg	NO
%r9	4th int arg	NO
%r10-11	syscall/sysret	NO
%r12-15	callee-saved regs	YES
%rdi,rsi,rbx	callee-saved regs	YES
%xmm0	1st arg/fp ret	NO
%xmm1-3	fp args	NO
%xmm6-15	pass fp args	YES
%st	returns `long double` args aliased with mmx regs	NO
%st1-7	scratch regs, aliased with mmx	NO

Table 5: WOW64 register summary

A.3.2 The calling sequence and stack frame

When your function first gets control, the stack frame looks like:

	Caller's frame
	last overflow arg
	⋮
40(%rsp)	1st overflow arg
32(%rsp)	r9 home
24(%rsp)	r8 home
16(%rsp)	rdx home
8(%rsp)	rcx home
0(%rsp)	ret @

WOW64 has no red zone, and the stack pointer must always be 16-byte aligned. It's horrific argument passing has to be at least part of the WOW in the name: only the first four arguments are passed in registers, and the rest are passed through the stack. Therefore, if you make a call like: `bob(1, 2.0, 2, 3.0, 4);` then `rcx=1`, `xmm1=2.0`, `r8=2`, `xmm3=3.0` and 4 is the first overflow argument. Yes, really.

All argument lengths are rounded up to 8 bytes (i.e., a 4-byte integer is passed in the `%edi` portion of the `%rdi` register, for instance), both in register passing and in stack passing. They are put in the overflow area in order, without regard to type.

The caller does not have to save the regs to the "home" slots; they are available to the callee.

A.4 Some notes on Sparc assembler

A Sparc assembler kernel should contain the following CPP lines at the beginning of the file:

```
#ifndef ATL_GAS_SPARC
   #error "This kernel requires gas SPARC assembler!"
#endif
```

This quick error exit keeps a non-SPARC assembler from generating hundreds or thousands of spurious error messages during install.

A.4.1 Register usage

Table 6 shows the registers available on the sparc. For floating point registers, double precision only uses even numbered registers, with the next odd numbered register holding the second half of the 64-bit value. Single precision instructions can't use %f32-%f62. However, one trick to keep in mind is anytime you need to load two consecutive single precision values, these can be loaded to an even and next odd register with one `ldd`, which is more efficient than two `ld`.

The sparc has only eight globally visable integer registers, the **Global** registers. Applications are not allowed to use the system registers (%g5-%g7), not even if they are saved and restored. Note that in the 64-bit variants of the ABI, %g5 can be used, and need not be saved for **v9**, but must be saved for **v8plus**. The other 24 registers (**In**, **Out**, **Local**) are in a moving register window, which is moved by the `save` and `restore` commands. Note that `%g0` *can* be the target of an operation, but it's value will not be effected (eg., its value will remain 0, even if you were to `subcc %i2, 8, %g0`).

REGISTER	USAGE	CALLEE SAVE
Global Registers		
%g0 (%r0)	Always zero	NO
%g1-%g4 (%r1-%r4)	global regs	NO
%g5-%g7 (%r5-%r7)	system regs - no use	NO USE
Out Registers		
%o0 (%r8)	Return value from callee outgoing arg 1 to caller	NO
%o1-%o5 (%r9-%r13)	outgoing arg 2-6 to caller	NO
%o6/%sp (%r14)	Stack pointer	YES
%o7 (%r15)	scratch/@ of CALL inst.	NO
Local Registers		
%l0-%l7 (%r16-%r23)	scratch registers	YES
In Registers		
%i0-%i5 (%r24-%r29)	incoming arg 1-6	YES
%i6/%fp (%r30)	frame pointer	YES
%i7 (%r31)	return address - 8	YES
Floating Point Registers		
%f0-%f31	single prec regs even #'s double prec	NO
%f32-%f62	double prec regs	NO

Table 6: Sparc register summary

Note that as long as you use the `save` statement, all callee-saved registers are implicitly saved by the register window.

The caller's **Out** registers are the callee's **In** registers.

