# A Set of Batched Basic Linear Algebra Subprograms and LAPACK Routines



AHMAD ABDELFATTAH, University of Tennessee, USA
TIMOTHY COSTA, NVIDIA, USA
JACK DONGARRA, University of Tennessee, Oak Ridge National Laboratory, and University of Manchester, USA
MARK GATES, University of Tennessee, USA
AZZAM HAIDAR, NVIDIA, USA
SVEN HAMMARLING and NICHOLAS J. HIGHAM, University of Manchester, UK
JAKUB KURZAK, AMD, USA
PIOTR LUSZCZEK and STANIMIRE TOMOV, University of Tennessee, USA
MAWUSSI ZOUNON, NAG Ltd., UK

This article describes a standard API for a set of Batched Basic Linear Algebra Subprograms (Batched BLAS or BBLAS). The focus is on many independent BLAS operations on small matrices that are grouped together and processed by a single routine, called a Batched BLAS routine. The matrices are grouped together in uniformly sized groups, with just one group if all the matrices are of equal size. The aim is to provide more efficient, but portable, implementations of algorithms on high-performance many-core platforms. These include multicore and many-core CPU processors, GPUs and coprocessors, and other hardware accelerators with floating-point compute facility. As well as the standard types of single and double precision, we also include half and quadruple precision in the standard. In particular, half precision is used in many very large scale applications, such as those associated with machine learning.

CCS Concepts: • **Mathematics of computing → Computations on matrices**;

Additional Key Words and Phrases: BLAS, batched BLAS

This material is based upon work supported in part by the National Science Foundation under Grants No. OAC 1740250 and CSR 1514286 and OAC 2004850, NVIDIA, the Department of Energy, and in part by the Russian Scientific Foundation, Agreement N14-11-00190. This project was also funded in part from the European Union's Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633.

Authors' addresses: A. Abdelfattah, M. Gates, P. Luszczek, and S. Tomov, University of Tennessee, 1122 Volunteer Blvd., Suite 203, Knoxville, TN 37996-3450, USA; emails: ahmad@icl.utk.edu, mgates3@icl.utk.edu, luszczek@icl.utk.edu, tomov@icl.utk.edu; T. Costa, NVIDIA, Santa Clara; J. Dongarra, University of Tennessee, Oak Ridge National Laboratory, and University of Manchester, Knoxville; email: dongarra@icl.utk.edu; A. Haidar, NVIDIA, Knoxville; email: azzamhaidar@nvidia.com; S. Hammarling and N. J. Higham, University of Manchester, Manchester, UK; emails: sven.hammarling@btinternet.com, nick.higham@manchester.ac.uk; J. Kurzak, AMD, Knoxville; M. Zounon, NAG Ltd., Manchester, UK; email: mawussi.zounon@nag.co.uk.


0098-3500/2021/06-ART21 $15.00
https://doi.org/10.1145/3431921

**ACM Reference format:**
Ahmad Abdelfattah, Timothy Costa, Jack Dongarra, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Mawussi Zounon. 2021. A Set of Batched Basic Linear Algebra Subprograms and LAPACK Routines. *ACM Trans. Math. Softw.* 47, 3, Article 21 (June 2021), 23 pages.
https://doi.org/10.1145/3431921

## 1 INTRODUCTION

### 1.1 The Batched BLAS

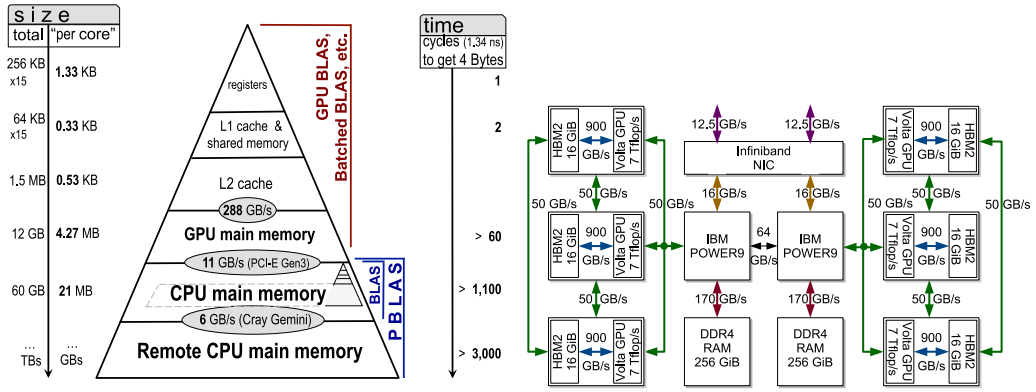
The specifications for the Level 1, 2, and 3 BLAS have been very successful in providing a standard for vector [38], matrix-vector [20, 21] and matrix-matrix [18, 19] operations, respectively. Vendors and other developers have provided highly efficient versions of the BLAS, and by using the standard interface have allowed software calling the BLAS to be portable. With the need to solve larger and larger problems on today's high-performance computers, the methods used in a number of applications, such as tensor contractions, finite element methods, and direct linear equation solvers, require a large number of small vector or matrix operations to be performed in parallel. So a typical example might be to perform

$$C_i \leftarrow \alpha_i A_i B_i + \beta_i C_i, \ i = 1, 2, \ldots \ell,$$

where $\ell$ is large, but $A_i, B_i$, and $C_i$ are small matrices. A routine to perform such a sequence of operations is called a Batched Basic Linear Algebra Subprogram, or Batched BLAS, or BBLAS.

### 1.2 History and Motivation

The origins of the **Basic Linear Algebra Subprograms (BLAS)** standard can be traced back to 1973, when Hanson, Krogh, and Lawson wrote an article in the *SIGNUM Newsletter* (Vol. 8, no. 4, p. 16) describing the advantages of adopting a set of basic routines for problems in linear algebra. This led to the development of the original BLAS [38], which indeed turned out to be advantageous and very successful. They were adopted as a standard and used in a wide range of numerical software, including LINPACK [13], and are now known as the Level 1 BLAS. An extended set of BLAS, the Level 2 BLAS, were proposed for matrix-vector operations [20, 21]. Unfortunately, while successful for the vector-processing machines at the time, the Level 2 BLAS were not a good fit for the cache-based and shared memory parallel machines that emerged in the 1980s. With such machines, it was preferable to express computations as matrix-matrix operations. Matrices were split into small blocks so basic operations were performed on blocks that could fit into cache memory. This approach avoids excessive movement of data to and from memory and gives a surface-to-volume effect for the ratio of data movement to floating-point operations. Hence, the Level 3 BLAS were proposed [18, 19] for matrix-matrix operations, and LAPACK [5] was developed to use the new Level 3 BLAS where possible. For the emerging multicore architectures of the 2000s, the PLASMA library [4, 15] introduced tiled algorithms and tiled data layouts. To handle parallelism, algorithms were split into tasks and data dependencies among the tasks were generated and used by runtime systems to properly schedule the tasks' execution over the available cores without violating any of the data dependencies. The scheduling overhead becomes a challenge in this approach, since a single Level 3 BLAS routine on large matrices would be split into many Level 3 BLAS computations on small matrices, all of which must be analyzed, scheduled, and launched, without using information that these are actually independent data-parallel operations that share similar data dependencies. To alleviate this scheduling overhead, the **SLATE library (Software**




(a) Memory hierarchy of a heterogeneous supercomputing system called Titan (installed in 2012 at Oak Ridge National Laboratory and decommissioned in 2019) from the point of view of a CUDA core of an NVIDIA K40c GPU with 2,880 CUDA cores.

(b) Diagram of a modern supercomputer called Summit with multiple CPUs and GPUs and the connection links' speeds between the components.

Fig. 1. Changes in memory hierarchy and node complexity across supercomputing hardware generations.

**for Linear Algebra Targeting Exascale)** [22] aggregates independent tile operations together and schedules them as a single task using a batched BLAS call on GPU accelerators.

In the 2010s, the apparently relentless trend in **high performance computing (HPC)** toward large-scale, heterogeneous systems with GPU accelerators and coprocessors made the near total absence of linear algebra software optimized for small matrix operations especially noticeable. The typical method of utilizing such hybrid systems is to increase the scale and resolution of the model used by an application, which in turn increases both matrix size and computational intensity; this tends to be a good match for the steady growth in performance and memory capacity of this type of hardware (see Figure 1 for an example of the memory hierarchy of this type of hardware).

The interface for the various implementations of the BBLAS differ, so software calling the BBLAS is not portable. Given the fundamental importance of numerical libraries to science and engineering applications of all types [35], the need for libraries that can perform batch operations on small matrices has clearly become acute. Therefore, to fill this critical gap, we propose standard interfaces for Batched BLAS operations.

The interfaces are intentionally designed to be close to the BLAS standard and to be *hardware independent*. They are given in C, rather than Fortran, but can nevertheless still be readily called from other languages and packages. The goal is to provide the developers of applications, compilers, and runtime systems with the option of expressing many small BLAS operations as a single call to a routine from the new batch operation standard, and thus to allow the entire **linear algebra (LA)** community to collectively attack a wide range of small matrix problems. We plan to also provide a Fortran interface in addition to the ISO C Binding introduced in Fortran 2003 that allows one to call C functions directly.

During the past standardization meetings [29, 30], a consensus emerged to amend the previous draft of the Batched BLAS standard [14] to include in the proposed interface the situation where the sizes of matrices in the batch vary by group. This is what we refer to as *grouped batch interface* and describe it in Section 2.2. When adding batched mode of operation, new functionality and data types were also considered during the standardization process. These included extending matrix layout definition to simultaneous non-unitary strides for both rows and columns, additional arguments to some routines such as $D \leftarrow C + A \times B$, or adding support BFloat16 data type that dramatically limits the size of significand compared to FP32 but keep its exponent range. However,

to keep the process focused and on track, it was decided to keep these out of the current standard definition and consider their inclusion in the future releases, as these new features gain significant foothold in practical uses of the standard.

