User contribution to ATLAS

R. Clint Whaley *

September 12, 2001

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 L1 matmul, and the Level 2 routines may be sped up by improving the GER and GEMV kernels. 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.

Although users may contribute any code improvements they like, the most useful contributions will probably be machine specific optimizations. Most general optimizations will be handled by ATLAS's code generators (the Level 2 kernels at present do not have code generators, but they eventually will), but it is not planned to have the code generators produce machine-specific code. Thus, adding machine-specific prefetch instructions to a kernel can provide optimizations that ATLAS will get in no other way. Some clear targets for such machine-specific intervention are:

- Intel-specific Streaming SIMD Extensions (SSE) optimizations (also known as KNI instructions). Single precision routines can utilize factor 4 speedup of floating point peak, and even double precision routines should benefit from prefetch instructions.
- Same thing as above for AMD's 3DNOW instructions
- Same for Apple's Altivec vector unit.
- Use of various system-specific prefetch (eg., PowerPC, UltraSparcII, etc).

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.

2 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 `atlas-comm@cs.utk.edu` 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.

2.1 Signing up for the ATLAS mailing list

We have set up a mailing list that we hope contributers and interested parties will subscribe to, `atlas-comm@cs.utk.edu`. The regular ATLAS mailing list, `atlas@cs.utk.edu`, is a fixed mailing list, whose recipients are limited to the UTK contingent of the ATLAS project. On the other hand, `atlas-comm@cs.utk.edu`, is a dynamic list, that users can subscribe and unsubscribe to at will.

To subscribe to this mailing list, send mail to `majordomo@cs.utk.edu`, and include the line `subscribe atlas-comm` as the only body to the message. Unsubscribing is accomplished by sending `unsubscribe atlas-comm` instead.

2.2 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.

2.3 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?").

2.4 Inclusion in the ATLAS tarfile

If you submit an improvement to `atlas-comm@cs.utk.edu`, 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.

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.).

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.

5 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 the L1 matmul. The L1 matmul 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.

ATLAS actually has two different L1 matmul kernels: one for copied matrices, and one that operates directly on the user's matrices. 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 L1 matmul 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 L1 matmul. For most problems, well over 98% of ATLAS time is spent in the copy L1 matmul, so this should not be much of a problem.

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

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 the L1 matmul).

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 L1 matmul which operates on non-copied matrices (i.e. directly on the user's operands). The non-copy L1 matmul will generally not be as efficient as the copy L1 matmul; at this problem size the main drawback is the additional pointer arithmetic required in order to support the user-supplied leading dimension.

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

Figure 1 shows the necessary steps for computing a $N_B \times N_B$ section of C using the L1 matmul.

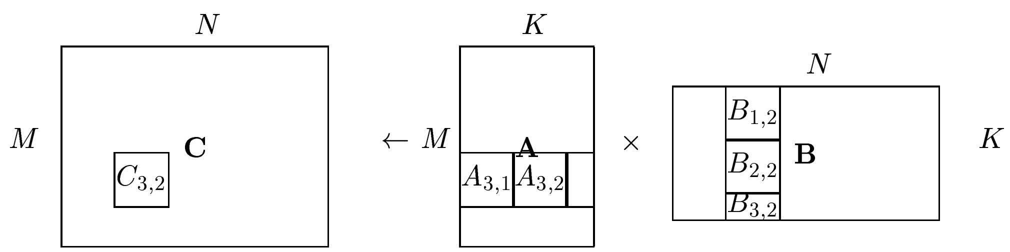

Figure 1: One step of matrix-matrix multiply

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

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

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.

5.2 The L1 matmul

So, there are actually three L1 matmul kernels (corresponding to different β values), and they are: $C \leftarrow A^TB$, $C \leftarrow A^TB + C$, $C \leftarrow A^TB + \beta C$. 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 $lda = ldb = M = N = K = N_B$.

5.2.1 L1 matmul macro definitions

In order to make writing an L1 matmul 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 L1 matmul, `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.

