Annex B

Legacy BLAS

B.1 Introduction

This chapter addresses additional language bindings for the original Level 1, 2, and 3 BLAS. The Level 1, 2, and 3 BLAS will hereafter be referred to as the Legacy BLAS.

B.2 C interface to the Legacy BLAS

This section gives a detailed discussion of the proposed C interface to the legacy BLAS. Every mention of "BLAS" in this chapter should be taken to mean the legacy BLAS. Each interface decision is discussed in its own section. Each section also contains a *Considered methods* subsection, where other solutions to that particular problem are discussed, along with the reasons why those options were not chosen. These *Considered methods* subsections are indented and *italicized* in order to distinguish them from the rest of the text.

It is largely agreed among the group (and unanimous among the vendors) that user demand for a C interface to the BLAS is insufficient to motivate vendors to support a completely separate standard. This proposal therefore confines itself to an interface which can be readily supported on top of the already existing Fortran 77 callable BLAS (i.e., the legacy BLAS).

The interface is expressed in terms of ANSI/ISO C. Very few platforms fail to provide ANSI/ISO C compilers at this time, and for those platforms, free ANSI/ISO C compilers are almost always available (eg., `gcc`).

B.2.1 Naming scheme

The naming scheme consists of taking the Fortran 77 routine name, making it lower case, and adding the prefix `cblas_`. Therefore, the routine `DGEMM` becomes `cblas_dgemm`.

Considered methods

> *Various other naming schemes have been proposed, such as adding* `C_` *or* `c_` *to the name. Most of these schemes accomplish the requirement of separating the Fortran 77 and C name spaces. It was argued, however, that the addition of the* `blas` *prefix unifies the naming scheme in a logical and useful way (making it easy to search for BLAS use in a code, for instance), while not placing too great a burden on the typist. The letter* `c` *is used to distinguish this language interface from possible future interfaces.*

B.2.2 Indices and I_AMAX

The Fortran 77 BLAS return indices in the range $1 \leq I \leq N$ (where N is the number of entries in the dimension in question, and I is the index), in accordance with Fortran 77 array indexing conventions. This allows functions returning indices to be directly used to index standard arrays. The C interface therefore returns indices in the range $0 \leq I < N$ for the same reason.

The only BLAS routine which returns an index is the function `I_AMAX`. This function is declared to be of type `CBLAS_INDEX`, which is guaranteed to be an integer type (i.e., no cast is required when assigning to any integer type). `CBLAS_INDEX` will usually correspond to `size_t` to ensure any array can be indexed, but implementors might choose the integer type which matches their Fortran 77 `INTEGER`, for instance. It is defined that zero is returned as the index for a zero length vector (eg., For $N = 0$, `I_AMAX` will always return zero).

B.2.3 Character arguments

All arguments which were characters in the Fortran 77 interface are handled by enumerated types in the C interface. This allows for tighter error checking, and provides less opportunity for user error. The character arguments present in the Fortran 77 interface are: `SIDE`, `UPLO`, `TRANSPOSE`, and `DIAG`. This interface adds another such argument to all routines involving two dimensional arrays, `ORDER`. The standard dictates the following enumerated types:

```
enum CBLAS_ORDER     {CblasRowMajor=101, CblasColMajor=102};
enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113};
enum CBLAS_UPLO      {CblasUpper=121, CblasLower=122};
enum CBLAS_DIAG      {CblasNonUnit=131, CblasUnit=132};
enum CBLAS_SIDE      {CblasLeft=141, CblasRight=142};
```

Considered methods

> *The other two most commonly suggested methods were accepting these arguments as either* `char *` *or* `char`. *It was noted that both of these options require twice as many comparisons as normally required to branch (so that the character may be either upper or lower case). Both methods also suffered from ambiguity (what does it mean to have* `DIAG='H'`, *for instance). If* `char` *was chosen, the words could not be written out as they can for the Fortran 77 interface (you couldn't write "NoTranspose"). If* `char *` *were used, some compilers might fail to optimize string constant use, causing unnecessary memory usage.*
>
> *The main advantage of enumerated data types, however, is that much of the error checking can be done at compile time, rather than at runtime (i.e., if the user fails to pass one of the valid options, the compiler can issue the error).*
>
> *There was much discussion as to whether the integer values should be specified, or whether only the enumerated names should be so specified. The group could find no substansive way in which specifying the integer values would restrict an implementor, and specifying the integer values was seen as an aid to inter-language calls.*

B.2.4 Handling of complex data types

All complex arguments are accepted as `void *`. A complex element consists of two consecutive memory locations of the underlying data type (i.e., `float` or `double`), where the first location contains the real component, and the second contains the imaginary part of the number.

In practice, programmers' methods of handling complex types in C vary. Some use various data structures (some examples are discussed below). Others accept complex numbers as arrays of the underlying type.