### 1.3 Community Involvement

A large number of people have contributed ideas to the Batched BLAS project. Many of the contributions in the form of papers and talks can be found at http://icl.utk.edu/bblas/. Two workshops were held in May 2016 and February 2017 [29, 30], Birds of a Feather sessions were held at SC17 in Denver, Colorado, at ISC 2018 in Frankfurt am Main, Germany, and SC18 in Dallas, Texas; and Batched BLAS talks have been given in a number of conferences, including WSSSPE4 in Manchester, SIAM CSE 17 in Atlanta, Georgia, SIAM PP18 in Tokyo, and SIAM CSE19 in Spokane, Washington. A previous proposal was presented in Reference [14]; see also References [16] and [17]. An early version of this current proposal was produced under NLAFET project as deliverable D7.6, see http://www.nlafet.eu/public-deliverables/.

## 2 NAMING CONVENTIONS

### 2.1 Data Type and Functionality Conventions

The name of a Batched BLAS routine follows, and extends as needed, the conventions of the corresponding BLAS routine. In particular, the name has a `BLAS_` prefix, then 4 or 5 characters specifying the BLAS routine as described below, followed by the suffix `_batched`, and finally the data type suffix.

- After the `BLAS_` prefix, the next two characters in the name refer to the kind of matrix involved, as follows:
  - —ge represents: All matrices are general rectangular
  - —he represents: One of the matrices is Hermitian
  - —sy represents: One of the matrices is symmetric
  - —tr represents: One of the matrices is triangular
- The next two or three characters in the name denote the operation. For example, for the Level 3 Batched BLAS, the operations are given as follows:
  - —mm represents: Matrix-matrix product
  - —rk represents: Rank-k update of a symmetric or Hermitian matrix
  - —r2k represents: Rank-2k update of a symmetric or Hermitian matrix
  - —sm represents: Solve a system of linear equations for a matrix of right-hand sides
- At the end, the data type suffix indicates the data type for the matrix arguments. This part has variable width and specifies the domain—either r for real or c for complex—and the bit width of binary representation. For example, r64 corresponds to real 64-bit (double) and c32 is for complex arguments with both real and imaginary parts represented by 32-bit floating point values (complex-single). This notation is consistent with the BLAS G2 proposal [10]. Table 1 shows the suffixes and their corresponding data types in C, Fortran, and C++. The BLAS G2 proposal has a more extensive set of data types and provisions for mixed-precision and reproducible routines.

### 2.2 Groups of Same-size Batched BLAS Routines

The grouped batch interface introduces the concept of groups of same-sized matrices. The following formula calculates the argument formerly called `batch_count` (the total number of matrices in a single call) from the number and size of individual groups of matrices:

Table 1. Floating Point Data Types in Three Programming Languages and Their Corresponding Suffixes in Batched BLAS Interface

| Language → suffix ↓ | C | Fortran | C++ |
|---|---|---|---|
| r16 | _Float16 | real(kind=2) | short float |
| r32 | float | real(kind=4) | float |
| r64 | double | real(kind=8) | double |
| r80 | _Float80 | real(kind=10) | — |
| r128 | _Float128 | real(kind=16) | — |
| c16 | _Complex _Float16 | complex(kind=2) | complex<short float> |
| c32 | _Complex float | complex(kind=4) | complex<float> |
| c64 | _Complex double | complex(kind=8) | complex<double> |
| c80 | _Complex _Float80 | complex(kind=10) | — |
| c128 | _Complex _Float128 | complex(kind=16) | — |

**Note 1** short float is a proposed extension to the C18 and C++17 language standards.
**Note 2** long double does not map to a specific bit-width in a portable way and may be implemented as 80-bit or 128-bit floating point value according to the C standard document.

$$\texttt{batch\_count} = \sum_{\texttt{g=0}}^{\texttt{group\_count-1}} \texttt{group\_sizes[g]}. \tag{1}$$

*2.2.1 Batch Style Specification.* The `batch_opts` argument from the previous proposal [14] was an enumerated value that specified the style for the batch computation. Permitted values were either `BLAS_BATCH_FIXED` or `BLAS_BATCH_VARIABLE`, which stood for computation of matrices with the same or varying sizes (including operation options, sizes, matrix leading dimensions, and scalars), respectively. This was superseded by the group interface that is more general and applies to more usage scenarios.

**Note** that through the group interface one can specify constant size or variable size Batched BLAS operations. If a constant size batch is requested, then the arguments point to the corresponding constant value. The goal of this interface is to remove the need for users to prepare and pass arrays whenever they have the same elements. Through an internal dispatch and based on the group sizes, an expert routine specific to the value/style can be called while keeping the top interface the same.

### 2.3 Argument Conventions

We follow a convention for the list of arguments that is similar to that for BLAS, with the necessary adaptations concerning the batch operations. The order of arguments is as follows:

(1) Argument specifying row- or column-major layout
(2) Arrays of arguments specifying options
(3) Arrays of arguments defining the sizes of the matrices
(4) Arrays of input scalars (associated with input and/or output matrices)
(5) Arrays of descriptions of the input and/or output matrices
(6) Integer specifying the number of groups in the batch
(7) Integer array specifying size of each group in the batch
(8) Array of info parameters

Note that not every category is present in each of the routines.

*2.3.1 Argument Specifying Matrix Layout.* Matrix layout is only used when matrix arguments are present. A single value defines the layout for all matrices in all groups.

- layout has two possible values that are used by the routines as follows:
  —`BlasColMajor`: designates column-major layout of matrix elements;
  —`BlasRowMajor`: designates row-major layout of matrix elements.

*2.3.2 Arguments Specifying Options.* The arguments that specify options are of enumeration type with names such as side, transA, transB, trans, uplo, and diag. These arguments, along with the values that they can take, are described below:

- side has two possible values that are used by the routines as follows:
  —`BlasLeft`: Specifies to multiply a general matrix by a symmetric, Hermitian, or triangular matrix on the left;
  —`BlasRight`: Specifies to multiply general matrix by a symmetric, Hermitian, or triangular matrix on the right.
- transA, transB, and trans can have three possible values each, which is used to specify the following:
  —`BlasNoTrans`: Operate with the matrix as it is;
  —`BlasTrans`: Operate with the transpose of the matrix;
  —`BlasConjTrans`: Operate with the conjugate transpose of the matrix.
  Note that in the real case, i.e., when routines take only real-valued arguments, the enumeration values `BlasTrans` and `BlasConjTrans` have the same effect, i.e., a transposition without conjugation is assumed.
- uplo is used by the Hermitian, symmetric, and triangular matrix routines to specify whether the upper or lower triangle is being referenced, as follows:
  —`BlasLower`: Lower triangle;
  —`BlasUpper`: Upper triangle.
- diag is used by the triangular matrix routines to specify whether the matrix is unit triangular, as follows:
  —`BlasUnit`: Unit triangular;
  —`BlasNonUnit`: Nonunit triangular.
  When diag is supplied as `BlasUnit`, the diagonal elements are not referenced.

*2.3.3 Arguments Defining the Sizes.* The sizes of matrices $A_i$, $B_i$, and $C_i$ for the $i$th BLAS operation that are in the $g$th group are determined by the corresponding values of the arrays $m$, $n$, and $k$ at position $g$ (see the routine interfaces in Section 4). It is permissible to call the routines with $m = 0$ or $n = 0$, in which case the routines do not reference their corresponding matrix arguments and do not perform any computation on the corresponding matrices $A_i$, $B_i$, and $C_i$. If $m > 0$ and $n > 0$, but $k = 0$, then the Level 3 BLAS operation reduces to $C = \beta C$ (this applies to the `gemm`, `syrk`, `herk`, `syr2k`, and `her2k` routines). The input-output matrix ($B$ for the `tr` routines, $C$ otherwise) is always $m \times n$ when it is rectangular, and $n \times n$ when it is a square matrix. If there is only a single group of matrices of the same sizes, then the $m$, $n$, and $k$ values for all matrices are specified by the `m[0]`, `n[0]`, and `k[0]` values, respectively.

*2.3.4 Arguments Describing the Input-output Matrices.* The description of the matrix consists of the array name (`A`, `B`, or `C`) followed by an array of the leading dimension as declared in the calling function (`A_ld`, `B_ld`, or `C_ld`). The $i$th values of the `A`, `B`, and `C` are pointers to the arrays of data $A_i$, $B_i$, and $C_i$, respectively. For matrices in the $g$th group, the values of `A_ld[g]`, `B_ld[g]`, and

C_ld[g] correspond to the leading dimensions of the matrices $A_i$, $B_i$, and $C_i$, respectively. If there is only a single group of matrices, then the leading dimensions are specified by A_ld[0], B_ld[0], and C_ld[0] for all $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$ matrices, respectively.

*2.3.5 Arguments Defining the Input Scalar.* Arrays of scalars are named alpha and beta, where values at position $g$ correspond to the $\alpha$ and $\beta$ scalars for the BLAS operations in group $g$, involving matrices $A_i$, $B_i$, and $C_i$. If there is only a single group of matrices, then the scalars are given by alpha[0] and beta[0].

### 2.4 Error Handling Defined by the INFO Array

For the Batched BLAS the argument info is an input/output argument.