A.4.2 The calling sequence and stack frame

After the callee performs the `save` statement, the stack frame looks like:

[%fp]	Caller's frame
	variable size local scratch
[%sp+92]	(if needed) strg for wrd8 (if needed) strg for wrd7
[%sp+68]	storage for words 1-6
[%sp+64]	struct/union return ptr
[%sp]	ireg window save

32 bit ABI

[%fp+2047]	Caller's frame
	variable size local scratch
[%sp+176+2047]	(if needed) strg for wrd8 (if needed) strg for wrd7
[%sp+128+2047]	storage for words 1-6
[%sp+2047]	ireg window save

64 bit ABI

The callee's arguments are stored in the *caller*'s stack frame. Note that the address of `%sp` must be kept 8-byte aligned for the 32 bit ABI, and 16-byte aligned for 64 bit.

For the 64-bit frame, the 2047 offset is called the BIAS, and it allow system libs to know

which ABI is in effect (if 1 in last binary digit, 64 bit). Integers are stored in 64 bit slots, with the last 4 bytes holding the good value. Therefore, to load an int stored at 176 you do a `ldsw [%sp+2047+176+4], reg`.

For the 32-bit ABIs, floating point args are passed in the integer regs (2 regs for double) and floating point functions return their value in %f0.

For the 64 bit v9 ABI, there is a complicated memory-slot/register mapping which determines how things are passed in registers, as shown in 2 (note that after the callee executes the `save` statement, the %o registers will of course be the %i registers). Note that v9a is just v9 with VIS extensions, and v9b also includes some UltraSPARC-III extensions, so all v9 ABIs are the same for the information discussed here.

Memory	Intergral	Float	Double	Quad
%sp+BIAS+128	%o0	%f1	%f0	%f0
%sp+BIAS+136	%o1	%f3	%f2	
%sp+BIAS+144	%o2	%f5	%f4	%f4
%sp+BIAS+152	%o3	%f7	%f6	
%sp+BIAS+160	%o4	%f9	%f8	%f8
%sp+BIAS+168	%o5	%f11	%f10	
%sp+BIAS+176	n/a	%f13	%f12	%f12
%sp+BIAS+184	n/a	%f15	%f14	
%sp+BIAS+192	n/a	%f17	%f16	%f16
%sp+BIAS+200	n/a	%f19	%f18	
%sp+BIAS+208	n/a	%f21	%f20	%f20
%sp+BIAS+216	n/a	%f23	%f22	
%sp+BIAS+224	n/a	%f25	%f24	%f24
%sp+BIAS+232	n/a	%f27	%f26	
%sp+BIAS+240	n/a	%f29	%f28	%f28
%sp+BIAS+248	n/a	%f31	%f30	

Figure 2: Argument passing for 64-bit SPARC v9 ABI

A.5 Some notes on PowerPC assembler

There are three OSes that I know something about that do PowerPC assembler: AIX, OS X, and Linux. All three of these OSes use the same ABI for 64-bit assembly (64-bit PowerPC ELF ABI Supplement 1.7). For 32 bits, the ABIs for AIX and OS X are essentially the same, but Linux differs very slightly in register usage and substantially in the way the stack is defined.

The standard ATLAS include file defines the macros `ATL_AS_OSX_PPC`, `ATL_AS_AIX_PPC` and `ATL_GAS_LINUX_PPC`, which can be used like:

```
#ifndef ATL_AS_OSX_PPC
   #error "This kernel requires OS X PPC assembler!"
#endif
```

to guard against invalid compilation.

A.5.1 Register usage for 64 bit PowerPC

The register usage for PPC64 is given in Table 7.

REGISTER	USAGE	CALLEE SAVE
Integer Registers		
r0[1]	Used in prolog/epilog	NO
r1	Stack pointer	YES
r2	TOC pointer (reserved)	YES
r3	1st para/return val	NO
r4-r10	3-8th para	NO
r11	Environment pointer	NO
r12	Used by global linkage	NO
r13	**reserved** system thread ID	N/A
r14-31	Global int registers	YES
Floating Point Registers		
f0	Scratch reg	NO
f1-13	1-13th fp para	NO
f14-f31	Global fp regs	YES
Special Registers		
LR	Link register	YES
CTR	Count register	NO
XER	Fixed pt exception	NO
FPSCR	fp status & ctrl	NO
CR0-CR7	Condition reg fields, each 4 bits wide	2, 3, 4 : YES
Vector Registers		
v0-v1	scratch regs	NO
v2-v13	vec para regs	NO
v14-v19	scratch regs	NO
v20-v31	global vregs	YES
vrsave	(32 bits)	YES

Table 7: Register usage for 64-bit PowerPC assembly

.

