D R A F T
Document for the BLAS Lite Specification

Anthony Skjellum and Purushotham Bangalore
Department of Computer Scince and
NSF Engineering Research Center
Mississippi State University
Mississippi State, MS 39762

{tony,puri}@cs.msstate.edu

February 14, 1997

Chapter 1

BLAS Lite

1.1 What is BLAS Lite ?

BLAS Lite is a language independent specification of standard routines used to build high performance linear algebra programs. BLAS Lite also includes language-specific optimizations and accommodations. The main objective of the BLAS Lite specification is to provide the basic functionality with "thin" interfaces (or avoid fat interfaces). In this document the term "lite" refers to the low-level primitive functions whereas "thin" refers to the simple interface without overloaded functionality.

1.2 Specifying BLAS Lite

The current proposal is to specify BLAS Lite with a language independent specification and language specific binding for Fortran-77, Fortran-90, C, and C++ languages.

1.2.1 Language independent specification (LIS)

We propose to use a language independent specification similar in style to MPI bindings such that the interface will provide the following features:

- thread safety – guarantee that the application program interface (API) is thread safe, so that a multi-threaded implementation will work correctly.
- thread compliant – guarantee that all the rules for behavior of multithreaded applications are satisfied.
- object-based – encapsulate the data and treat them as opaque objects and provide functions to access and manipulate data.
- clear naming – avoid using 6 letter names and use appropriate names to describe the functionality.
- profiling interface – assist tools developers to develop profiling tools over a standard interface

Also subprograms calls are recommended instead of function calls to avoid copies and creation of temporary variables. Error checking is performed only by separate compilation with the use of a debug flag.

1.2.2 Language specific bindings (LSB)

We propose to include language bindings for Fortran-77, Fortran-90, C, and C++ languages. In order to obtain better preformance we propose to use some of the features like inlining and macros if they are supported by the language. We also propose to include both early and late binding forms and rely on the compiler to perform the necessary optimizations. If such features are not supported by a langauge then we propose to use a preprocessor to obtain an equivalent form that can be handled by that language. In this document we have not explained the interlanguage features, which need to be addressed if the language bindings are not uniform between different languages.

1.3 Is BLAS Lite to be big or small ?

A simple strategy will be to have a huge number of thin interfaces with late bindings to incorporate the complete functionality. Also we will need error and non-error versions for each function. This will make the BLAS Lite very big and will require too much vendor effort. Also the semantic bottleneck still remains and the design will not be expandable to new data structures, which in turn would suggest that we would be better off with the existing BLAS.

The other option is to keep BLAS Lite small with a minimum number of operations to be implemented and optimized by vendor for high performance. These building blocks can then be used to build sparse or dense operations. To obtain maximum performance and encapsulate the details of memory hierarchy we need to provide inquiring/cache friendly operations that will provide information about different cache sizes and other performance related parameters. This option will make the vendor effort bounded since there are a few functions that need to be optimized.

1.4 Tiny BLAS Lite

The building blocks for BLAS Lite will be the tiny BLAS that address finite size problems wherein the problem size is chosen such that the entire block of data resides in the cache or the highest level of memory hierarchy. These building blocks are developed only for canonical storage and data is accesses directly without the use of iterators and no runtime checking is performed inside these routines. To obtain performance we propose to use composition, compile time optimization and inlining for C++ and Fortran-90 languages. For C and Fortran-77 we propose to use macros and preprocessing. Also we propose to support data structures that are natural to a particular language, that is, column storage format for Fortran-77 and Fortran-90, and row storage format for C and C++. A simple example for using the tiny BLAS Lite within a hypothetical BLAS is given in the examples section at the end of this document.

1.4.1 Environmental inquiry functions

These functions provide some of the following environment related information:

- size restrictions (maximum block sizes)
- runtime information
- compile time or offline information

In this subsection we describe some of the important environmental inquiry functions. The general specification for a tiny BLAS Lite function call will be as follows:

BLAS_LITE_FUNCTION_NAME(arg1,arg2,arg3)

IN	arg1	explanation (type)
OUT	arg2	explanation (type)
INOUT	arg3	explanation (type)

In the above function specification the first column indicates whether the arguments are input (IN) or output (OUT) or input-output (INOUT). Input indicates that the value of the argument will not be altered by the function call and output indicates that the argument will have new data after the function call. Input-output arguments indicate that there was valid data when the function was called and will have new data when the function returns. When the argument is a handle to an opaque object and the contents of the object are modified by a function call still we consider it as input-output (INOUT). The second column above lists the arguments and the third column has the arguments explanation and the type.