On input, the value of info[0] should have one of the following values:

- BblasErrorsReportAll, which indicates that all errors will be specified on output. The length of the info array should be greater than or equal to the ($\sum$ group_sizes[g])+1. One extra element is the first element that pertains to all groups and matrices and indicates if the routine finished successfully (info[0] is set to 0) or an error occurred (info[0] is set to non-zero value). Thus, there is no need to check all elements of info to find out if there were any issues.
- BblasErrorsReportGroup, which indicates that only a single error will be reported for each group, independently. The length of the info array should be greater than or equal to the group_count+1.
- BblasErrorsReportAny, which indicates that the occurrence of errors will be specified on output as a single integer value. The length of the info array should be at least one.
- BblasErrorsReportNone, which indicates that no errors will be reported on output. The length of the info array should be at least one.

The following values of arguments are invalid:

- Any value of the enumeration arguments side, transA, transB, trans, uplo, or diag whose meaning is not specified;
- If any of m, n, k, A_ld, B_ld, or C_ld is less than zero.

If no errors are detected, or if info[0] is BblasErrorsReportNone on input, then info[0] will be returned as zero. In the former case, the information returned to user clearly indicates the lack of errors during the call. In the latter case, there is no clear differentiation between success or error during execution and there is no indication what the error was—this behavior might be desired in some situations, often for performance reasons.

When info[0] is set to BblasErrorsReportNone on input, then no effort is made to report invalid input. Otherwise, if a routine is called with an invalid value for arguments group_count or group_sizes, then the routine will return an error in info[0]. The errors related to the other arguments are signaled with info[0] set to the number of the group in which the invalid argument was encountered. The returned group numbers are counted from 1, because the value of 0 is reserved as an indicator of lack of errors. In other words, if a Batched BLAS routine is called with an invalid value for any of its arguments (other than group_count and group_sizes) for matrix $i$ in group $g$, then the routine will return an error in position info[1+p+i] where $p$ is the total number of matrices in groups 0 through $g-1$:

$$p = \sum_{\ell=0}^{g-1} \texttt{group\_sizes}[\ell].$$

The errors could be related to arguments or to the numerical issues with matrix or vector data. The former is indicated by setting `info[1+p+i]` to $-a$ where $a$ is the number of the first invalid argument (counting from one with number 1: $-1$ for invalid first argument, $-2$–for the second, and so on). The positive value indicates numerical issues, for example the column number with zero pivot in LU factorization or number of diagonal entry that was negative during Cholesky factorization. Upon successful completion, the routine will store 0 in the appropriate location of the `info` array.

It should be noted that this is a departure from the BLAS themselves. The Level 1 BLAS had no error reporting, but the Level 2 and 3 BLAS use a routine XERBLA for error handling, which by default issues an error message and halts execution. To override the default behavior, it is necessary to implement a custom version of XERBLA. LAPACK introduced the argument `info`, but only as an output argument, and also uses XERBLA for error handling. With the extra complexity of the Batched BLAS and to be more in line with current coding practices, it was felt that the proposed error mechanism gives the required flexibility.

## 3 NUMERICAL STABILITY

Although it is intended that the Batched BLAS be implemented as efficiently as possible, as with the original BLAS, this should not be achieved at the cost of sacrificing numerical stability. See Section 7 of the original Level 3 BLAS specification [18] and Section 4.13 of the LAPACK User's Guide [5] for the definition of the numerical stability and expected bounds on the round-off errors in the computed results. It is worth noting that the standard numerical error analysis takes the size of the input matrix into consideration and those tend to be smaller for Batched BLAS. However, a counter-intuitive result holds that the error bounds are more precise for smaller matrices due to, partially, random cancellation of errors: With high probability, the error constant is proportional to $u\sqrt{n \log n}$ instead of the classic result of $nu$ [31] (where $n$ is the matrix size and $u$ is the machine precision).

## 4 SPECIFICATION OF BATCHED BLAS ROUTINES

### 4.1 Scope and Specifications of the Level 3 Batched BLAS

The Level 3 Batched BLAS routines described here have been derived in a fairly obvious manner from the interfaces of their corresponding Level 3 BLAS routines. The advantage in keeping the design of the software as consistent as possible with that of the BLAS is that it will be easier for users to replace their BLAS calls by calling the Batched BLAS when needed, and to remember the calling sequences and the parameter conventions. The operations proposed for the Level 3 Batched BLAS have an interface described as follows.

*4.1.1 General Matrix-matrix Products GEMM.* Depending on the values of `transA` and `transB`, this routine performs a batch of one of the matrix-matrix operations described below, for which $C$ is always an $m \times n$ matrix:

- $C \leftarrow \alpha \cdot A \times B + \beta \cdot C$, $A$ is $m \times k$, $B$ is $k \times n$
- $C \leftarrow \alpha \cdot A^T \times B + \beta \cdot C$, $A$ is $k \times m$, $B$ is $k \times n$
- $C \leftarrow \alpha \cdot A^H \times B + \beta \cdot C$, $A$ is $k \times m$, $B$ is $k \times n$
- $C \leftarrow \alpha \cdot A \times B^T + \beta \cdot C$, $A$ is $m \times k$, $B$ is $n \times k$

- $C \leftarrow \alpha \cdot A \times B^H + \beta \cdot C$, $A$ is $m \times k$, $B$ is $n \times k$
- $C \leftarrow \alpha \cdot A^T \times B^T + \beta \cdot C$, $A$ is $k \times m$, $B$ is $n \times k$
- $C \leftarrow \alpha \cdot A^H \times B^H + \beta \cdot C$, $A$ is $k \times m$, $B$ is $n \times k$.

C language declarations of GEMM functions in multiple precisions and argument domains are shown in Listing 1.

```
int BLAS_gemm_batched_r16(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Float16 * alpha, _Float16 ** A, int * A_ld, _Float16 ** B,
    int * B_ld, _Float16 * beta, _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c16(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Complex _Float16 * alpha, _Complex _Float16 ** A, int *
    A_ld, _Complex _Float16 ** B, int * B_ld, _Complex _Float16 * beta, _Complex _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_r32(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, float * alpha, float ** A, int * A_ld, float ** B, int * B_ld,
    float * beta, float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c32(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Complex float * alpha, _Complex float ** A, int * A_ld,
    _Complex float ** B, int * B_ld, _Complex float * beta, _Complex float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_r64(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, double * alpha, double ** A, int * A_ld, double ** B, int *
    B_ld, double * beta, double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c64(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Complex double * alpha, _Complex double ** A, int *
    A_ld, _Complex double ** B, int * B_ld, _Complex double * beta, _Complex double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_r128(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Float128 * alpha, _Float128 ** A, int * A_ld, _Float128 **
    B, int * B_ld, _Float128 * beta, _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c128(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Complex _Float128 * alpha, _Complex _Float128 ** A,
    int * A_ld, _Complex _Float128 ** B, int * B_ld, _Complex _Float128 * beta, _Complex _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
```

Listing 1. GEMM (general matrix-matrix multiply) interface in multiple precisions.

The `transA` and `transB` arrays are of size `group_count`. Each value defines the operation on matrices in the corresponding group. The `m`, `n`, and `k` arrays of integers are of size `group_count`, where each value defines the dimension of the operation on matrices in each corresponding group. The `alpha` and `beta` arrays of size `group_count` provide the scalars $\alpha$ and $\beta$, described in the equation above. They are of the same precision as the arrays `A`, `B`, and `C`. The arrays of pointers `A`, `B`, and `C` are of size `batch_count` and point to the matrices $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$. For matrices in group $g$, the size of the matrix $C_i$ is `m[g]*n[g]`. The sizes of the matrices $A_i$ and $B_i$ depend on `transA[g]` and `transB[g]`; their corresponding sizes are mentioned in the equation above. The arrays `A_ld`, `B_ld`, and `C_ld` of size `group_count` define the leading dimension of each of the matrices. For matrices in group $g$, $\{A_i$`[A_ld[g]][*]`$\}$, $\{B_i$`[B_ld[g]][*]`$\}$, $\{C_i$`[C_ld[g]][*]`$\}$, respectively.[1]

If there is only one group of matrices (`group_count == 1`), then only `transA[0]`, `transB[0]`, `m[0]`, `n[0]`, `k[0]`, `alpha[0]`, `A_ld[0]`, `B_ld[0]`, `beta[0]`, and `C_ld[0]` are used to specify the `gemm` parameters for the batch.

The `info` array defines the behavior of the error handler, as described in Section 2.4.

*4.1.2 Hermitian and Symmetric Matrix-matrix Products: HEMM and SYMM.* These routines perform a batch of matrix-matrix products, each expressed in one of the following forms, for which $B$ and $C$ are $m \times n$ matrices:

- $C \leftarrow \alpha \cdot A \times B + \beta \cdot C$ for `side==BlasLeft`, $A$ is $m \times m$
- $C \leftarrow \alpha \cdot B \times A + \beta \cdot C$ for `side==BlasRight`, $A$ is $n \times n$

where $A$ is either real symmetric (for `BLAS_symm_batched_r*` and `BLAS_hemm_batched_r*`), complex symmetric (for `BLAS_symm_batched_c*`), or complex Hermitian (for `BLAS_hemm_batched_c*`), and $\alpha$ and $\beta$ are real or complex scalars according to the data type suffix.

[1]The `layout` argument specifies whether leading dimension is across rows or columns.