Complex numbers are accepted as void pointers so that widespread type casting will not be required to avoid warning or errors during compilation of complex code.

An ANSI/ISO committee is presently working on an extension to ANSI/ISO C which defines complex data types. The definition of a complex element is the same as given above, and so the handling of complex types by this interface will not need to be changed when ANSI/ISO C standard is extended.

Considered methods

> *Probably the most strongly advocated alternative was defining complex numbers via a structure such as*
> `struct NON_PORTABLE_COMPLEX {float r; float i;};` *The main problem with this solution is the lack of portability. By the ANSI/ISO C standard, elements in a structure are not guaranteed to be contiguous. With the above structure, padding between elements has been experimentally observed (on the CRAY T3D), so this problem is not purely theoretical.*
>
> *To get around padding problems within the structure, a structure such as*
> `struct NON_PORTABLE_COMPLEX {float v[2];};` *has been suggested. With this structure there will obviously be no padding between the real and imaginary parts. However, there still exists the possibility of padding between elements within an array. More importantly, this structure does not lend itself nearly as well as the first to code clarity.*
>
> *A final proposal is to define a structure which may be addressed the same as the one above (i.e.,* `ptr->r`, `ptr->i`*), but whose actual definition is platform dependent. Then, hopefully, various vendors will either use the above structure and ensure via their compilers its contiguousness, or they will create a different structure which can be accessed in the same way.*
>
> *This requires vendors to support something which is not in the ANSI C standard, and so there is no way to ensure this would take place. More to the point, use of such a structure turns out to not offer much in the way of real advantage, as discussed in the following section.*
>
> *All of these approaches require the programmer to either use the specified data type throughout the code which will call the BLAS, or to perform type casting on each BLAS call. When complex numbers are accepted as void pointers, no type casting or data type is dictated, with the only restriction being that a complex number have the definition given above.*

B.2.5 Return values of complex functions

BLAS routines which return complex values in Fortran 77 are instead recast as subroutines in the C interface, with the return value being an output parameter added to the end of the argument list. This allows the output parameter to be accepted as void pointers, as discussed above.

Further, the name is suffixed by `_sub`. There are two main reasons for this name change. First, the change from a function to a subroutine is a significant change, and thus the name should reflect this. More importantly, the "traditional" name space is specifically reserved for use when the forthcoming ANSI/ISO C extension is finalized. When this is done, this C interface will be extended to include functions using the "traditional" names which utilize the new ANSI/ISO complex type to return the values.

Considered methods

This is the area where use of a structure is most desired. Again, the most common suggestion is a structure such as `struct NON_PORTABLE_COMPLEX {float r; float i;};`.

If one is willing to use this structure throughout one's code, then this provides a natural and convenient mechanism. If, however, the programmer has utilized a different structure for complex, this ease of use breaks down. Then, something like the following code fragment is required:

```
NON_PORTABLE_COMPLEX ctmp;
float cdot[2];

ctmp = cblas_cdotc(n, x, 1, y, 1);
cdot[0] = ctmp.r;
cdot[1] = ctmp.i;
```

which is certainly much less convenient than: `cblas_cdotc_sub(n, x, 1, y, 1, cdot)`.

It should also be noted that the primary reason for having a function instead of a subroutine is already invalidated by C's lack of a standard complex type. Functions are most useful when the result may be used directly as part of an in-line computation. However, since ANSI/ISO C lacks support for complex arithmetic primitives or operator overloading, complex functions cannot be standardly used in this way. Since the function cannot be used as a part of a larger expression, nothing is lost by recasting it as a subroutine; indeed a slight performance win may be obtained.

B.2.6 Array arguments

Arrays are constrained to being contiguous in memory. They are accepted as pointers, not as arrays of pointers.

All BLAS routines which take one or more two dimensional arrays as arguments receive one extra parameter as their first argument. This argument is of the enumerated type
`enum CBLAS_ORDER {CblasRowMajor=101, CblasColMajor=102};`.
If this parameter is set to `CblasRowMajor`, it is assumed that elements within a row of the array(s) are contiguous in memory, while elements within array columns are separated by a constant stride given in the `stride` parameter (this parameter corresponds to the leading dimension [e.g. `LDA`] in the Fortran 77 interface).

If the order is given as `CblasColMajor`, elements within array columns are assumed to be contiguous, with elements within array rows separated by `stride` memory elements.

Note that there is only one `CBLAS_ORDER` parameter to a given routine: all array operands are required to use the same ordering.

Considered methods

This solution comes after much discussion. C users appear to split roughly into two camps. Those people who have a history of mixing C and Fortran 77 (in particular making use of the Fortran 77 BLAS from C), tend to use column-major arrays in order to allow ease of inter-language operations. Because of the flexibility of pointers, this is *not appreciably harder than using row-major arrays, even though C "natively" possesses row-major arrays.*

The second camp of C users are not interested in overt C/Fortran 77 interoperability, and wish to have arrays which are row-major, in accordance with standard C conventions. The idea that they must recast their row-oriented algorithms to column-major algorithms is unacceptable; many in this camp would probably not utilize any BLAS which enforced a column-major constraint.