A.5.2 The calling sequence and stack frame for 64 bit PowerPC

The 64-bit PowerPC ELF ABI defines a 288-byte red zone beneath the stack pointer which can be used by leaf functions in lieu of allocating their own stack frame. The stack frame is:

[1] r0 interpretated as 0 for many instructions, so don't use unless you are really sure of what you are doing!

	fp reg save area (optional)
	ireg save area (optional)
	VRSAVE save word (32 bits, optional)
	padding (optional)
	Local storage (optional)
48(r1)	Parameter area (>= 8 words)
40(r1)	TOC save area
32(r1)	Link editor doubleword
24(r1)	Compiler doubleword
16(r1)	Link register (LR) save
8(r1)	Condition register (CR) save
0(r1)	ptr to callee's stack

If the LR is changed, it is first saved to the LR save area, and similarly if any of the callee-saved condition register fields are modified, it must be saved to the CR save area.

A.5.3 Register usage for 32-bit OS X or AIX

REGISTER	USAGE	CALLEE SAVE
Integer Registers		
r0[1]	Used in prolog/epilog	NO
r1	Stack pointer	YES
r2	TOC pointer (reserved)	YES
r3	1st para/return	NO
r4	2nd para/return	NO
r5-r10	3-8th para	NO
r11	Environment pointer	NO
r12	Used by global linkage	NO
r13-31	Global int registers	YES
Floating Point Registers		
f0	Scratch reg	NO
f1-13	1-13th fp para	NO
f14-f31	Global fp regs	YES
Special Registers		
LR	Link register	YES
CTR	Count register	NO
XER	Fixed pt exception	NO
FPSCR	fp status & ctrl	NO
CR0-CR7	Condition reg fields, each 4 bits wide	2, 3, 4 : YES

[1] r0 interpretated as 0 for many instructions, so don't use unless you are really sure of what you are doing!

Note that if an OS X/AIX routine accepts a floating point register, the appropriate number of integer registers are skipped, even though the value is passed in a fp reg. Linux does not waste registers in this way.

A.5.4 The calling sequence and stack frame for 32-bit OS X/AIX

When control is passed to the called routine, the stack pointer points to the callee's frame, which looks like:

	fp reg save area (optional)
	ireg save area (optional)
	padding (optional)
	Local storage (optional)
24(r1)	Parameter area (>= 8 words)
20(r1)	TOC save area
16(r1)	Link editor doubleword
12(r1)	Compiler doubleword
8(r1)	Link register (LR) save
4(r1)	Condition register (CR) save
0(r1)	ptr to callee's stack

- Stack pointer must be quadword aligned
- Start of fp & ireg save areas always double word aligned
- Before calling another func, must save the LR in LR save area
- Have 224-byte (OSX) / 220-byte (AIX) red zone below stack ptr so leaf func frame

A.5.5 Register usage for 32 bit Linux

A.5.6 The calling sequence and stack frame for Linux

When control is passed to the called routine, the stack pointer points to the callee's frame, which looks like:

	fp reg save area (optional)
	ireg save area (optional)
	CR save area (optional)
	Local storage (optional)
8(r1)	Parameter area (optional)
4(r1)	Link register (LR) save
0(r1)	ptr to callee's stack

- Stack pointer is always 16-byte (quad-word) aligned.
- Stack pointer (sp) updated atomically via "store word with update".

[1] r0 interpretated as 0 for many instructions, so don't use unless you are really sure of what you are doing!