C language declarations of SYMM functions in multiple precisions and argument domains are shown in Listing 2.

```
int BLAS_symm_batched_r16(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Float16 * alpha, _Float16 ** A, int * A_ld, _Float16 ** B, int * B_ld,
        _Float16 * beta, _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_symm_batched_c16(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Complex _Float16 * alpha, _Complex _Float16 ** A, int * A_ld,
        _Complex _Float16 ** B, int * B_ld, _Complex _Float16 * beta, _Complex _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_symm_batched_r32(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, float * alpha, float ** A, int * A_ld, float ** B, int * B_ld, float * beta,
        float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_symm_batched_c32(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Complex float * alpha, _Complex float ** A, int * A_ld, _Complex
        float ** B, int * B_ld, _Complex float * beta, _Complex float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_symm_batched_r64(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, double * alpha, double ** A, int * A_ld, double ** B, int * B_ld, double
        * beta, double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_symm_batched_c64(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Complex double * alpha, _Complex double ** A, int * A_ld,
        _Complex double ** B, int * B_ld, _Complex double * beta, _Complex double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_symm_batched_r128(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Float128 * alpha, _Float128 ** A, int * A_ld, _Float128 ** B, int *
        B_ld, _Float128 * beta, _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_symm_batched_c128(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Complex _Float128 * alpha, _Complex _Float128 ** A, int * A_ld,
        _Complex _Float128 ** B, int * B_ld, _Complex _Float128 * beta, _Complex _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
```

Listing 2. SYMM: Symmetric matrix-matrix product.

C language declarations of HEMM functions in multiple precisions and argument domains are shown in Listing 3.

```
int BLAS_hemm_batched_r16(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Float16 * alpha, _Float16 ** A, int * A_ld, _Float16 ** B, int * B_ld,
        _Float16 * beta, _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_hemm_batched_c16(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Complex _Float16 * alpha, _Complex _Float16 ** A, int * A_ld,
        _Complex _Float16 ** B, int * B_ld, _Complex _Float16 * beta, _Complex _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_hemm_batched_r32(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, float * alpha, float ** A, int * A_ld, float ** B, int * B_ld, float * beta,
        float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_hemm_batched_c32(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Complex float * alpha, _Complex float ** A, int * A_ld, _Complex
        float ** B, int * B_ld, _Complex float * beta, _Complex float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_hemm_batched_r64(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, double * alpha, double ** A, int * A_ld, double ** B, int * B_ld, double
        * beta, double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_hemm_batched_c64(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Complex double * alpha, _Complex double ** A, int * A_ld,
        _Complex double ** B, int * B_ld, _Complex double * beta, _Complex double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_hemm_batched_r128(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Float128 * alpha, _Float128 ** A, int * A_ld, _Float128 ** B, int *
        B_ld, _Float128 * beta, _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_hemm_batched_c128(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, int * m, int * n, _Complex _Float128 * alpha, _Complex _Float128 ** A, int * A_ld,
        _Complex _Float128 ** B, int * B_ld, _Complex _Float128 * beta, _Complex _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
```

Listing 3. HEMM: Hermitian matrix-matrix product.

The `side` array is of size `group_count` and each value defines the operation on matrices in each group as described in the equations above. The `uplo` array is of size `group_count` and defines whether the upper or the lower triangular part of the matrix is to be referenced. The `m` and `n` arrays of integers are of size `group_count` and define the dimension of the operation on each matrix. The `alpha` and `beta` arrays of size `group_count` provide the scalars $\alpha_i$ and $\beta_i$ described in the equation above. They are of the same precision as the arrays $A$, $B$, and $C$. The arrays `A`, `B`, and `C` are the arrays of pointers of size `batch_count` that point to the matrices $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$. For matrices in group $g$, the size of matrix $C_i$ is `m[g]*n[g]`. The sizes of the matrices $A_i$ and $B_i$ depend on side[g]; their corresponding sizes are mentioned in the equations above. The arrays `A_ld`, `B_ld`, and `C_ld` of size `group_count` define the leading dimension of each of the matrices $\{A_i$`[A_ld[g]][*]`$\}$, $\{B_i$`[B_ld[g]][*]`$\}$, $\{C_i$`[C_ld[g]][*]`$\}$, respectively.[2] The `info` array defines the behavior of the error handler, as described in Section 2.4.

*4.1.3 Rank-k Updates of a Symmetric/Hermitian Matrix HERK and SYRK.* These routines performs a batch of rank-k updates of real or complex symmetric (SYRK), or complex Hermitian (HERK) matrices in one of the following forms, for which $C$ is an $n \times n$ matrix:

[2]The `layout` argument specifies whether leading dimension is across rows or columns.

- $C \leftarrow \alpha \cdot A \times A^T + \beta \cdot C$ for `trans==BlasNoTrans` (syrk), $A$ is $n \times k$
- $C \leftarrow \alpha \cdot A^T \times A + \beta \cdot C$ for `trans==BlasTrans` (syrk), $A$ is $k \times n$
- $C \leftarrow \alpha \cdot A \times A^H + \beta \cdot C$ for `trans==BlasNoTrans` (herk), $A$ is $n \times k$
- $C \leftarrow \alpha \cdot A^H \times A + \beta \cdot C$ for `trans==BlasTrans` (herk), $A$ is $k \times n$

C language declarations of SYRK functions in multiple precisions and argument domains are shown in Listing 4.

```
int BLAS_syrk_batched_r16(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Float16 * alpha, _Float16 ** A, int * A_ld, _Float16 * beta, _Float16 ** C,
    int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syrk_batched_c16(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex _Float16 * alpha, _Complex _Float16 ** A, int * A_ld,
    _Complex _Float16 * beta, _Complex _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syrk_batched_r32(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, float * alpha, float ** A, int * A_ld, float * beta, float ** C, int * C_ld, int
    group_count, int * group_sizes, int * info);
int BLAS_syrk_batched_c32(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex float * alpha, _Complex float ** A, int * A_ld, _Complex float
    * beta, _Complex float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syrk_batched_r64(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, double * alpha, double ** A, int * A_ld, double * beta, double ** C, int *
    C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syrk_batched_c64(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex double * alpha, _Complex double ** A, int * A_ld, _Complex
    double * beta, _Complex double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syrk_batched_r128(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Float128 * alpha, _Float128 ** A, int * A_ld, _Float128 * beta,
    _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syrk_batched_c128(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex _Float128 * alpha, _Complex _Float128 ** A, int * A_ld,
    _Complex _Float128 * beta, _Complex _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
```

Listing 4. SYRK: Symmetric rank-k matrix-matrix product.

The `uplo` array is of size `group_count` and defines whether the upper or the lower triangular part of the matrix is to be referenced. The `trans` array is of size `group_count`, where each value defines the operation on matrices in each group. In the complex case, `trans == BlasConjTrans` is not allowed in syrk case. The `n` and `k` arrays of integers are of size `batch_count` and define the dimensions of the matrices in each group. The `alpha` and `beta` arrays of size `group_count` provide the scalars $\alpha$ and $\beta$ described in the equation above. They are of the same precision as the arrays $A_i$ and $C_i$ . The arrays of pointers `A` and `C` are of size `batch_count` and point to the matrices $\{A_i\}$ and $\{C_i\}$. For matrices in group $g$, the size of matrix $C_i$ is `m[g]*n[g]`. All matrices $\{C_i\}$ are either real or complex symmetric. The size of the matrix $A_i$ depends on `trans[g]`; its corresponding size is mentioned in the equation above. The arrays `A_ld` and `C_ld` of size `group_count` define the leading dimension of each of the matrices $\{A_i$`[A_ld[g]][*]`$\}$, $\{C_i$`[C_ld[g]][*]`$\}$, respectively.[3] The `info` array defines the behavior of the error handler, as described in Section 2.4.

C language declarations of HERK functions in multiple precisions and argument domains are shown in Listing 5.

```
int BLAS_herk_batched_c16(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Float16 * alpha, _Complex _Float16 ** A, int * A_ld, _Float16 * beta,
    _Complex _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_herk_batched_c32(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, float * alpha, _Complex float ** A, int * A_ld, float * beta, _Complex
    float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_herk_batched_c64(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, double * alpha, _Complex double ** A, int * A_ld, double * beta,
    _Complex double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_herk_batched_c128(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Float128 * alpha, _Complex _Float128 ** A, int * A_ld, _Float128 *
    beta, _Complex _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
```

Listing 5. HERK: Hermitian rank-k matrix-matrix product.

This routine is only available for the complex precisions. It has the same parameters as syrk batch except that the `trans == BlasTrans` is not allowed in herk batch and that `alpha` and `beta` are real. The matrices $\{C_i\}$ are complex Hermitian. The `info` array defines the behavior of the error handler, as described in Section 2.4.

[3]The `layout` argument specifies whether leading dimension is across rows or columns.