5.2.2 L1 matmul API

The L1 matmul 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`

5.2.3 L1 matmul description file

In the install process, ATLAS first searches through the L1 matmul 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 matmul search script opens a description file for each precision (`scases.dsc`, `dcases.dsc`, `ccases.dsc`, `zcases.dsc`) in the `ATLAS/tune/blas/gemm/<arch>` 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 5.2.4 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 L1 matmul. 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

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.

5.2.4 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: $200 \leq ID < 300$.

5.3 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 `ATLAS/tune/blas/gemm/<arch>`, 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, const double *B, const int ldb, const double beta, double *C, const int ldc)
{
   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"
```

5.4 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
```

5.5 Complex L1 matmul

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 L1 matmul stride C by 2. An example will solidify the confusion.

A simple 3-loop implementation of an ATLAS complex L1 matmul 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
```

5.6 Providing ATLAS with kernel cleanup code

As mentioned in Section 5.1, 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 5.6.5 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.

5.6.1 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.

5.6.2 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 5.6.5 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.

5.6.3 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.

5.6.4 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
```

5.6.5 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 L1 matmul 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 not be used anyway.

5.7 L1 matmul 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 L1 matmul 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.

5.8 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`), and `<arch>` is replaced by your architecture ID (eg., `Linux_PII`).

5.8.1 Putting it in by hand

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

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

5.8.2 With a fresh install

First, run config as usual. Then, tail the created `Make.<arch>` file, and see if the macro `ARCHDEF` is set. If so, cd to the directory pointed at by `ARCHDEF`, and then issue

```
rm <pre>u*
rm <pre>MMRES
```

Now, continue install as normal, and your kernel should be used if it beats what ATLAS is presently using.

5.8.3 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. Tail your `Make.<arch>` file, and see if the macro `ARCHDEF` is set. If so, cd to the directory pointed at by `ARCHDEF`, and then issue

```
rm <pre>u*
rm <pre>MMRES
```

From your ATLAS directory, then issue:

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

5.9 Contributing a complete GEMM implementation

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 5.1, 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.

5.9.1 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.<arch>. 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.

5.9.2 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 (where `<arch>` is replaced by the `ARCH` string of your `Make.<arch>`; eg, `Linux_PII`, `AIX_POWER3`, `OSF_21264`, etc.):

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 5.9.3, which discusses how to make ATLAS use a contributed GEMM that config has not setup for you

5.9.3 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.<arch>` 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 `ATLAS/src/blas/gemm/<arch>` type `make lib`
4. In `ATLAS/include/<arch>`, issue
   - `rm ?Xover.h atlas_cacheedge.h`
   - `touch altas_cacheedge.h sXover.h dXover.h cXover.h zXover.h`
5. In `ATLAS/tune/blas/gemm/<arch>`, 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`

6 Speeding up the Level 2 BLAS

All Level 2 BLAS are written in terms of 3 kernel routines:

1. Notranspose GEMV
2. Transpose GEMV
3. GER

For complex codes, these kernels must also supply conjugate cases.

6.1 Speeding Up GEMV, HEMV, SYMV, TRMV and TRSV

These routines are all based on GEMV. Therefore, to speed them up, the user needs to supply a more efficient GEMV primitive. The hand-coded GEMV primitives may be found in ATLAS/tune/blas/gemv/CASES.