REGISTER	USAGE	CALLEE SAVE
Integer Registers		
r0[1]	Used in prolog/epilog	NO
r1	Stack pointer	YES
r2	TOC pointer (reserved)	YES
r3-r4	1/2 para and return	NO
r5-r10	3-8th integer para	NO
r11-r12	Func linkage regs	NO
r12	Used by global linkage	NO
r13	Small data area ptr reg	NO
r14-30	Global int registers	YES
r31	Global/environment ptr	YES
Floating Point Registers		
f0	Scratch reg	NO
f1	1st para / return	NO
f2-8	2-8th fp para	NO
f9-f13	Scratch reg	NO
f14-f31	Global fp regs	YES
Special Registers		
CR0-CR7	Condition reg fields, each 4 bits wide	2, 3, 4 : YES
LR	Link register	YES
CTR	Count register	NO
XER	Fixed pt exception	NO
FPSCR	fp status & ctrl	NO

- Any non-scratch reg f# must be saved to the fp reg save area at location 8*(32-#) from the previous frame (i.e., the register f31 is saved adjacent to previous ptr to callee's stack).
- Any non-scratch reg r# must be saved in the ireg save area 4*(32-#) bytes before the low-addressed end of the fp reg save area.
- Minimum stack frame callee save and LR save.
- No red zone is specifically mandated.
- If number if ipara is ≤ 8 and the number of fppara ≤ 8, then no values are stored in the parameter area, and if this is true for all calls made by the routine, the parameter area will be of size 0.
- Stack pointer must be quadword aligned
- If parameters are contained in registers, no space allocated in frame

A.5.7 Mixing OS X, AIX and Linux PPC assembler

Other than register usage and stack frame issues (which can be isolated into the prologue and epilogue sections of code), the only real difference between these assemblies lies in the way registers are addressed. Under Linux, all registers are addressed by their number only, whereas under OS X, integer registers are prefixed by `r`, and floating point registers are prefixed by `f`. Therefore, I usually write a routine with two different prologues and epilogues, and at the start of the file I do something like:

```
#if defined(ATL_GAS_LINUX_PPC) || defined(ATL_AS_AIX_PPC)
   #define r0 0
   #define r1 1
   ...
   #define r31 31

   #define f0 0
   #define f1 1
   ...
   #define f31 31
#endif
```

and then use the OS X-style of register naming.

Also, since I'm using cpp anyway, I use C style comments, to avoid problems with OS X and Linux having different comment characters.

A.6 Some notes on HP PA-RISC assembler

ATLAS presently supports only 32 bit PA-RISC assembly. Such routines should include:

```
#if !defined(ATL_LINUX_PARISC) && !defined(ATL_HPUX_PARISC)
   #error "This kernel requires PA-RISC assembler!"
#endif
```

This quick error exit keeps a non-parisc assembler from generating hundreds or thousands of spurious error messages during install.

A.6.1 Register usage

Table 8 shows the registers available on the PA-RISC. For floating point registers, single precision registers get additional modifier 'L' or 'R' for left or right 32 bits.

- Non-leaf functions must save %r2 before calling a func.
- single prec fp args are passed in right side of fp arg ptrs
- If single prec fp arg has no 'L' or 'R' suffix, 'L' is assumed

REGISTER	USAGE	CALLEE SAVE
%r0	Always zero	NO
%r1	general reg	NO
%r2 (%rp)	return ptr/address	NO
%r3-%r18	general regs	YES
%r19	shared lib link reg	NO
%r20-%r22	general regs	NO
%r23 (%arg3)	4th iarg	NO
%r24 (%arg2)	3rd iarg	NO
%r25 (%arg1)	2nd iarg	NO
%r26 (%arg0)	1st iarg	NO
%r27 (%dp)	RESERVED: global data pointer	NA
%r28 (%ret0)	func ret reg	NO
%r29	static link reg	NO
%r30 (%sp)	stack pointer	YES
%r31	general reg	NO
Floating Point Registers		
%fr0 / always zero	fp status reg	NO
%fr1-%fr3	exceptions regs	NO
%fr4-%fr7	fp arg regs	NO
%fr8-%fr11	fp regs	NO
%fr12-%fr21	fp regs	YES
%fr22-%fr31	fp regs	NO

Table 8: HPPA register summary

A.6.2 Calling sequence and stack frame

Stack pointer always kept 64-bit aligned, and it grows *upward*.