*4.1.4 Rank-2k Updates of a Symmetric/Hermitian Matrix HER2K and SYR2K.* This routine performs batch rank-2k updates on real or complex symmetric (SYR2K), or complex Hermitian (HER2K) matrices of the following forms, for which $C$ is an $m \times n$ matrix:

- $C \leftarrow \alpha \cdot A \times B^T + \alpha \cdot B \times A^T + \beta \cdot C$ for `trans == BlasNoTrans` (syr2k), $A, B$ are $n \times k$
- $C \leftarrow \alpha \cdot A^T \times B + \alpha \cdot B^T \times A + \beta \cdot C$ for `trans == BlasTrans` (syr2k), $A, B$ are $k \times n$
- $C \leftarrow \alpha \cdot A \times B^H + \alpha \cdot B \times A^H + \beta \cdot C$ for `trans == BlasNoTrans` (her2k), $A, B$ are $n \times k$
- $C \leftarrow \alpha \cdot A^H \times B + \alpha \cdot B^H \times A + \beta \cdot C$ for `trans == BlasConjTrans` (her2k), $A, B$ are $k \times n$

C language declarations of SYR2K functions in multiple precisions and argument domains are shown in Listing 6.

```
int BLAS_syr2k_batched_r16(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Float16 * alpha, _Float16 ** A, int * A_ld, _Float16 * beta, _Float16 **
    C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syr2k_batched_c16(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex _Float16 * alpha, _Complex _Float16 ** A, int * A_ld,
    _Complex _Float16 * beta, _Complex _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syr2k_batched_r32(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, float * alpha, float ** A, int * A_ld, float * beta, float ** C, int * C_ld, int
    group_count, int * group_sizes, int * info);
int BLAS_syr2k_batched_c32(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex float * alpha, _Complex float ** A, int * A_ld, _Complex
    float * beta, _Complex float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syr2k_batched_r64(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, double * alpha, double ** A, int * A_ld, double * beta, double ** C, int *
    C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syr2k_batched_c64(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex double * alpha, _Complex double ** A, int * A_ld, _Complex
    double * beta, _Complex double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syr2k_batched_r128(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Float128 * alpha, _Float128 ** A, int * A_ld, _Float128 * beta,
    _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_syr2k_batched_c128(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex _Float128 * alpha, _Complex _Float128 ** A, int * A_ld,
    _Complex _Float128 * beta, _Complex _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
```

Listing 6. SYR2K: Symmetric rank-2k matrix-matrix product.

The `uplo` array is of size `group_count` and defines whether the upper or the lower triangular part of the matrix is to be referenced. The `trans` array is of size `group_count`, where each value defines the operation on matrices in each group. In the complex case, `trans == BlasConjTrans` is not allowed in syr2k batch. The `n` and `k` arrays of integers are of size `group_count` and define the dimensions of the matrices in each group. The `alpha` and `beta` arrays of size `group_count` provide the scalars $\alpha$ and $\beta$ described in the equations above. They are of the same precision as the arrays `A`, `B`, and `C`. The arrays `A`, `B`, and `C` are the arrays of pointers of size `batch_count` that point to the matrices $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$. For matrices in group $g$, the size of matrix $C_i$ is `m[g]*n[g]`. All matrices $\{C_i\}$ are either real or complex symmetric. The size of the matrices $A_i$ and $B_i$ depends on `trans[g]`; its corresponding size is mentioned in the equation above. The arrays `A_ld`, `B_ld`, and `C_ld` of size `group_count` define the leading dimension of the matrices $\{A_i$`[A_ld[g]][*]`$\}$, $\{B_i$`[B_ld[g]][*]`$\}$, $\{C_i$`[C_ld[g]][*]`$\}$, respectively.[4] The `info` array defines the behavior of the error handler, as described in Section 2.4.

C language declarations of HER2K functions in multiple precisions and argument domains are shown in Listing 7.

```
int BLAS_her2k_batched_c16(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex _Float16 * alpha, _Complex _Float16 ** A, int * A_ld,
    _Complex _Float16 ** B, _Complex _Float16 ** B, int * B_ld, _Complex _Float16 * beta, _Complex _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info
    );
int BLAS_her2k_batched_c32(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex float * alpha, _Complex float ** A, int * A_ld, _Complex
    float ** B, _Complex float ** B, int * B_ld, _Complex float * beta, _Complex float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_her2k_batched_c64(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex double * alpha, _Complex double ** A, int * A_ld, _Complex
    double ** B, _Complex double ** B, int * B_ld, _Complex double * beta, _Complex double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_her2k_batched_c128(BLAS_Layout layout, BLAS_UpLo * uplo, BLAS_Op * trans, int * n, int * k, _Complex _Float128 * alpha, _Complex _Float128 ** A, int * A_ld,
    _Complex _Float128 ** B, _Complex _Float128 ** B, int * B_ld, _Complex _Float128 * beta, _Complex _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int *
    info);
```

Listing 7. HER2K: Hermitian rank-2k matrix-matrix product.

[4]The `layout` argument specifies whether leading dimension is across rows or columns.

This routine is only available for the complex precision. It has the same parameters as the syr2k batch routine except that the `trans == BlasTrans` is not allowed in her2k batch. The matrices $\{C_i\}$ are complex Hermitian. The `info` array defines the behavior of the error handler, as described in Section 2.4.

*4.1.5 Multiplying a Matrix by a Triangular Matrix TRMM.* These routines perform a batch of one of the following matrix-matrix products, where $A$ is an $m \times m$ upper or lower triangular matrix, $B$ is an $m \times n$ matrix, and $\alpha$ is a scalar:

- $B \leftarrow \alpha \cdot A \times B$ for `side == BlasLeft` and `trans == BlasNoTrans`
- $B \leftarrow \alpha \cdot A^T \times B$ for `side == BlasLeft` and `trans == BlasTrans`
- $B \leftarrow \alpha \cdot A^H \times B$ for `side == BlasLeft` and `trans == BlasConjTrans`
- $B \leftarrow \alpha \cdot B \times A$ for `side == BlasRight` and `trans == BlasNoTrans`
- $B \leftarrow \alpha \cdot B \times A^T$ for `side == BlasRight` and `trans == BlasTrans`
- $B \leftarrow \alpha \cdot B \times A^H$ for `side == BlasRight` and `trans == BlasConjTrans`

C language declarations of TRMM functions in multiple precisions and argument domains are shown in Listing 8.

```
int BLAS_trmm_batched_r16(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Float16 * alpha, _Float16 **
    A, int * A_ld, _Float16 ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trmm_batched_c16(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Complex _Float16 * alpha,
    _Complex _Float16 ** A, int * A_ld, _Complex _Float16 ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trmm_batched_r32(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, float * alpha, float ** A, int *
    A_ld, float ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trmm_batched_c32(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Complex float * alpha,
    _Complex float ** A, int * A_ld, _Complex float ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trmm_batched_r64(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, double * alpha, double ** A,
    int * A_ld, double ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trmm_batched_c64(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Complex double * alpha,
    _Complex double ** A, int * A_ld, _Complex double ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trmm_batched_r128(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Float128 * alpha,
    _Float128 ** A, int * A_ld, _Float128 ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trmm_batched_c128(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Complex _Float128 *
    alpha, _Complex _Float128 ** A, int * A_ld, _Complex _Float128 ** B, int * B_ld, int group_count, int * group_sizes, int * info);
```

Listing 8. TRMM: Triangular matrix-matrix product.

The `side` array is of size `group_count` and each value defines the operation on each matrix as described in the equations above. The `uplo` array is of size `group_count` and defines whether the upper or the lower triangular part of the matrices $\{A_i\}$ are to be referenced. The `trans` is an array of size `group_count`, where each value defines the operation on each matrix. The `diag` array is of size `group_count`, where each value defines whether the corresponding matrix $A$ is assumed to be unit or non-unit triangular. The `m` and `n` arrays of integers are of size `group_count` and define the dimension of the operation on each matrix. The `alpha` array of size `group_count` provides the scalars $\alpha$ described in the equation above. It is of the same precision as the arrays `A` and `B`. The arrays of pointers `A` and `B` are of size `batch_count` and point to the matrices $\{A_i\}$ and $\{B_i\}$. For matrices in group $g$, the size of matrix $B_i$ is `m[g]*n[g]`. The size of matrix $A_i$ depends on `side[g]`; its corresponding size is mentioned in the equation above. The arrays `A_ld` and `B_ld` of size `group_count` define the leading dimension of the $\{A_i$`[A_ld[g]][*]`$\}$, and $\{B_i$`[B_ld[g]][*]`$\}$ matrices, respectively.[5] The `info` array defines the behavior of the error handler, as described in Section 2.4.

[5]The `layout` argument specifies whether leading dimension is across rows or columns.

*4.1.6 Solving Triangular Systems of Equations with Multiple Right-hand Sides TRSM.* This routine solves a batch of one of the following matrix equations, where the matrix $A$ is an $m \times m$ upper or lower triangular matrix, $B$ is an $m \times n$ matrix, and $\alpha$ is scalar:

- $B \leftarrow \alpha \cdot A^{-1} \times B$ for `side == BlasLeft` and `trans == BlasNoTrans`
- $B \leftarrow \alpha \cdot A^{-T} \times B$ for `side == BlasLeft` and `trans == BlasTrans`
- $B \leftarrow \alpha \cdot A^{-H} \times B$ for `side == BlasLeft` and `trans == BlasConjTrans`
- $B \leftarrow \alpha \cdot B \times A^{-1}$ for `side == BlasRight` and `trans == BlasNoTrans`
- $B \leftarrow \alpha \cdot B \times A^{-T}$ for `side == BlasRight` and `trans == BlasTrans`
- $B \leftarrow \alpha \cdot B \times A^{-H}$ for `side == BlasRight` and `trans == BlasConjTrans`

C language declarations of TRSM functions in multiple precisions and argument domains are shown in Listing 9.

```
int BLAS_trsm_batched_r16(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Float16 * alpha, _Float16 **
        A, int * A_ld, _Float16 ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trsm_batched_c16(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Complex _Float16 * alpha,
        _Complex _Float16 ** A, int * A_ld, _Complex _Float16 ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trsm_batched_r32(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, float * alpha, float ** A, int *
        A_ld, float ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trsm_batched_c32(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Complex float * alpha,
        _Complex float ** A, int * A_ld, _Complex float ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trsm_batched_r64(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, double * alpha, double ** A,
        int * A_ld, double ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trsm_batched_c64(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Complex double * alpha,
        _Complex double ** A, int * A_ld, _Complex double ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trsm_batched_r128(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Float128 * alpha, _Float128
        ** A, int * A_ld, _Float128 ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int BLAS_trsm_batched_c128(BLAS_Layout layout, BLAS_Side * side, BLAS_UpLo * uplo, BLAS_Op * trans, BLAS_Diagonal * diag, int * m, int * n, _Complex _Float128 * alpha
        , _Complex _Float128 ** A, int * A_ld, _Complex _Float128 ** B, int * B_ld, int group_count, int * group_sizes, int * info);
```

Listing 9. TRSM: Triangular matrix-matrix solve.