6.1.1 The Kernel Description File

The most important file `ATLAS/tune/blas/gemv/CASES` is the primitive description file, `<pre>cases.dsc`. Each precision has its own description file (as indicated by `<pre>`), and this file describes all of the routines to time in order to find the best. For instance, for double precision, we see:

```
speedy. cat CASES/dcases.dsc
9
 8  0  0 ATL_gemvN_mm.c     "R. Clint Whaley"
 0  1  1 ATL_gemvN_1x1_1.c  "R. Clint Whaley"
16 32  1 ATL_gemvN_1x1_1a.c "R. Clint Whaley"
 0  4  2 ATL_gemvN_4x2_0.c  "R. Clint Whaley"
 0  4  4 ATL_gemvN_4x4_1.c  "R. Clint Whaley"
 0  8  4 ATL_gemvN_8x4_1.c  "R. Clint Whaley"
 0 16  2 ATL_gemvN_16x2_1.c "R. Clint Whaley"
 0 16  4 ATL_gemvN_16x4_1.c "R. Clint Whaley"
16 32  4 ATL_gemvN_32x4_1.c "R. Clint Whaley"
 6
 8  0  0 ATL_gemvT_mm.c      "R. Clint Whaley"
 0  2  8 ATL_gemvT_2x8_0.c   "R. Clint Whaley"
 0  4  8 ATL_gemvT_4x8_1.c   "R. Clint Whaley"
 0  4 16 ATL_gemvT_4x16_1.c  "R. Clint Whaley"
 0  2 16 ATL_gemvT_2x16_1.c  "R. Clint Whaley"
 0  1  1 ATL_gemvT_1x1_1.c   "R. Clint Whaley"
```

The first number (in this case 9) is the number of NoTranspose primitives to time. This is followed by that number of primitive lines describing those NoTrans primitives, and then we supply the number of Transpose primitives to time (in this example, 6), followed by that number of primitive lines describing the Transpose primitives.

As you can see, each line supplies three integers and a filename to the search routine. The filename is the filename of the primitive to time. The three integers supply information necessary in order for the higher level routines to do blocking.

This is the first piece of important information about these primitive routines: no blocking should be done in them. The appropriate blocking is done by higher level ATLAS routines. Most primitives employ some kind of loop unrolling, and when these higher level routines block in order to reuse vectors or matrices, it is important that this blocking does not conflict with the primitives' unrolling factors (for instance, if the primitive unrolls a given dimension by 8, but ATLAS blocks that dimension to 3, ATLAS would always call the cleanup code). So this is the information conveyed by these three integers.

The form of a GEMV primitive line is:

```
<flag> <Yunroll> <Xunroll> <filename> "<author(s)>"
```

As mentioned previously, `<filename>` is the primitive source file. `<Yunroll>` is the unrolling used for the loop that loops over the Y vector, and `<Xunroll>` is the unrolling used for the loop that loops over the X vector. `<flag>` is a less obvious parameter which is used to tell the search script about special properties of a kernel.

It is assumed that the user has supplied a "inner-product" based GEMV implementation (i.e., an implementation which basically does `<Yunroll>` simultaneous dot products). This default state is expressed to the search by a `<flag>` value of 0. However, since the inner product formulation of NoTranspose GEMV loops across the non-contiguous dimension of the matrix, some architectures need to employ an "outer-product" based NoTranspose GEMV (i.e., a GEMV which is performed by doing `<Xunroll>` simultaneous axpy's). This is indicated by a `<flag>` value of 16. Finally, since ATLAS's GEMM has a code generator which allows it to achieve very good portable performance, it is always worth seeing how optimal a GEMV can be obtained by simply making the appropriate call to GEMM. `<flag>` of 8 indicates that this is what the kernel is doing.

In summary:

FLAG	MEANING
0	Normal
8	GEMM-based primitive
16	Outer-product or AXPY-based primitive (only valid for Notranspose GEMV)

6.1.2 Writing a GEMV kernel

There are several assumptions that need to hold true for a user-supplied GEMV primitive. First, the loop ordering must be that implied by the `<flag>` setting the user supplies in the primitive description file, as discussed in Section 6.1.1. Each primitive makes assumptions about the arguments it handles, and these assumptions are reflected in the routine name. The function name of a GEMV primitive is:

```
ATL_<pre>gemv<Trans>_a1_x1_<betanam>_y1
```

where:

- `<pre>` is replaced by the precision prefix: `s`, `d`, `c`, or `z`.
- `<Trans>` is replaced by transpose specifier:
  - `N` : NoTranspose
  - `T` : Transpose
  - `C` : Conjugate Transpose
  - `Nc` : NoTranspose, with conjugation
- `<betanam>` is replaced by the beta specifier this kernel supplies. All GEMV kernels must supply the following beta specifiers and names:
  - `b0` : $\beta = 0$
  - `b1` : $\beta = 1$
  - `bXi0` : for complex GEMV only, specifies when $\beta \neq 0$ and $\beta \neq 1$, but the imaginary component is zero.
  - `bX` : beta is a input variable without known characteristics.