Because both camps are fairly widely represented within the target audience, it is impossible to choose one solution to the exclusion of the other.

Column-major array storage can obviously be supported directly on top of the legacy Fortran 77 BLAS. Recent work, particularly code provided by D.P. Manley of DEC, has shown that row-major array storage may also be supported in this way with little cost. Appendix B.2.12 discusses this issue in detail. To preview it here, we can say the level 1 and 3 BLAS require no extra operations or storage to support row-major operations on top of the legacy BLAS. Level 2 real routines also require no extra operations or storage. Some complex level 2 routines involving the conjugate transpose will require extra storage and operations in order to form explicit conjugates. However, this will always involve vectors, not the matrix. In the worst case, we will need n *extra storage, and* $3n$ *sign changes.*

One proposal was to accept arrays as arrays of pointers, instead of as a single pointer. The problems with this approach are manifold. First, the existing Fortran 77 BLAS could not be used, since they demand contiguous (though strided) storage. Second, this approach requires users of standard C 2D arrays or 1D arrays to allocate and assign the appropriate pointer array.

Beyond this, many of the vectors used in level 1 and level 2 BLAS come from rows or columns of two dimensional arrays. Elements within columns of row-major arrays are not uniformly strided, which means that a `n`*-element column vector would need* `n` *pointers to represent it. This then leads to vectors being accepted as arrays of pointers as well.*

Now, assuming both our one and two dimensional arrays are accepted as arrays of pointers, we have a problem when we wish to perform sub-array access. If we wish to pass an $m \times n$ *subsection of a this array of pointers, starting at row* i *and column* j*, we must allocate* m *pointers, and assign them in a section of code such as:*

```
float **A, **subA;

subA = malloc(m*sizeof(float*));
for (k=0; k != m; k++) subA[k] = A[i+k] + j;
cblas_rout(... subA ...);
```

The same operation must be done if we wish to use a row or column as a vector. This is not only an inconvenience, but can add up to a non-negligible performance loss as well.

A fix for these problems is that one and two dimensional arrays be passed as arrays of pointers, and then indices are passed in to indicate the sub-portion to access. Thus you have a call that looks like: `cblas_rout(... A, i, j, ...);`*. This solution still requires some additional tweaks to allow using two dimensional array rows and columns as vectors. Users presently using C 2D arrays or 1D arrays would have to malloc the* *array of pointers as shown in the preceding example in order to use this kind of interface. At any rate, a library accepting pointers to pointers cannot be supported on top of the Fortran 77 BLAS, while one supporting simple pointers can.*

If the programmer is utilizing the pointer to pointer style of array indexing, it is still possible to use this library providing that the user ensures that the operand matrix is contiguous, and that the rows are constantly strided. If this is the case, the user may pass the operand matrix to the library in precicely the same way as with a 2D C array: `cblas_rout(... &A[i][j] ...);`.

Example 1: making a library call with a C 2D array:

```
double A[50][25];  /* standard C 2D array */

cblas_rout(CblasRowMajor, ... &A[i][j], 25, ...);
```

Example 2: Legal use of pointer to pointer style programming and the CBLAS

```
double **A, *p;

A = malloc(M);
p = malloc(M*N*sizeof(double));
for (i=0; i < M; i++) A[i] = &p[i*N];

cblas_rout(CblasRowMajor, ... &A[i][j], N, ...);
```

Example 3: Illegal use of pointer to pointer style programming and the CBLAS

```
double **A, *p;

A = malloc(M);
p = malloc(M*N*sizeof(double));
for (i=0; i < M; i++) A[i] = malloc(N*sizeof(double));

cblas_rout(CblasRowMajor, ... &A[i][j], N, ...);
```

Note that Example 3 is illegal because the rows of A have no guaranteed stride.

B.2.7 Aliasing of arguments

Unless specified otherwise, only input-only arguments (specified with the `const` qualifier), may be legally aliased on a call to the C interface to the BLAS.

Considered methods

The ANSI C standard allows for the aliasing of output arguments. However, allowing this often carries a substantial performance penalty. This, along with the fact that Fortran 77 (which we hope to call for optimized libraries) does not allow aliasing of output arguments, led us to make this restriction.

B.2.8 C interface include file

The C interface to the BLAS will have a standard include file, called `cblas.h`, which minimally contains the definition of the CBLAS types and ANSI/ISO C prototypes for all BLAS routines. It is not an error to include this file multiple times. Refer to section B.2.11 for an example of a minimal `cblas.h`.