The `side` array is of size `group_count`, where each value defines the operation on each matrix as described in the equation above. The `uplo` array is of size `group_count` and defines whether the upper or the lower triangular part of the matrices $\{A_i\}$ are to be referenced. The `trans` array is of size `group_count`, where each value defines the operation on each matrix. The `diag` array is of size `group_count`, where each value defines whether the corresponding matrix $A$ is assumed to be unit or non-unit triangular. The `m` and `n` arrays of integers are of size `group_count` and define the dimension of the operation on each matrix. The `alpha` array of size `group_count` provides the scalars $\alpha$ described in the equation above. It is of the same precision as the arrays $A$ and $B$. The arrays of pointers `A` and `B` are of size `batch_count` and point to the matrices $\{A_i\}$ and $\{B_i\}$. For matrices in group $g$, the size of matrix $B_i$ is `m[g]*n[g]`. The size of the matrix $A_i$ depends on `side[g]`; its corresponding size is mentioned in the equation above. The arrays `A_ld` and `B_ld` of size `group_count` define the leading dimension of the matrices $\{A_i$`[A_ld[g]][*]`$\}$, and $\{B_i$`[B_ld[g]][*]`$\}$, respectively.[6] The `info` array defines the behavior of the error handler, as described in Section 2.4.

## 4.2 Specifications of the Level 1 and Level 2 Batched BLAS

Similarly to the derivation of a Level 3 Batched BLAS from the Level 3 BLAS, we can derive Level 1 and Level 2 Batched BLAS from the corresponding Level 1 and Level 2 BLAS routines. Although we do not provide reference versions, we give as examples the Level 1 AXPY: $y \leftarrow \alpha \cdot x + y$ (Listing 10) and the Level 2 GEMV: $y \leftarrow \alpha \cdot A \times x + \beta \cdot y$ APIs (Listing 11.) The number of routines in Level

[6]The `layout` argument specifies whether leading dimension is across rows or columns.

1 and 2 BLAS is very large and providing all the details along the lines of what was done for Level 3 BLAS would easily double the size of this document without adding substantive content. Consequently, the details are left out.

```
int BLAS_axpy_batched_r16(int * n, _Float16 * alpha, _Float16 ** X, int * X_inc, _Float16 ** Y, int * Y_inc, int group_count, int * group_sizes, int * info);
int BLAS_axpy_batched_c16(int * n, _Complex _Float16 * alpha, _Complex _Float16 ** X, int * X_inc, _Complex _Float16 ** Y, int * Y_inc, int group_count, int * group_sizes,
    int * info);
int BLAS_axpy_batched_r32(int * n, float * alpha, float ** X, int * X_inc, float ** Y, int * Y_inc, int group_count, int * group_sizes, int * info);
int BLAS_axpy_batched_c32(int * n, _Complex float * alpha, _Complex float ** X, int * X_inc, _Complex float ** Y, int * Y_inc, int group_count, int * group_sizes, int * info);
int BLAS_axpy_batched_r64(int * n, double * alpha, double ** X, int * X_inc, double ** Y, int * Y_inc, int group_count, int * group_sizes, int * info);
int BLAS_axpy_batched_c64(int * n, _Complex double * alpha, _Complex double ** X, int * X_inc, _Complex double ** Y, int * Y_inc, int group_count, int * group_sizes, int *
    info);
int BLAS_axpy_batched_r128(int * n, _Float128 * alpha, _Float128 ** X, int * X_inc, _Float128 ** Y, int * Y_inc, int group_count, int * group_sizes, int * info);
int BLAS_axpy_batched_c128(int * n, _Complex _Float128 * alpha, _Complex _Float128 ** X, int * X_inc, _Complex _Float128 ** Y, int * Y_inc, int group_count, int *
    group_sizes, int * info);
```

Listing 10. AXPY: Scaling a vector and adding another vector.

Fox AXPY in Listing 10, arrays `incx[g]` and `incy[g]` from the $g$th group BLAS operation must not be zero and specify the increments for the elements of `X[i]` and `Y[i]`, respectively, in group $g$.

```
int BLAS_gemv_batched_r16(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Float16 * alpha, _Float16 ** A, int * A_ld, _Float16 * beta, _Float16 ** Y, int * Y_inc, int
    group_count, int * group_sizes, int * info);
int BLAS_gemv_batched_c16(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Complex _Float16 * alpha, _Complex _Float16 ** A, int * A_ld, _Complex _Float16 *
    beta, _Complex _Float16 ** Y, int * Y_inc, int group_count, int * group_sizes, int * info);
int BLAS_gemv_batched_r32(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, float * alpha, float ** A, int * A_ld, float * beta, float ** Y, int * Y_inc, int group_count,
    int * group_sizes, int * info);
int BLAS_gemv_batched_c32(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Complex float * alpha, _Complex float ** A, int * A_ld, _Complex float * beta,
    _Complex float ** Y, int * Y_inc, int group_count, int * group_sizes, int * info);
int BLAS_gemv_batched_r64(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, double * alpha, double ** A, int * A_ld, double * beta, double ** Y, int * Y_inc, int
    group_count, int * group_sizes, int * info);
int BLAS_gemv_batched_c64(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Complex double * alpha, _Complex double ** A, int * A_ld, _Complex double * beta,
    _Complex double ** Y, int * Y_inc, int group_count, int * group_sizes, int * info);
int BLAS_gemv_batched_r128(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Float128 * alpha, _Float128 ** A, int * A_ld, _Float128 * beta, _Float128 ** Y, int *
    Y_inc, int group_count, int * group_sizes, int * info);
int BLAS_gemv_batched_c128(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Complex _Float128 * alpha, _Complex _Float128 ** A, int * A_ld, _Complex
    _Float128 * beta, _Complex _Float128 ** Y, int * Y_inc, int group_count, int * group_sizes, int * info);
```

Listing 11. GEMV: General matrix-vector product.

For GEMV in Listing 11, array `incy[g]` at the $g$th position must not be zero and specifies the increment for the elements of `Y[i]` in group $g$.

## 5 SPECIFICATION OF BATCHED LAPACK ROUTINES

The batched approach to BLAS can be applied to higher-level libraries, and in particular to LAPACK. In this extension, the Batched LAPACK routines can be derived from the interfaces of their corresponding non-batched LAPACK routines, similarly to the derivation of Batched BLAS from the classic non-batched BLAS. For example, for the batched LU factorization routine of an $m \times n$ matrix A, with partial pivoting based on row interchanges, based on the LAPACK routine GETRF. Due to space restrictions and a large number of LAPACK routines, we limit to only a few examples as we did for Level 1 and 2 BLAS.

C language declarations of GETRF functions in multiple precisions and argument domains are shown in Listing 12.

```
int LAPACK_getrf_batched_r16(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Float16 ** A, int * A_ld, int ** piv, int group_count, int * group_sizes, int * info);
int LAPACK_getrf_batched_c16(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Complex _Float16 ** A, int * A_ld, int ** piv, int group_count, int * group_sizes, int
    * info);
int LAPACK_getrf_batched_r32(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, float ** A, int * A_ld, int ** piv, int group_count, int * group_sizes, int * info);
int LAPACK_getrf_batched_c32(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Complex float ** A, int * A_ld, int ** piv, int group_count, int * group_sizes, int *
    info);
int LAPACK_getrf_batched_r64(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, double ** A, int * A_ld, int ** piv, int group_count, int * group_sizes, int * info);
int LAPACK_getrf_batched_c64(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Complex double ** A, int * A_ld, int ** piv, int group_count, int * group_sizes, int *
    info);
int LAPACK_getrf_batched_r128(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Float128 ** A, int * A_ld, int ** piv, int group_count, int * group_sizes, int * info);
int LAPACK_getrf_batched_c128(BLAS_Layout layout, BLAS_Op * A_trans, int * m, int * n, _Complex _Float128 ** A, int * A_ld, int ** piv, int group_count, int * group_sizes,
    int * info);
```

Listing 12. GETRF: LU matrix factorization.