For a given gemv primitive (either NoTranspose or Transpose), if the cpp macro `Conj_` is defined we want the conjugate form of that transpose setting (i.e., `Nc` or `C`).

Each file is further compiled with differing cpp settings to generate the various beta cases. The beta macro settings and their meanings are:

CPP MACRO	MEANING
`BETA0`	Primitive should provide $y \leftarrow Ax$
`BETA1`	Primitive should provide $y \leftarrow y + Ax$
`BETAX`	Primitive should provide $y \leftarrow \beta y + Ax$
`BETAXI0`	For complex only, primitive should provide $y \leftarrow \beta y + Ax$, where the imaginary component of beta is zero.

In terms of the BLAS API, the GEMV kernels additionally assume

- $\alpha = 1$
- `incX = 1`
- `incY = 1`
- Column-major storage of A

Higher level ATLAS routines ensure these assumptions are true before calling the primitive.

Therefore, the routine:

```
ATL_dgemvN_a1_x1_b0_y1
```

supplies a primitive doing notranspose gemv, on a column-major array with $\alpha = 1$, $\beta = 0$, `incX` $= 1$ and `incY` $= 1$. while:

```
ATL_cgemvNc_a1_x1_bXi0_y1:
```

supplies a primitive doing notranspose gemv, on a column-major array whose elements should be conjugated before the multiplication, with $\alpha = 1$, `incX` $= 1$, `incY` $= 1$, and β whose real component is unknown, but whose imaginary component is known to be zero.

For greater understanding of how these CPP macros are used to compile multiple primitives from one file, examine the provided CASES files.

The API of the primitive is:

```
ATL_<pre>gemv<Trans>_a1_x1_<betanam>_y1
(
   const int M,        /* length of Y vector */
   const int N,        /* length of X vector */
   const SCALAR alpha,/* ignored, assumed to be one */
   const TYPE *A,      /* pointer to column-major matrix */
   const int lda,      /* leading dimension of A, or row-stride */
   const TYPE *X,      /* vector to multiply A by */
   const int incX,     /* ignored, assumed to be one */
   const SCALAR beta,  /* value of beta */
   TYPE *Y,            /* output vector */
   const int incY      /* ignored, assumed to be one */
);
```

where,

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

Note that the meaning of M and N are slightly different than that used by the Fortran77 API, in that they give the vector lengths, not array dimensions.

6.1.3 GEMV examples

Probably the simplest, notranspose GEMV kernel implementation is:

```
#ifdef BETA0
void ATL_dgemvN_a1_x1_b0_y1
#elif defined (BETA1)
void ATL_dgemvN_a1_x1_b1_y1
#else
void ATL_dgemvN_a1_x1_bX_y1
#endif
   (const int M, const int N, const double alpha, const double *A,
    const int lda, const double *X, const int incX, const double beta,
    double *Y, const int incY)
{
   int i, j;
   register double y0;

   for (i=0; i != M; i++)
   {
      #ifdef BETA0
         y0 = 0.0;
      #elif defined(BETA1)
         y0 = Y[i];
      #elif defined(BETAX)
         y0 = Y[i] * beta;
      #endif
      for (j=0; j != N; j++) y0 += A[i+j*lda] * X[j];
      Y[i] = y0;
   }
}
```