1.4.2 Concept #1

Consider a general matrix-matrix update operation:

$$\theta C + \phi D = \alpha A^{[T]} op B^{[T]} + \beta C$$

where

- A, B, C, and D are matrices of finite size (say 16×16)
- *op* is $\times$ or $:$ (tensor contraction)
- α and β are any constants and θ and ϕ have mutually exclusive values of either 0 or 1
- all operands use same canonical storage
- implementation defines primitive precision set
- no runtime checking mandated

Then the possible number of functions are as follows:

- $\alpha : +1|-1|0|\gamma(4)$
- $A : A|A^T(2)$
- $B : B|B^T(2)$
- $\beta : +1|-1|0|\gamma(4)$
- $op : \times| : (2)$

where γ is any constant. This will make the total number of functions $= 4 \times 2 \times 2 \times 4 \times 2 = 128$.

In order to name these functions we need to encode some special behavior in function naming. For instance, in C++ function overloading can be used and in C/F77 macros or preprocessing will be required to select the appropriate function. Thus we emphasize the use of compiler and preprocessers to perform checking at compile time than run time. For the matrix-matrix operations described above one way of naming the functions is shown below:

```
BLAS_LITE_MPY(arg1, .. ,argn)
BLAS_LITE_MPY_AT(arg1, .. , argn)
BLAS_LITE_MPY_ATBT(arg1, .. , argn)
```

1.5 Concept #2

This is a specialization of concept #1 where A or B can be a vectors. Some compilers may be able to optimize concept #1 when A or B are vectors. If it is possible to obtain better performance for vector related operations by having separate functions then we will have twice the number of functions resulting in a total of 256 functions.

1.6 Concept #3

This section will include all functions of the form $f(input, args, output)$, where f is some operation that is performed on *input* based on the parameters *args* and the result is returned as *output*. The user should be able to compose any inlinable functions. We have determined two classes of functions, namely, shape dependent and shape independent functions. Examples of shape dependent functions are copy, transpose (inplace also), and downselect and some of the shape independent functions are 2-norm (F-norm for matrices), 1-norm, etc. Shape independent functions can be viewed as functions that are performed on vectors or contiguous data whereas shape dependent functions can be viewed as functions on matrices or non-contiguous data. This will make the total functions equal to 300 (approximately).

1.7 Canonical storage enough ?

For the tiny BLAS Lite we need to determine whether the canonical storage formats are sufficient to optimize or other storage formats also needs to be implemented. We propose to support storage formats that are natural to a particular language, for example, row major for C and C++ and column major for Fortran-77 and Fortran-90. But this will make interoperability between different languages difficult. Other storage formats that can be considered are row, column, tiled-row, and tiled-column for all functions irrespective of the language. The tile sizes are implementation dependent and the environmental inquiry functions should be used to determine the implementation dependent information.

If we decide to support row, column, tiled-row, tiled-column storage formats for dense and compressed row and column storage formats for sparse data (with zero reference) we will have the function count equal to $6 \times 300 = 1800$. Other storage formats must use these 6 major formats as building blocks.

1.8 Copying between formats

We will also need functions to support coping between different sparse formats to dense and vise-versa. For special cases we can use a fixed intermediate format (say, dense). We propose to support only copying and operations on homogeneous data formats and allow user inlinable copies. These functions only have an additive effect, with the total number of functions equal to $1800 + 6 * 5 = 1830$.

1.9 Tiny BLAS Lite: Level 0

We further classify tiny BLAS Lite into level-0 and level-1/2. Tiny BLAS Lite will include minimum number of functions (20/format) and some of the typical functions would be:

- copying
- transpose inplace
- operation updates
- shape dependent/independent functions

We intend to rely on compile time optimization with inlining as well as runtime optimization to obtain better memory reuse and performance. All the tiny BLAS Lite functions are require to be optimized for all formats.

Some of the Level 0 functions (per format) based on most used functions are:

- $D = C^T, C = C^T, C = AB, C = A^TB, C = AB^T$
- $C = AB + C, C = AB - C$
- $C = A^TB + C, C = A^TB - C$
- $C = AB^T + C, C = AB^T - C$
- 1-norm, 2-norm, ∞-norm, etc.,
- z = v : w
- ....

1.10 Tiny BLAS Lite: Level 1/2

Level 1/2 of tiny BLAS Lite is composed as a thin layer on top of Level 0. Composition is obtained through inlining/macros. This will enable not only spatial reuse but also temporal reuse. For example,

```
loop {
   line1
   line2
   ....
   linen } ;
```

In the above example each of the lines correspond to a tiny Level 0 BLAS call composed within an outer loop. Level 1/2 tiny BLAS will be a portable code developed with one version for each language and shared by the community. Vendors are free to migrate more functions to native layer at a later time.

1.11 Tiny BLAS Lite - Advantages

Some of the main advantages of layering BLAS Lite on tiny BLAS are as follows:

- vendors can easily implement the level 0 functions and include more functions from level 1/2 later on
- since level 0 functions will be used as the high performance kernels by the other BLAS layers, they are guaranteed to produce high performance.
- optimal number of level 0 functions determined based on user expectations
- later revisions can mandate more functions
- quality of implementation allows flexibility for vendors
- a native implementation done in short time

1.12 Down sizing

The total number functions that vendors need to optimize will be the total number of functions in level 0, copying functions, and inquiry functions. That will make the total number of functions = 20*6 (level 0) + 30 (copying) + 40 (inquiry) = 190.