C language declarations of GETRS functions in multiple precisions and argument domains are shown in Listing 13.

```
int LAPACK_getrs_batched_r16(BLAS_Layout layout, BLAS_Op * A_trans, int * n, int * nrhs, _Float16 ** A, int * A_ld, int ** piv, _Float16 ** B, int * B_ld, int group_count, int *
    group_sizes, int * info);
int LAPACK_getrs_batched_c16(BLAS_Layout layout, BLAS_Op * A_trans, int * n, int * nrhs, _Complex _Float16 ** A, int * A_ld, int ** piv, _Complex _Float16 ** B, int * B_ld,
    int group_count, int * group_sizes, int * info);
int LAPACK_getrs_batched_r32(BLAS_Layout layout, BLAS_Op * A_trans, int * n, int * nrhs, float ** A, int * A_ld, int ** piv, float ** B, int * B_ld, int group_count, int *
    group_sizes, int * info);
int LAPACK_getrs_batched_c32(BLAS_Layout layout, BLAS_Op * A_trans, int * n, int * nrhs, _Complex float ** A, int * A_ld, int ** piv, _Complex float ** B, int * B_ld, int
    group_count, int * group_sizes, int * info);
int LAPACK_getrs_batched_r64(BLAS_Layout layout, BLAS_Op * A_trans, int * n, int * nrhs, double ** A, int * A_ld, int ** piv, double ** B, int * B_ld, int group_count, int *
    group_sizes, int * info);
int LAPACK_getrs_batched_c64(BLAS_Layout layout, BLAS_Op * A_trans, int * n, int * nrhs, _Complex double ** A, int * A_ld, int ** piv, _Complex double ** B, int * B_ld, int
    group_count, int * group_sizes, int * info);
int LAPACK_getrs_batched_r128(BLAS_Layout layout, BLAS_Op * A_trans, int * n, int * nrhs, _Float128 ** A, int * A_ld, int ** piv, _Float128 ** B, int * B_ld, int group_count,
    int * group_sizes, int * info);
int LAPACK_getrs_batched_c128(BLAS_Layout layout, BLAS_Op * A_trans, int * n, int * nrhs, _Complex _Float128 ** A, int * A_ld, int ** piv, _Complex _Float128 ** B, int *
    B_ld, int group_count, int * group_sizes, int * info);
```

Listing 13. GETRS: LU matrix solve.

C language declarations of GESV functions in multiple precisions and argument domains are shown in Listing 14.

```
int LAPACK_gesv_batched_r16(BLAS_Layout layout, int * n, int * nrhs, _Float16 ** A, int * A_ld, int ** piv, _Float16 ** B, int * B_ld, int group_count, int * group_sizes, int * info
    );
int LAPACK_gesv_batched_c16(BLAS_Layout layout, int * n, int * nrhs, _Complex _Float16 ** A, int * A_ld, int ** piv, _Complex _Float16 ** B, int * B_ld, int group_count, int *
    group_sizes, int * info);
int LAPACK_gesv_batched_r32(BLAS_Layout layout, int * n, int * nrhs, float ** A, int * A_ld, int ** piv, float ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int LAPACK_gesv_batched_c32(BLAS_Layout layout, int * n, int * nrhs, _Complex float ** A, int * A_ld, int ** piv, _Complex float ** B, int * B_ld, int group_count, int *
    group_sizes, int * info);
int LAPACK_gesv_batched_r64(BLAS_Layout layout, int * n, int * nrhs, double ** A, int * A_ld, int ** piv, double ** B, int * B_ld, int group_count, int * group_sizes, int * info);
int LAPACK_gesv_batched_c64(BLAS_Layout layout, int * n, int * nrhs, _Complex double ** A, int * A_ld, int ** piv, _Complex double ** B, int * B_ld, int group_count, int *
    group_sizes, int * info);
int LAPACK_gesv_batched_r128(BLAS_Layout layout, int * n, int * nrhs, _Float128 ** A, int * A_ld, int ** piv, _Float128 ** B, int * B_ld, int group_count, int * group_sizes, int *
    info);
int LAPACK_gesv_batched_c128(BLAS_Layout layout, int * n, int * nrhs, _Complex _Float128 ** A, int * A_ld, int ** piv, _Complex _Float128 ** B, int * B_ld, int group_count,
    int * group_sizes, int * info);
```

Listing 14. GESV: LU matrix factorization and solve.

C language declarations of POTRF functions in multiple precisions and argument domains are shown in Listing 15.

```
int LAPACK_potrf_batched_r16(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, _Float16 ** A, int * A_ld, int group_count, int * group_sizes, int * info);
int LAPACK_potrf_batched_c16(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, _Complex _Float16 ** A, int * A_ld, int group_count, int * group_sizes, int * info);
int LAPACK_potrf_batched_r32(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, float ** A, int * A_ld, int group_count, int * group_sizes, int * info);
int LAPACK_potrf_batched_c32(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, _Complex float ** A, int * A_ld, int group_count, int * group_sizes, int * info);
int LAPACK_potrf_batched_r64(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, double ** A, int * A_ld, int group_count, int * group_sizes, int * info);
int LAPACK_potrf_batched_c64(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, _Complex double ** A, int * A_ld, int group_count, int * group_sizes, int * info);
int LAPACK_potrf_batched_r128(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, _Float128 ** A, int * A_ld, int group_count, int * group_sizes, int * info);
int LAPACK_potrf_batched_c128(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, _Complex _Float128 ** A, int * A_ld, int group_count, int * group_sizes, int * info);
```

Listing 15. POTRF: Cholesky matrix factorization.

C language declarations of POTRS functions in multiple precisions and argument domains are shown in Listing 16.

```
int LAPACK_potrs_batched_r16(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Float16 ** A, int * A_ld, _Float16 ** B, int * B_ld, int group_count, int *
        group_sizes, int * info);
int LAPACK_potrs_batched_c16(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Complex _Float16 ** A, int * A_ld, _Complex _Float16 ** B, int * B_ld, int
        group_count, int * group_sizes, int * info);
int LAPACK_potrs_batched_r32(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, float ** A, int * A_ld, float ** B, int * B_ld, int group_count, int * group_sizes, int *
        info);
int LAPACK_potrs_batched_c32(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Complex float ** A, int * A_ld, _Complex float ** B, int * B_ld, int group_count,
        int * group_sizes, int * info);
int LAPACK_potrs_batched_r64(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, double ** A, int * A_ld, double ** B, int * B_ld, int group_count, int * group_sizes,
        int * info);
int LAPACK_potrs_batched_c64(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Complex double ** A, int * A_ld, _Complex double ** B, int * B_ld, int
        group_count, int * group_sizes, int * info);
int LAPACK_potrs_batched_r128(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Float128 ** A, int * A_ld, _Float128 ** B, int * B_ld, int group_count, int *
        group_sizes, int * info);
int LAPACK_potrs_batched_c128(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Complex _Float128 ** A, int * A_ld, _Complex _Float128 ** B, int * B_ld, int
        group_count, int * group_sizes, int * info);
```

Listing 16. POTRS: Cholesky matrix solve.

C language declarations of POSV functions in multiple precisions and argument domains are shown in Listing 17.

```
int LAPACK_posv_batched_r16(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Float16 ** A, int * A_ld, _Float16 ** B, int * B_ld, int group_count, int * group_sizes
        , int * info);
int LAPACK_posv_batched_c16(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Complex _Float16 ** A, int * A_ld, _Complex _Float16 ** B, int * B_ld, int
        group_count, int * group_sizes, int * info);
int LAPACK_posv_batched_r32(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, float ** A, int * A_ld, float ** B, int * B_ld, int group_count, int * group_sizes, int *
        info);
int LAPACK_posv_batched_c32(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Complex float ** A, int * A_ld, _Complex float ** B, int * B_ld, int group_count, int
         * group_sizes, int * info);
int LAPACK_posv_batched_r64(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, double ** A, int * A_ld, double ** B, int * B_ld, int group_count, int * group_sizes,
        int * info);
int LAPACK_posv_batched_c64(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Complex double ** A, int * A_ld, _Complex double ** B, int * B_ld, int
        group_count, int * group_sizes, int * info);
int LAPACK_posv_batched_r128(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Float128 ** A, int * A_ld, _Float128 ** B, int * B_ld, int group_count, int *
        group_sizes, int * info);
int LAPACK_posv_batched_c128(BLAS_Layout layout, BLAS_UpLo * uplo, int * n, int * nrhs, _Complex _Float128 ** A, int * A_ld, _Complex _Float128 ** B, int * B_ld, int
        group_count, int * group_sizes, int * info);
```

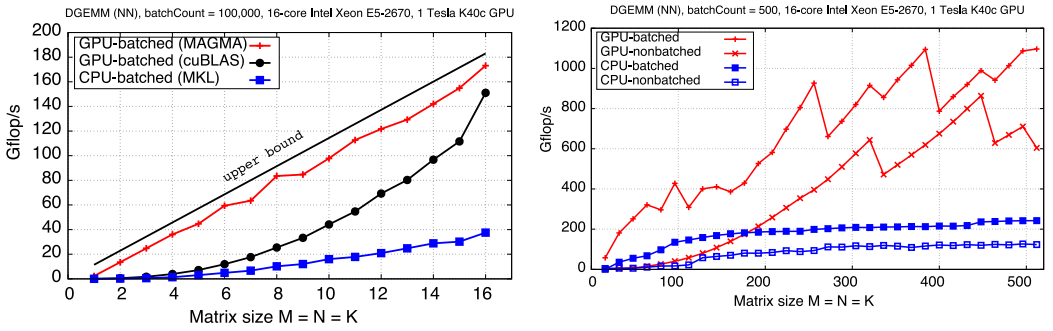
Listing 17. POSV: Cholesky matrix factorization and solve.

## 6 IMPLEMENTATION OF THE BATCHED BLAS

The key to efficient BLAS implementation is to hierarchically block the BLAS computation into tasks that operate on data that fits into the corresponding hierarchical memory levels of the computer architecture at hand (see for example the K40 GPU memory hierarchy in Figure 1). The goal is to reduce expensive data movements by loading the data required for a task into fast memory and reusing it in computations from there as many times as possible. An example for achieving this on Level 3 BLAS for GPUs is the MAGMA GEMM [42]. This GEMM harnesses hierarchical blocking on the memory levels available on the Kepler GPUs, including a new register blocking, and is still in use on current GPUs. Hierarchical blocking and communications are needed for optimal performance even for memory-bound computations like Level 2 BLAS, e.g., see the matrix-vector kernels developed and optimized for Xeon Phi architectures [34].