Saving this file to `ATLAS/tune/blas/gemv/CASES/ATL_dgemvN_1x1_1.c`, from the `ATLAS/tune/blas/gemv/<arch>` directory, we can test and time the implementation by:

```
make dmvtstcaseN mvrout=../CASES/ATL_dgemvN_1x1_1.c yu=1 xu=1
BEGINNING GEMV PRIMITIVE TESTING:

   TEST TA=N, M=997, N=177, lda=1004, beta=0.000000 STARTED
   TEST TA=N, M=997, N=177, lda=1004, beta=0.000000 PASSED
   TEST TA=N, M=997, N=177, lda=1004, beta=1.000000 STARTED
```

```
   TEST TA=N, M=997, N=177, lda=1004, beta=1.000000 PASSED
   TEST TA=N, M=997, N=177, lda=1004, beta=0.800000 STARTED
   TEST TA=N, M=997, N=177, lda=1004, beta=0.800000 PASSED


GEMV PRIMITIVE PASSED ALL TESTS

speedy. make dmvcaseN mvrout=../CASES/ATL_dgemvN_1x1_1.c yu=1 xu=1
      gemvN_0 : 49.484536 MFLOPS
      gemvN_0 : 49.484536 MFLOPS
      gemvN_0 : 49.230769 MFLOPS
   gemvN_0 : 49.40 MFLOPS
```

In the above examples, we pass `yu`, the unrolling along the y vector, and `xu`, the unrolling along the x vector, so that when ATLAS blocks the operation, it knows the correct unrolling to use to avoid cleanup code.

A similarly sophisticated transpose primitive is:

```
#ifdef BETA0
void ATL_dgemvT_a1_x1_b0_y1
#elif defined (BETA1)
void ATL_dgemvT_a1_x1_b1_y1
#else
void ATL_dgemvT_a1_x1_bX_y1
#endif
   (const int M, const int N, const double alpha, const double *A,
    const int lda, const double *X, const int incX, const double beta,
    double *Y, const int incY)

{
   int i, j;
   register double y0;

   for (j=0; j != M; j++)
   {
      #ifdef BETA0
         y0 = 0.0;
      #elif defined(BETA1)
         y0 = Y[j];
      #else
         y0 = Y[j] * beta;
      #endif
      for (i=0; i != N; i++) y0 += A[i+j*lda] * X[i];
      Y[j] = y0;
   }
}
```

Which we could test and time by:

```
speedy. make dmvtstcaseT mvrout=../CASES/ATL_dgemvT_1x1_1.c xu=1 yu=1
BEGINNING GEMV PRIMITIVE TESTING:

   TEST TA=T, M=997, N=177, lda=1004, beta=0.000000 STARTED
   TEST TA=T, M=997, N=177, lda=1004, beta=0.000000 PASSED
   TEST TA=T, M=997, N=177, lda=1004, beta=1.000000 STARTED
   TEST TA=T, M=997, N=177, lda=1004, beta=1.000000 PASSED
   TEST TA=T, M=997, N=177, lda=1004, beta=0.800000 STARTED
   TEST TA=T, M=997, N=177, lda=1004, beta=0.800000 PASSED


GEMV PRIMITIVE PASSED ALL TESTS

speedy. make dmvcaseT mvrout=../CASES/ATL_dgemvT_1x1_1.c xu=1 yu=1
      gemvT_0 : 37.647059 MFLOPS
      gemvT_0 : 37.647059 MFLOPS
      gemvT_0 : 37.647059 MFLOPS
   gemvT_0 : 37.65 MFLOPS
```

An unsophisticated notranspose GEMV implementation for double precision complex would be:

```
#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
```

```
   (const int M, const int N, const double *alpha,
    const double *A, const int lda, const double *X, const int incX,
    const double *beta, double *Y, const int incY)
{
   int i, j;
   const int M2 = M<<1, N2 = N<<1;
   #if defined(BETAX)
      const double rbeta = *beta, ibeta = beta[1];
   #elif defined(BETAXI0)
      const double rbeta = *beta;
   #endif
   register double ra, ia, rx, ix, ry, iy;

   for (i=0; i != M2; i += 2)
   {
      #ifdef BETA0
         ry = iy = 0.0;
      #elif defined(BETAX)
         rx = rbeta; ix = ibeta;
         ra = Y[i]; ia = Y[i+1];
         ry = ra * rx - ia * ix;
         iy = ra * ix + ia * rx;
      #else
         ry = Y[i];
         iy = Y[i+1];
         #ifdef BETAXI0
            rx = rbeta;
            ry *= rx;
            iy *= rx;
         #endif
      #endif
      for(j=0; j != N2; j += 2)
      {
         ra = A[i+j*lda]; ia = A[i+j*lda+1];
         rx = X[j]; ix = X[j+1];
         ry += ra * rx;
         iy += ra * ix;
         #ifdef Conj_
            ry += ia * ix;
            iy -= ia * rx;
         #else
            ry -= ia * ix;
            iy += ia * rx;
         #endif
      }
      Y[i] = ry;
```

```
      Y[i+1] = iy;
   }
}
```