ADVICE TO THE IMPLEMENTOR:
Note that the vendor is not constrained to using precisely this include file; only the enumerated type definitions are fully specified. The implementor is free to make any other changes which are not apparent to the user. For instance, all matrix dimensions might be accepted as `size_t` *instead of* `int`*, or the implementor might choose to make some routines inline.*

B.2.9 Error checking

The C interface to the legacy BLAS must supply error checking corresponding to that provided by the reference Fortran 77 BLAS implementation.

B.2.10 Rules for obtaining the C interface from the Fortran 77

- The Fortran 77 routine name is changed to lower case, and prefixed by `cblas_`.
- All routines which accept two dimensional arrays (i.e., level 2 and 3), acquire a new parameter of type `CBLAS_ORDER` as their first argument, which determines if the two dimensional arrays are row or column major.
- *Character arguments* are replaced by the appropriate enumerated type, as shown in Section B.2.3.
- *Input arguments* are declared with the `const` modifier.
- *Non-complex scalar input arguments* are passed by value. This allows the user to put in constants when desired (eg., passing 10 on the command line for `N`).
- *Complex scalar input arguments* are passed as void pointers, since they do not exist as a predefined data type in ANSI/ISO C.
- *Array arguments* are passed by address.
- *Output scalar arguments* are passed by address.
- *Complex functions* become subroutines which return the result via a void pointer, added as the last parameter. The name is suffixed with `_sub`.

B.2.11 cblas.h include file

The `cblas.h` include file can be found on the BLAS Technical Forum webpage:

`http://www.netlib.org/blas/blast-forum/cblas.h`

B.2.12 Using Fortran 77 BLAS to support row-major BLAS operations

This section is not part of the standard per se. Rather, it exists as an advice to the implementor on how row-major BLAS operations may be implemented using column-major BLAS. This allows vendors to leverage years of Fortran 77 BLAS developement in producing the C BLAS.

Before this issue is examined in detail, a few general observations on array storage are helpful. We must distinguish between the matrix and the array which is used to store the matrix. The matrix, and its rows and columns, have mathematical meaning. The array is simply the method of storing the matrix, and its rows and columns are significant only for memory addressing.

Thus we see we can store the columns of a matrix in the rows of an array, for instance. When this occurs in the BLAS, the matrix is said to be stored in transposed form.

A row-major array stores elements along a row in contiguous storage, and separates the column elements by some constant stride (often the actual length of a row). Column-major arrays have contiguous columns, and strided rows. The importance of this is to note that a row-major array storing a matrix in the natural way, is a transposed column-major array (i.e., it can be thought of as a column-major array where the rows of the matrix are stored in the columns of the array).

Similarly, an upper triangular row-major array corresponds to a transposed lower triangular column-major array (the same is true in reverse [i.e., lower-to-upper], obviously). To see this, simply think of what a upper triangular matrix stored in a row-major array looks like. The first n entries contain the first matrix row, followed by a non-negative gap, followed by the second matrix row.

If this same array is viewed as column-major, the first n entries are a column, instead of a row, so that the columns of the array store the rows of the matrix (i.e., it is transposed). This means that if we wish to use the Fortran 77 (column-major) BLAS with triangular matrices coming from C (possibly row-major), we will be reversing the setting of `UPLO`, while simultaneously reversing the setting of `TRANS` (this gets slightly more complicated when the conjugate transpose is involved, as we will see).

Finally, note that if a matrix is symmetric or Hermitian, its rows are the same as its columns, so we may merely switch `UPLO`, without bothering with `TRANS`.

In the BLAS, there are two separate cases of importance. one dimensional arrays (storage for vectors) have the same meaning in both C and Fortran 77, so if we are solving a linear algebra problem who's answer is a vector, we will need to solve the same problem for both languages. However, if the answer is a matrix, in terms of calling routines which use column-major storage from one using row-major storage, we will want to solve the *transpose* of the problem.

To get an idea of what this means, consider a contrived example. Say we have routines for simple matrix-matrix and matrix-vector multiply. The vector operation is $y \leftarrow A \times x$, and the matrix operation is $C \leftarrow A \times B$. Now say we are implementing these as calls from row-major array storage to column-major storage. Since the matrix-vector multiply's answer is a vector, the problem we are solving is remains the same, but we must remember that our C array A is a Fortran 77 A^T. On the other hand, the matrix-matrix multiply has a matrix for a result, so when the differing array storage is taken into account, the problem we want to solve is $C^T \leftarrow B^T \times A^T$.

This last example demonstrates another general result. Some level 3 BLAS contain a `SIDE` parameter, determining which side a matrix is applied on. In general, if we are solving the transpose of this operation, the side parameter will be reversed.