A smaller number of functions make the task of implementation and optimzation feasible. The crucial part of the process will be in deciding which level 0 functions to support and what data formats to support.

1.13 Summary and Conclusions

This proposal provides an orthogonal view of functionality and provides a specification for the light-weight kernel for BLAS, namely, BLAS Lite. The main idea is to use a small number of concepts that can be used as building blocks, these concepts are called the tiny BLAS. The dense and the sparse cases are unified for the tiny BLAS. Also we propose that the language bindings be specified in a way that is natural to that language so that it is easy to explain and document. The emphasis is to use inlining, macros, preprocessing, compile time optimizations and composition to obtain higher performance and make the task of development of complex linear algebra operations easier. The purpose of this document is to provide a background to identify some of the key issues that are important in developing a specification for BLAS Lite. Some of the important issues that should be identified by the BLAST Forum are as follows:

- data formats to be supported at Level 0
- critical level 0 functionality
- level 1/2 functionality.

Examples

Example 1.1 BLAS Lite from Fortran-77

```
        include 'blast_lite.inc'
        double precision TA(BLAS_LITE_MAXM,BLAS_LITE_MAXN)
        double precision TB(BLAS_LITE_MAXM,BLAS_LITE_MAXN)
        double precision TC(BLAS_LITE_MAXM,BLAS_LITE_MAXN)
        integer TLDA,TMA,TNA, TLDB,TMB,TNB, TLDC,TMC,TNC
        TLDA = BLAST_LITE_MAXM
        TLDB = BLAST_LITE_MAXM
        TLDC = BLAST_LITE_MAXM

C General Multiply:
        call DBLAS_LITE_MPY(TA,TLDA,TMA,TNA,
     $       TB,TLDB,TMB,TNB,TC,TLDC,TMC,TNC, ALPHA, BETA)

C Multiplication where constants are unity:
        call DBLAS_LITE_MPY_ALPHA1_BETA1(TA,TLDA,TMA,TNA,
     $       TB,TLDB,TMB,TNB,TC,TLDC,TMC,TNC)

C Multiplication where constants are unity:
        call DBLAS_LITE_MPY_ALPHA1_BETA1(TA,TLDA,TMA,TNA,
     $       TB,TLDB,TMB,TNB,TC,TLDC,TMC,TNC)

C Multiplication with A transposed:
        call DBLAS_LITE_MPY_AT(TA,TLDA,TMA,TNA,
     $       TB,TLDB,TMB,TNB,TC,TLDC,TMC,TNC,ALPHA,BETA)

C Multiplication with B a vector, ΔAlpha = Beta = 1:
C (C is a vector by implication also)
        call DBLAS_LITE_MPY_BV_ALPHA1_BETA1(TA,TLDA,TMA,TNA,
     $       TB,TBSTRIDE,TC,TCSTRIDE)

Conditions:
C        Vectors have starting address and constant stride
C
C        Matrices have LDA, shape (M,N)
C
```

Example 1.2 Layering BLAS Lite

```
C Higher level from lower level (hypothetical fn):
        include 'hypo_blas.inc'
        integer DESA(BLAS_DLEN),DESB(BLAS_DLEN),DESC(BLAS_DLEN)
        double precision A(LDA,NA), B(LDB), C(LDC)
        HDBLAS_DESCR_NEW(A,LDA,MA,NA, DESA)
        HDBLAS_DESCR_NEW(B,LDB,MB,1,  DESB)
        HDBLAS_DESCR_NEW(C,LDC,MC,1,  DESC)
C ...
        call HDBLAS_MPY_BV_ALPHA1_BETA1(DESA, DESB, DESC)
C
C You can actually access the submatrices
C       1) you define the structure above
C       2) hypothetical blas tells you about how subblocks
C               are stored, and provides accessor that are
C               not shown here, but easily worked out.
C       3) descriptors hide lots of the one-time integer
C               computations, like strides for vectors, etc.


       subroutine HDBLAS_MPY_BV_ALPHA1_BETA1(DESA, DESB, DESC)
       include 'hypo_blas.inc'
       integer DESA(BLAS_DLEN),DESB(BLAS_DLEN),DESC(BLAS_DLEN)
       BBS = HBLAS_BLOCK_SIZE
       do I = 1,DESA(HBLAS_BLOCK_ROWS)
            ICURS = (I-1)*BBS + 1
            do J = 1,DESA(HBLAS_BLOCK_COLS)
                 JCURS = (J-1)*LDA*BBS + 1
                call DBLAS_LITE_MPY_BV_ALPHA1_BETA1(
     $               A(ICURS,JCURS), BBS, BBS, BBS,
     $               B(JCURS), BBS, DESB(HBLAS_STRIDE),
     $               C(ICURS), BBS, DESC(HBLAS_STRIDE))
            enddo
       enddo
       return
       end
```