Which, when saved to `ATLAS/tune/blas/gemv/CASES/ATL_zgemvN_1x1_1.c`, we could test and time by:

```
 make zmvtstcaseN mvrout=../CASES/ATL_zgemvN_1x1_1.c xu=1 yu=1

BEGINNING GEMV PRIMITIVE TESTING:

   TEST TA=N, M=997, N=177, lda=1004, beta=(0.000000,0.000000) STARTED
   TEST TA=N, M=997, N=177, lda=1004, beta=(0.000000,0.000000) PASSED
   TEST TA=-, M=997, N=177, lda=1004, beta=(0.000000,0.000000) STARTED
   TEST TA=-, M=997, N=177, lda=1004, beta=(0.000000,0.000000) PASSED
   TEST TA=N, M=997, N=177, lda=1004, beta=(1.000000,0.000000) STARTED
   TEST TA=N, M=997, N=177, lda=1004, beta=(1.000000,0.000000) PASSED
   TEST TA=-, M=997, N=177, lda=1004, beta=(1.000000,0.000000) STARTED
   TEST TA=-, M=997, N=177, lda=1004, beta=(1.000000,0.000000) PASSED
   TEST TA=N, M=997, N=177, lda=1004, beta=(0.800000,0.000000) STARTED
   TEST TA=N, M=997, N=177, lda=1004, beta=(0.800000,0.000000) PASSED
   TEST TA=-, M=997, N=177, lda=1004, beta=(0.800000,0.000000) STARTED
   TEST TA=-, M=997, N=177, lda=1004, beta=(0.800000,0.000000) PASSED
   TEST TA=N, M=997, N=177, lda=1004, beta=(0.800000,0.300000) STARTED
   TEST TA=N, M=997, N=177, lda=1004, beta=(0.800000,0.300000) PASSED
   TEST TA=-, M=997, N=177, lda=1004, beta=(0.800000,0.300000) STARTED
   TEST TA=-, M=997, N=177, lda=1004, beta=(0.800000,0.300000) PASSED


GEMV PRIMITIVE PASSED ALL TESTS

speedy. make zmvcaseN mvrout=../CASES/ATL_zgemvN_1x1_1.c xu=1 yu=1
      gemvN_0 : 78.688525 MFLOPS
      gemvN_0 : 78.688525 MFLOPS
      gemvN_0 : 78.688525 MFLOPS
   gemvN_0 : 78.69 MFLOPS
```

6.1.4 GEMV kernel notes

All routines except SYMV call the GEMV kernel in the same fashion. Other than SYMV, all routines cannot reduce the load of A from $O(N^2)$, but can reduce the memory access of both X and Y from $O(N^2)$ to $O(N)$. In general, the Y access is reduced by register blocking in the GEMV kernel. Therefore, the higher level routines block X such it is reused across kernel invocations in L1 (if you write a axpy-based notranspose GEMV kernel, Y is blocked instead of X). What this amounts to is partitioning X via: $N_p = \frac{S_1 - R_y}{R_y + 1}$, where S_1 is the size, in elements, of the Level 1 cache, N_p is the partitioning of X, and R_y corresponds to the

`Yunroll` of your input file. The equation is actually a little more complicated than this, as ATLAS may want to use less than the full S_1 to avoid cache thrashing and throwing useful sections away between kernel calls. However, this gives the user some idea of the importance of these parameters. In particular, it shows that `Yunroll` should not be allowed to grow too large, for fear of causing the X loop to be too short to support good optimization.