With these general principles, it is possible to show that all that row-major level 3 BLAS can be expressed in terms of column-major BLAS without any extra array storage or extra operations. In the level 2 BLAS, no extra storage or array accesses are required for the real routines. Complex routines involving the conjugate transpose, however, may require a n-element temporary, and up to $3n$ more operations (vendors may avoid all extra workspace and operations by overloading the `TRANS` option for the level 2 BLAS: letting it also allow conjugation without doing the transpose). The level 1 BLAS, which deal exclusively with vectors, are unaffected by this storage issue.

With these ideas in mind, we will now show how to support a row-major BLAS on top of a column major BLAS. This information will be presented in tabular form. For brevity, row-major storage will be referred to as coming from C (even though column-major arrays can also come from C), while column-major storage will be referred to as F77.

Each table will show a BLAS invocation coming from C, the operation that the BLAS should perform, the operation required once F77 storage is taken into account (if this changes), and the call to the appropriate F77 BLAS. Not every possible combination of parameters is shown, since many are simply reflections of another (i.e., when we are applying the `Upper, NoTranspose` becomes `Lower, Transpose` rule, we will show it for only the upper case. In order to make the notation more concise, let us define $\overline{x}$ to be $conj(x)$.

Level 2 BLAS

GEMV

C call	cblas_cgemv(CblasRowMajor, CblasNoTrans, m, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A x + \beta y$
F77 call	CGEMV('T', n, m, α, A, lda, x, incx, β, y, incy)

C call	cblas_cgemv(CblasRowMajor, CblasTrans, m, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A^T x + \beta y$
F77 call	CGEMV('N', n, m, α, A, lda, x, incx, β, y, incy)

C call	cblas_cgemv(CblasRowMajor, CblasConjTrans, m, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A^H x + \beta y \Rightarrow \overline{(\overline{y} \leftarrow \overline{\alpha} A^T \overline{x} + \overline{\beta}\overline{y})}$
F77 call	CGEMV('N', n, m, $\overline{\alpha}$, A, lda, $\overline{x}$, 1, $\overline{\beta}$, $\overline{y}$, incy)

Note that we switch the value of transpose to handle the row/column major ordering difference. In the last case, we will require n elements of workspace so that we may store the conjugated vector $\overline{x}$. Then, we set $y = \overline{y}$, and make the call. This gives us the conjugate of the answer, so we once again set $y = \overline{y}$. Therefore, we see that to support the conjugate transpose, we will need to allocate an n-element vector, and perform $2m + n$ extra operations.

SYMV

SYMV requires no extra workspace or operations.

C call	cblas_csymv(CblasRowMajor, CblasUpper, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A x + \beta y \Rightarrow y \leftarrow \alpha A^T x + \beta y$
F77 call	CSYMV('L', n, α, A, lda, x, incx, β, y, incy)

HEMV

HEMV routine requires $3n$ conjugations, and n extra storage.

C call	cblas_chemv(CblasRowMajor, CblasUpper, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A x + \beta y \Rightarrow y \leftarrow \alpha A^H x + \beta y \Rightarrow \overline{(\overline{y} \leftarrow \overline{\alpha} A^T \overline{x} + \overline{\beta}\overline{y})}$
F77 call	CHEMV('L', n, $\overline{\alpha}$, A, lda, $\overline{x}$, incx, $\overline{\beta}$, $\overline{y}$, incy)

TRMV/TRSV

C call	`cblas_ctrmv(CblasRowMajor, CblasUpper, CblasNoTrans, diag, n, A, lda, x, incx)`
op	$x \leftarrow Ax$
F77 call	`CTRMV('L', 'T', diag, n, A, lda, x, incx)`

C call	`cblas_ctrmv(CblasRowMajor, CblasUpper, CblasTrans, diag, n, A, lda, x, incx)`
op	$x \leftarrow A^T x$
F77 call	`CTRMV('L', 'N', diag, n, A, lda, x, incx)`

C call	`cblas_ctrmv(CblasRowMajor, CblasUpper, CblasConjTrans, diag, n, A, lda, x, incx)`
op	$x \leftarrow A^H x \Rightarrow \overline{(\overline{x} = A^T \overline{x})}$
F77 call	CTRMV('L', 'N', diag, n, A, lda, $\overline{x}$, incx)

Again, we see that we will need some extra operations when we are handling the conjugate transpose. We conjugate x before the call, giving us the conjugate of the answer we seek. We then conjugate this again to return the correct answer. This routine therefore needs $2n$ extra operations for the complex conjugate case.

The calls with the C array being `Lower` are merely the reflection of these calls, and thus are not shown. The analysis for TRMV is the same, since it involves the same principle of what a transpose of a triangular matrix is.

GER/GERU

This is our first routine that has a matrix as the solution. Recalling that this means we solve the transpose of the original problem, we get:

C call	cblas_cgeru(CblasRowMajor, m, n, α, x, incx, y, incy, A, lda)
C op	$A \leftarrow \alpha x y^T + A$
F77 op	$A^T \leftarrow \alpha y x^T + A^T$
F77 call	CGERU(n, m, α, y, incy, x, incx, A, lda)

No extra storage or operations are required.