	:previous stack frame
	reg save
	local vars
-4*(N+9)(r30)	arg word N (opt)
-52(r30)	arg word 4 (optional)
-48(r30)	arg word 3 (required)
-44(r30)	arg word 2 (required)
-40(r30)	arg word 1 (required)
-36(r30)	arg word 0 (required)
-32(r30)	External Data/LT ptr
-28(r30)	External SR4/LT
-24(r30)	External RP
-20(r30)	Current RP
-16(r30)	Static Link
-12(r30)	Cleanup
-8(r30)	Reloc stub RP
-4(r30)	ptr to callee's stack

A.7 Some notes on MIPS assembler

ATLAS presently supports only the 64-bit ISAs (MIPS III and MIPS IV) using the ABIs discussed below, which seem to work for Linux and IRIX. This information is presently priliminary, but as far as I can tell, both IRIX and Linux use the same ABIs. The only difference appears to be in their comment character, which is easily worked around by using C-style comments (assuming you use cpp, as I do).

There seem to be a lot of MIPS ABIs, and different archs use different ones. As far as I know, Linux follows the SGI/IRIX ABIs, which can be sorted into the following catagories, based on the IRIX **cc** flag or **gcc**'s **-mabi=** flag:

-32 : this is the classic 32 ABI build for old MIPS I MIPS II 32 bit hardware. Has only 16 fp regs, and integer regs are 32 bits long. Not presently supported by ATLAS.

-o64 : Seems to be a 64-bit extension of **-32**, which is also not presently supported by ATLAS.

-64 : 64 bit ABI for MIPS III and MIPS IV ISA machines. Integer regs are 64 bits long, and 32 fp regs are allowed. Supported by ATLAS.

-n32 : 32 bit ABI for MIPS III and MIPS IV ISA machines. Integer regs are 64 bits long, and 32 fp regs are allowed. Has same register usage as **-64**, but integers, longs, and pointers all 32 bit.

NOTE: rest of this document describes my understanding of **-64** and **-n32** ABIs only. MIPS assembler routines in ATLAS should include:

```
#if !defined(ATL_GAS_MIPS)
   #error "This kernel requires MIPS assembler!"
#endif
```

This quick error exit keeps a non-MIPS assembler from generating hundreds or thousands of spurious error messages during install.

A.7.1 Register usage

Table 9 shows the registers available on MIPS.

Note that in parameter passing, you can only pass 8 args in registers, regardless of what type, and args always consume a register of the correct type, and cause a skip of the other type. So, here's the register passing of a simple routine with mixed arguments:

```
//          $f12    $5       $f14     $7
void bob(float s, int i, double d, int k);
```

A.7.2 Calling sequence and stack frame

Stack pointer always kept 16-byte aligned (8-byte aligned for the old unsupported **-32** ABI), and it grows downward:

REGISTER	USAGE	CALLEE SAVE
$0	Always zero	n/a
$1/$at	assemb temp	NO
$2-$3	integer return	NO
$4-$11	1st 8 iargs	NO
$12-$15	scratch	NO
$16-$23	saved regs	YES
$24-$25	scratch	NO
$26-$27	reserved for kernel	n/a
$28 ($gp)	global pointer	YES
$29 ($sp)	stack pointer	YES
$30	optional frame ptr	YES
$31	return address	NO
Floating Point Registers for -64		
$f0,$f2	fp func return vals	NO
$f1,$f3	scratch	NO
$f4-$f11	scratch	NO
$f12-$f19	1st 8 fp args	NO
$f20-$f23	scratch	NO
$f24-$f31	scratch	YES
Floating Point Registers for -n32		
$f0,$f2	fp func return vals	NO
$f1,$f3	scratch	NO
$f4-$f11	scratch	NO
$f12-$f19	1st 8 fp args	NO
$f21,23,25,27,29,31	scratch	NO
$f20,22,24,26,28,30	scratch	YES

Table 9: MIPS register summary

	Caller's frame
	⋮ remaining overflow args
16($sp)	1st overflow arg
$sp	16 bytes reserved for args passed in regs

Note that integers are promoted to 64 bits, and all optional parameters are allocated at least 64 bits, even if they are short.