Also, note that after the first invocation of the kernel, X will come from L1, leaving A the dominant data cost.

At present, SYMV does a different blocking that blocks both X **and** Y (all other routines block only one dimension), so that A is reused between calls to the Transpose and NoTranspose kernels. This may eventually change as greater sophistication is achieved (as you might imagine, you get two very different GEMV kernels if one is expecting A from main memory, and the other expects A to come from L1, as in this case; this means we may at some time generate a specialized L1-contained GEMV kernel).

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

6.2 Speeding Up GER, GERU, GERC, HER, HER2, SYR and SYR2

All of these routines rely on the GER primitive for their performance. The hand-written primitives tried by ATLAS may be found in

```
ATLAS/tune/blas/ger/CASES.
```

Most of the discussion of the GEMV primitives applies to the GER primitives as well, so I assume you have read and are familiar with the concepts discussed above. As before, the routines to be timed are given in a kernel description file, `<pre>cases.dsc`. GER does not have a transpose case, so this file first lists the number of GER primitives to search, followed by that many primitive lines describing them.

GER primitive lines are of the form:

```
<flag> <Xunroll> <Yunroll> <filename> "<author(s)>"
```

- `<flag>`: is an integer flag which is ignored at the moment
- `<Xunroll>`: is the unrolling of the loop over the X vector (i.e. the M-loop)
- `<Yunroll>`: is the unrolling of the loop over the Y vector (i.e. the N-loop)
- `<filename>`: is the name of the C source file for the primitive.
- `<author(s)>`: author(s) name(s)

The API for the ger primitive is:

```
#if defined(SCPLX) || defined(DCPLX)
   #ifdef Conj_
      ATL_<pre>ger1c_a1_x1_yX
   #else
      ATL_<pre>ger1u_a1_x1_yX
   #endif
#else
```

```
   ATL_<pre>ger1_a1_x1_yX
#endif
   (
      const int M,       /* length of X vector */
      const int N,       /* length of Y vector */
      const SCALAR alpha,/* ignored, assumed to be one */
      const TYPE *X,     /* pointer to X vector */
      const int incX,    /* ignored, assumed to be one */
      const TYPE *Y,     /* pointer to Y vector */
      const int incY     /* increment of Y vector; NOTE: NOT IGNORED */
      TYPE *A,     /* pointer to column-major matrix */
      const int lda,     /* leading dimension of A, or row-stride */
   );
```

Assumptions:

- $\alpha = 1$
- `incX = 1`
- Column-major storage of A

6.2.1 GER examples

A simple double precision real implementation of GER would then be:

```
void ATL_dger1_a1_x1_yX(const int M, const int N, const double alpha,
                        const double *X, const int incX, const double *Y,
                        const int incY, double *A, const int lda)
{
   int i, j;
   register double y0;

   for (j=0; j < N; j++)
   {
      y0 = Y[j*incY];
      for (i=0; i < M; i++) A[i+j*lda] += X[i] * y0;
   }
}
```

Which we can test and time with:

```
speedy. make dr1tstcase r1rout=../CASES/ATL_dger1_1x1_1.c xu=1 yu=1
   TEST CONJ=0, M=997, N=177, lda=1006, incY=1, STARTED
   TEST CONJ=0, M=997, N=177, lda=1006, incY=1, PASSED
   TEST CONJ=0, M=997, N=177, lda=1006, incY=3, STARTED
   TEST CONJ=0, M=997, N=177, lda=1006, incY=3, PASSED
   TEST CONJ=0, M=997, N=177, lda=1006, incY=-3, STARTED
   TEST CONJ=0, M=997, N=177, lda=1006, incY=-3, PASSED
```

```
speedy. make dr1case r1rout=../CASES/ATL_dger1_1x1_1.c xu=1 yu=1
      dger_0 : 31.168831 MFLOPS
      dger_0 : 31.788079 MFLOPS
      dger_0 : 31.578947 MFLOPS
   dger_0 : 31.51 MFLOPS
```