GERC

C call	cblas_cgerc(CblasRowMajor, m, n, α, x, incx, y, incy, A, lda)
C op	$A \leftarrow \alpha x y^H + A$
F77 op	$A^T \leftarrow \alpha (x y^H)^T + A^T = \alpha \overline{y} x^T + A^T$
F77 call	CGERU(n, m, α, $\overline{y}$, incy, x, incx, A, lda)

Note that we need to allocate n-element workspace to hold the conjugated y, and we call `GERU`, not `GERC`.

HER

C call	cblas_cher(CblasRowMajor, CblasUpper, n, α, x, incx, A, lda)
C op	$A \leftarrow \alpha x x^H + A$
F77 op	$A^T \leftarrow \alpha \overline{x} x^T + A^T$
F77 call	CHER('L', n, α, $\overline{x}$, 1, A, lda)

Again, we have an n-element workspace and n extra operations.

HER2

C call	`cblas_cher2(CblasRowMajor, CblasUpper, n,` α`, x, incx, y, incy, A, lda)`
C op	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$
F77 op	$A^T \leftarrow \alpha \overline{y} x^T + \overline{\alpha x} y^T + A^T = \alpha \overline{y} (\overline{x})^H + \overline{x}(\alpha \overline{y})^H + A^T$
F77 call	`CHER2('L', n,` α`,` $\overline{y}$`, 1,` $\overline{x}$`, 1, A, lda)`

So we need $2n$ extra workspace and operations to form the conjugates of x and y.

SYR

C call	`cblas_ssyr(CblasRowMajor, CblasUpper, n,` α`, x, incx, A, lda)`
C op	$A \leftarrow \alpha x x^T + A$
F77 op	$A^T \leftarrow \alpha x x^T + A^T$
F77 call	`SSYR('L', n,` α`, x, incx, A, lda)`

No extra storage or operations required.

SYR2

C call	`cblas_ssyr2(CblasRowMajor, CblasUpper, n,` α`, x, incx, y, incy, A, lda)`
C op	$A \leftarrow \alpha x y^T + \alpha y x^T + A$
F77 op	$A^T \leftarrow \alpha y x^T + \alpha x y^T + A^T$
F77 call	`SSYR2('L', n,` α`, y, incy, x, incx, A, lda)`

No extra storage or operations required.

Level 3 BLAS

GEMM

C call	cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A^T + \beta C^T$
F77 call	CGEMM('N', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB^T + \beta C$
F77 op	$C^T \leftarrow \alpha BA^T + \beta C^T$
F77 call	CGEMM('T', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasConjTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{B} A^T + \beta C^T$
F77 call	CGEMM('C', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasTrans, CblasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A + \beta C^T$
F77 call	CGEMM('N', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasTrans, CblasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B^T + \beta C$
F77 op	$C^T \leftarrow \alpha BA + \beta C^T$
F77 call	CGEMM('T', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasTrans, CblasConjTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{B} A + \beta C^T$
F77 call	CGEMM('C', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasConjTrans, CblasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^H B + \beta C$
F77 op	$C^T \leftarrow \alpha B^T \overline{A} + \beta C^T$
F77 call	CGEMM('N', 'C', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasConjTrans, CblasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^H B^T + \beta C$
F77 op	$C^T \leftarrow \alpha B \overline{A} + \beta C^T$
F77 call	CGEMM('T', 'C', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasConjTrans, CblasConjTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^H B^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{BA} + \beta C^T$
F77 call	CGEMM('C', 'C', n, m, k, α, B, ldb, A, lda, β, C, ldc)

SYMM/HEMM

C call	cblas_chemm(CblasRowMajor, CblasLeft, CblasUpper, m, n, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A^T + \beta C^T$
F77 call	CHEMM('R', 'L', n, m, α, A, lda, B, ldb, β, C, ldc)

C call	cblas_chemm(CblasRowMajor, CblasRight, CblasUpper, m, n, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha BA + \beta C$
F77 op	$C^T \leftarrow \alpha A^T B^T + \beta C^T$
F77 call	CHEMM('L', 'L', n, m, α, A, lda, B, ldb, β, C, ldc)

SYRK

C call	cblas_csyrk(CblasRowMajor, CblasUpper, CblasNoTrans, n, k, α, A, lda, β, C, ldc)
C op	$C \leftarrow \alpha AA^T + \beta C$
F77 op	$C^T \leftarrow \alpha AA^T + \beta C^T$
F77 call	CSYRK('L', 'T', n, k, α, A, lda, B, ldb, β, C, ldc)

C call	cblas_csyrk(CblasRowMajor, CblasUpper, CblasTrans, n, k, α, A, lda, β, C, ldc)
C op	$C \leftarrow \alpha A^T A + \beta C$
F77 op	$C^T \leftarrow \alpha A^T A + \beta C^T$
F77 call	CSYRK('L', 'N', n, k, α, A, lda, B, ldb, β, C, ldc)

In reading the above descriptions, it is important to remember a few things. First, the symmetric matrix is C, and thus we change `UPLO` to accommodate the differing storage of C. `TRANSPOSE` is then varied to handle the storage effects on A.

HERK

C call	cblas_cherk(CblasRowMajor, CblasUpper, CblasNoTrans, n, k, α, A, lda, β, C, ldc)
C op	$C \leftarrow \alpha AA^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{A} A^T + \beta C^T$
F77 call	CHERK('L', 'C', n, k, α, A, lda, B, ldb, β, C, ldc)

C call	cblas_cherk(CblasRowMajor, CblasUpper, CblasConjTrans, n, k, α, A, lda, β, C, ldc)
C op	$C \leftarrow \alpha A^H A + \beta C$
F77 op	$C^T \leftarrow \alpha A^T \overline{A} + \beta C^T$
F77 call	CHERK('L', 'N', n, k, α, A, lda, B, ldb, β, C, ldc)

SYR2K

C call	cblas_csyr2k(CblasRowMajor, CblasUpper, CblasNoTrans, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
F77 op	$C^T \leftarrow \alpha BA^T + \alpha AB^T + \beta C^T = \alpha AB^T + \alpha BA^T + \beta C^T$
F77 call	CSYR2K('L', 'T', n, k, α, A, lda, B, ldb, β, C, ldc)

C call	cblas_csyr2k(CblasRowMajor, CblasUpper, CblasTrans, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A + \alpha A^T B + \beta C^T = \alpha A^T B + \alpha B^T A + \beta C^T$
F77 call	CSYR2K('L', 'N', n, k, α, A, lda, B, ldb, β, C, ldc)

Note that we once again wind up with an operation that looks the same from C and Fortran 77, saving that the C operations wishes to form C^T, instead of C. So once again we flip the setting of `UPLO` to handle the difference in the storage of C. We then flip the setting of `TRANS` to handle the storage effects for A and B.

HER2K

C call	`cblas_cher2k(CblasRowMajor, CblasUpper, CblasNoTrans, n, k, `α`, A, lda, B, ldb, `β`, C, ldc)`
C op	$C \leftarrow \alpha AB^H + \overline{\alpha} BA^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{B} A^T + \overline{\alpha}\overline{A}B^T + \beta C^T = \overline{\alpha}\overline{A}B^T + \alpha\overline{B}A^T + \beta C^T$
F77 call	`CHER2K('L', 'C', n, k, `$\overline{\alpha}$`, A, lda, B, ldb, `β`, C, ldc)`

C call	`cblas_cher2k(CblasRowMajor, CblasUpper, CblasConjTrans, n, k, `α`, A, lda, B, ldb, `β`, C, ldc)`
C op	$C \leftarrow \alpha A^H B + \overline{\alpha} B^H A + \beta C$
F77 op	$C^T \leftarrow \alpha B^T \overline{A} + \overline{\alpha} A^T \overline{B} + \beta C^T = \overline{\alpha} A^T \overline{B} + \alpha B^T \overline{A} + \beta C^T$
F77 call	`CHER2K('L', 'N', n, k, `$\overline{\alpha}$`, A, lda, B, ldb, `β`, C, ldc)`

TRMM/TRSM

Because of their identical use of the `SIDE`, `UPLO`, and `TRANSA` parameters, TRMM and TRSM share the same general analysis. Remember that A is a triangular matrix, and thus when we handle its storage by flipping `UPLO`, we implicitly change its `TRANS` setting as well. With this in mind, we have:

C call	`cblas_ctrmm(CblasRowMajor, CblasLeft, CblasUpper, CblasNoTrans, diag, m, n, `α`, A, lda, B, ldb)`
C op	$B \leftarrow \alpha AB$
F77 op	$B^T \leftarrow \alpha B^T A^T$
F77 call	`CTRMM('R', 'L', 'N', diag, n, m, `α`, A, lda, B, ldb)`

C call	`cblas_ctrmm(CblasRowMajor, CblasLeft, CblasUpper, CblasTrans, diag, m, n, `α`, A, lda, B, ldb)`
C op	$B \leftarrow \alpha A^T B$
F77 op	$B^T \leftarrow \alpha B^T A$
F77 call	`CTRMM('R', 'L', 'T', diag, n, m, `α`, A, lda, B, ldb)`

C call	`cblas_ctrmm(CblasRowMajor, CblasLeft, CblasUpper, CblasConjTrans, diag, m, n, `α`, A, lda, B, ldb)`
C op	$B \leftarrow \alpha A^H B$
F77 op	$B^T \leftarrow \alpha B^T \overline{A}$
F77 call	`CTRMM('R', 'L', 'C', diag, n, m, `α`, A, lda, B, ldb)`

Banded routines

The above techniques can be used for the banded routines only if a C (row-major) banded array has some sort of meaning when expanded as a Fortran banded array. It turns out that when this is done, you get the transpose of the C array, just as in the dense case.