A simple double precision complex implementation would be:

```
#ifdef Conj_
   void ATL_zger1c_a1_x1_yX
#else
   void ATL_zger1u_a1_x1_yX
#endif
   (const int M, const int N, const double *alpha, const double *X,
    const int incX, const double *Y, const int incY, double *A, const int lda)
{
   const int M2 = M<<1, N2 = N<<1;
   int i, j;
   register double ry, iy, ra, ia, rx, ix;

   for (j=0; j < N2; j += 2)
   {
      ry = Y[incY*j];
      iy = Y[incY*j+1];
      for (i=0; i < M2; i += 2)
      {
         rx = X[i];
         ix = X[i+1];
         ra = rx * ry;
         ia = ix * ry;
         #ifdef Conj_
            ra += ix * iy;
            ia -= rx * iy;
         #else
            ra -= ix * iy;
            ia += rx * iy;
         #endif
         A[i+j*lda] += ra;
         A[i+j*lda+1] += ia;
      }
   }
}
```

Which we test in time by:

```
speedy. make zr1tstcase r1rout=../CASES/ATL_zger1_1x1_1.c xu=1 yu=1
   TEST CONJ=0, M=997, N=177, lda=1006, incY=1, STARTED
```

```
   TEST CONJ=0, M=997, N=177, lda=1006, incY=1, PASSED
   TEST CONJ=0, M=997, N=177, lda=1006, incY=3, STARTED
   TEST CONJ=0, M=997, N=177, lda=1006, incY=3, PASSED
   TEST CONJ=0, M=997, N=177, lda=1006, incY=-3, STARTED
   TEST CONJ=0, M=997, N=177, lda=1006, incY=-3, PASSED
   TEST CONJ=1, M=997, N=177, lda=1006, incY=1, STARTED
   TEST CONJ=1, M=997, N=177, lda=1006, incY=1, PASSED
   TEST CONJ=1, M=997, N=177, lda=1006, incY=3, STARTED
   TEST CONJ=1, M=997, N=177, lda=1006, incY=3, PASSED
   TEST CONJ=1, M=997, N=177, lda=1006, incY=-3, STARTED
   TEST CONJ=1, M=997, N=177, lda=1006, incY=-3, PASSED

speedy. make zr1case r1rout=../CASES/ATL_zger1_1x1_1.c xu=1 yu=1
      zger_0 : 39.024390 MFLOPS
      zger_0 : 38.709677 MFLOPS
      zger_0 : 39.024390 MFLOPS
   zger_0 : 38.92 MFLOPS
```

6.2.2 GER kernel notes

As in GEMV, GER blocks X so that it is reused in L1. Since the dominant direction of the loop is expected to be down the columns of A, `incY` is not restricted to 1, while `incX` is. As in GEMV, all routines except SYR2 block only the the X dimension, while SYR2 blocks both, so that A can be reused in L1.

7 Speeding up the Level 1 BLAS

7.1 General comments for Level 1 optimization

All Level 1 optimizations are carried on in the `ATLAS/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 8.1), which makes prefetch instructions for various systems available for C programmers.

7.1.1 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`).

7.1.2 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> [\
```

7.1.3 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.

7.2 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.
```

7.3 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 . . .

7.4 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.

7.5 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 . . .

7.6 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`.

7.7 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`.

7.8 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.

7.9 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.

7.10 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.

7.11 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).

7.12 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>
   ```

7.13 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.

7.14 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.

7.15 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.

7.16 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.

7.17 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 ATLAS/tune/blas/level1/<arch>/res/<pre><blas>_SUMM. Then, go to ATLAS/src/blas/level1/<arch>, 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 ATLAS
rm tune/blas/level1/Linux_PIIISSE1/res/dAXPY_SUMM.
cd ATLAS/src/blas/level1/Linux_PIIISSE1
rm Make_daxpy
make Make_daxpy
make dlib
```

7.17.1 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 ATLAS/tune/blas/level1/<arch>.

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`

8 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
```

8.1 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_pf l1R(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_pf l1W(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).

9 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 atlas-comm 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.