In Fortran 77, the banded array is an array whose rows correspond to the diagonals of the matrix, and whose columns contain the selected portion of the matrix column. To rephrase this, the diagonals of the matrix are stored in strided storage, and the relevant pieces of the columns of the matrix are stored in contiguous memory. This makes sense: in a column-based algorithm, you will want your columns to be contiguous for efficiency reasons.

In order to ensure our columns are contiguous, we will structure the banded array as shown below. Notice that the first K_U rows of the array store the superdiagonals, appropriately spaced to line up correctly in the column direction with the main diagonal. The last K_L rows contain the subdiagonals.

```
       ------  Super diagonal KU
  -----------  Super diagonal 2
 ------------  Super diagonal 1
-------------  main diagonal (D)
------------   Sub diagonal 1
-----------    Sub diagonal 2
------         Sub diagonal KL
```

If we have a row-major storage, and thus a row-oriented algorithm, we will similarly want our rows to be contiguous in order to ensure efficiency. The storage scheme that is thus dictated is shown below. Notice that the first K_L columns store the subdiagonals, appropriately padded to line up with the main diagonal along rows.

```
 KL     D   KU
        |  |  |  |
     |  |  |  |  |
  |  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |
|  |  |  |
```

Now, let us contrast these two storage schemes. Both store the diagonals of the matrix along the non-contiguous dimension of the matrix. The column-major banded array stores the matrix columns along the contiguous dimension, whereas the row-major banded array stores the matrix rows along the contiguous storage.

This gives us our first hint as to what to do: rows stored where columns should be, indicated, in the dense routines, that we needed to set a transpose parameter. We will see that we can do this for the banded routines as well.

We can further note that in the column-major banded array, the first part of the non-contiguous dimension (i.e. the first rows) store superdiagonals, whereas the first part of the non-contiguous dimension of row-major arrays (i.e., the first columns) store the subdiagonals.

We now note that when you transpose a matrix, the superdiagonals of the matrix become the subdiagonals of the matrix transpose (and vice versa).

Along the contiguous dimension, we note that we skip K_U elements before coming to our first entry in a column-major banded array. The same happens in our row-major banded array, except that the skipping factor is K_L.

All this leads to the idea that when we have a row-major banded array, we can consider it as a transpose of the Fortran 77 column-major banded array, where we will swap not only m and n, but also K_U and K_L. An example should help demonstrate this principle. Let us say we have the matrix $A = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{bmatrix}$

If we express this entire array in banded form (a fairly dumb thing to do, but good for example purposes), we get $K_U = 3$, $K_L = 1$. In row-major banded storage this becomes: $C_b = \begin{bmatrix} X & 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 & X \end{bmatrix}$

So, we believe this should be the transpose if interpreted as a Fortran 77 banded array. The matrix transpose, and its Fortran 77 banded storage is shown below:

$$A^T = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \Rightarrow F_b = \begin{bmatrix} X & 2 \\ 1 & 4 \\ 3 & 6 \\ 5 & 8 \\ 7 & X \end{bmatrix}$$

Now we simply note that since C_b is row major, and F_b is column-major, they are actually the same array in memory.

With the idea that row-major banded matrices produce the transpose of the matrix when interpreted as column-major banded matrices, we can use the same analysis for the banded BLAS as we used for the dense BLAS, noting that we must also always swap K_U and K_L.

Packed routines

Packed routines are much simpler than banded. Here we have a triangular, symmetric or Hermitian matrix which is packed so that only the relevant triangle is stored. Thus if we have an upper triangular matrix stored in column-major packed storage, the first element holds the relevant portion of the first column of the matrix, the next two elements hold the relevant portion of the second column, etc.

With an upper triangular matrix stored in row-major packed storage, the first N elements hold the first row of the matrix, the next $N - 1$ elements hold the next row, etc.

Thus we see in the Hermitian and symmetric cases, to get a row-major packed array correctly interpreted by Fortran 77, we will simply switch the setting of `UPLO`. This will mean that the rows of the matrix will be read in as the columns, but this is not a problem, as we have seen before. In the symmetric case, since $A = A^T$ the column and rows are the same, so there is obviously no problem. In the Hermitian case, we must be sure that the imaginary component of the diagonal is not used, and it assumed to be zero. However, the diagonal element in a row when our matrix is upper will correspond to the diagonal element in a column when our matrix is called lower, so this is handled as well.

In the triangular cases, we will need to change both `UPLO` and `TRANS`, just as in the dense routines.

With these ideas in mind, the analysis for the dense routines may be used unchanged for packed.