Thus, splitting an algorithm into hierarchical tasks that block the computation over the available memory hierarchies (to reduce data movement) is essential for implementing high-performance BLAS.

Details on how these techniques can be extended to develop high-performance Batched BLAS, and in particular, the extensively used batch GEMM, can be found elsewhere [2]. The routines developed thereby are released through the MAGMA library, providing a model Batched BLAS




(a) Performance on matrices of sizes less than 32.

(b) Performance on larger-sized matrices.

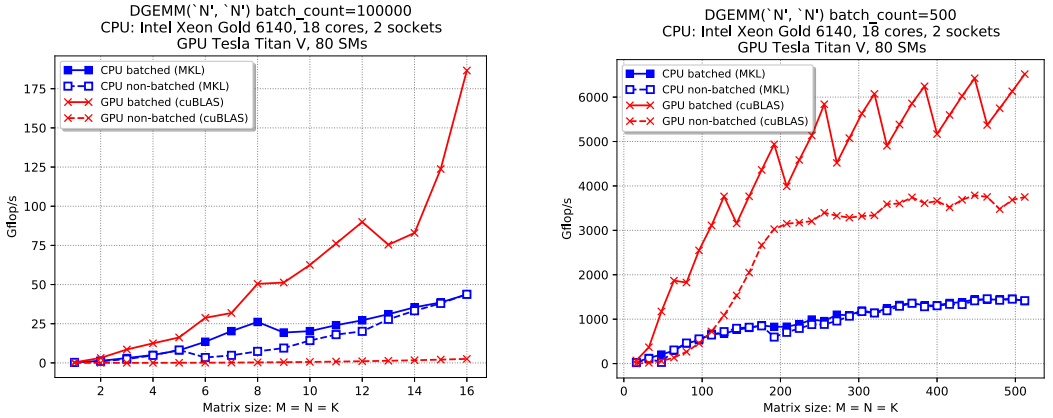
Fig. 2. Performance of batch DGEMM on K40c GPU and dual 16-core Intel Xeon ES-2670 (Sandy Bridge) 2.60 GHz CPU in MAGMA, NVIDIA cuBLAS, and Intel MKL.

implementation for GPUs. The goal of this model implementation and the API proposed here is that similarly to BLAS, hardware vendors adopt the Batched BLAS API and maintain highly tuned implementations for their corresponding platforms.

The MAGMA performance is shown in Figure 2. Besides hierarchical blocking, specialized kernels are designed for various sizes, and a comprehensive autotuning process is applied to all kernels. For very small matrix sizes, e.g., sub-vector/warp in size, the performance is memory-bound. Techniques such as grouping several GEMMs to be executed on the same multiprocessor, vectorization across GEMMs, along with data prefetching optimizations, are used to reach 90+% of the theoretical peak on either multicore CPUs or GPUs [1, 39] (see Figure 2, left). This performance is obtained on CPUs using compiler intrinsics, while on GPUs peak still can be reached by coding in CUDA. For larger sizes on GPUs, e.g., up to about 200 on K40 GPUs, best results are obtained by mapping a single GEMM (from the batch) to a multiprocessor, where the usual hierarchical blocking is applied. For larger matrix sizes, streaming is applied to GEMMs tuned for larger sizes. This results in using more than one multiprocessor for a single GEMM (see Figure 2, right). For these sizes, similar to CPUs, coding multilevel blocking types of algorithms on GPUs must be in native machine language to overcome some limitations of the CUDA compiler or warp scheduler (or both) [44]. Assembly implementations [23, 37] are used today in cuBLAS for Kepler and Maxwell GPUs to obtain higher performance than corresponding CUDA codes. Running these types of implementations through different streams gives the currently best performing batch implementations for large size matrices. See Figures 3 and 4 for more details.

## 7 BATCHED BLAS IN APPLICATIONS

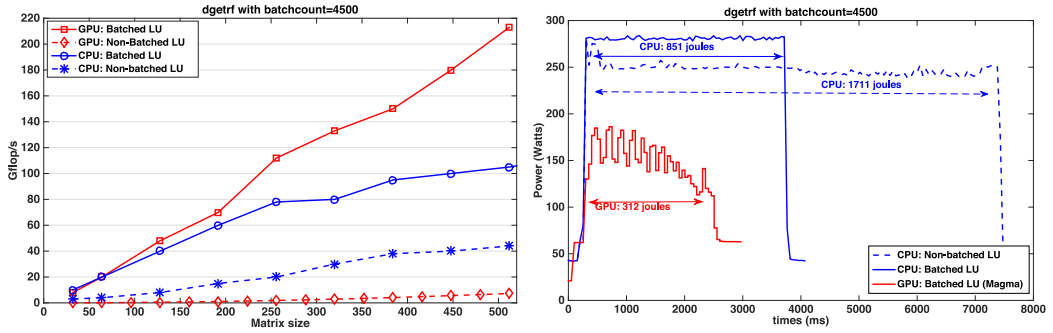
Many modern applications are cast in terms of a solution of many small matrix operations; that is, at some point in their execution, programs must perform a computation that is cumulatively very large, but whose individual parts are very small; when such operations are implemented naïvely using the typical approach, they perform poorly. Applications that suffer from this problem include those that require tensor contractions (as in the quantum Hall effect), astrophysics [40], metabolic networks [36], CFD and resulting PDEs through direct and multifrontal solvers [48], high-order FEM schemes for hydrodynamics [11], direct-iterative preconditioned solvers [33], quantum chemistry [7], image [41], and signal processing [6]. Batch LU factorization was used in subsurface transport simulation [46], whereby many chemical and microbiological reactions in a flow path are simulated in parallel [47]. Finally, small independent problems also occur as a very important aspect of computations on hierarchical matrices (H-matrices) [25].




(a) Performance on matrices of sizes less than 32.

(b) Performance on larger-sized matrices.

Fig. 3. Performance of batch DGEMM on Titan V GPU and dual 18-core Intel Xeon Gold 6140 (Cascade Lake) 2.30 GHz CPU in NVIDIA cuBLAS and Intel MKL.

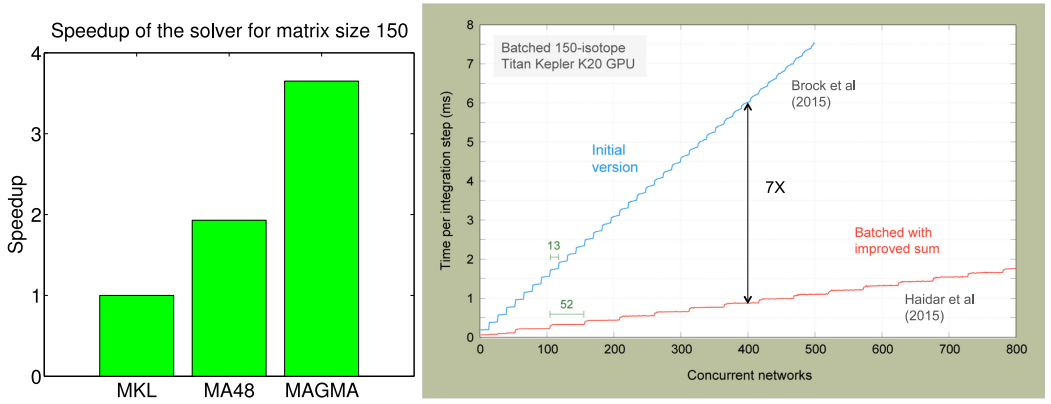



(a) Speedup for MAGMA's batched LU.

(b) Power consumption during MAGMA's batched LU.

Fig. 4. Speedup and power consumption achieved by the MAGMA's batched LU factorization on NVIDIA's Kepler K40c GPU from Tesla product line compared against 16-core Intel Xeon ES-2670 (Sandy Bridge) 2.60 GHz CPUs.

One might expect that such applications would be well suited to accelerators or coprocessors, like GPUs. Due to the high levels of parallelism that these devices support, they can efficiently achieve very high performance for large data-parallel computations when they are used in combination with a CPU that handles the part of the computation that is difficult to parallelize [3, 26, 45]. But for several reasons, this turns out not to be the case for applications involving large amounts of data that come in small units. For the case of LU, QR, and Cholesky factorizations of many small matrices, we have demonstrated that, under such circumstances, by creating software that groups these small inputs together and runs them in large "batches," we can dramatically improve performance by exploiting the increased parallelism that the grouping provides as well as the opportunities for algorithmic improvements and code optimizations [27, 28]. By using batch operations to overcome the bottleneck, small problems can be solved two to three times faster on GPUs, and with four to five times better energy efficiency than on multicore CPUs alone (subject to the same power draw). For example, Figure 1(a) illustrates this for the case of many small LU factorizations—even in a multicore setting, the batch processing approach outperforms its




(a) Speedup for the batched LU solver. (b)

Fig. 5. Acceleration of XNet and astrophysics' thermonuclear networks applications using batched approach.

non-batch counterpart by a factor of approximately two, while the batch approach in MAGMA[7] on a NVIDIA Kepler K40c GPU outperforms by about 2× the highly optimized CPU batch version running on 16 Intel Sandy Bridge cores [28]. Moreover, similarly to the way LAPACK routines benefit from BLAS, we have shown that these batch factorizations can be organized as a sequence of Batched BLAS calls, and their performance is portable across architectures, provided that the Batched BLAS needed are available and well optimized. Note that NVIDIA is already providing some optimized Batched BLAS implementations in cuBLAS [43], and Intel also included a batch matrix-matrix product (GEMM BATCH) in MKL [24]. Consequently, batch factorizations, and the underlying Batched BLAS, can be used in applications. For example, as shown in Figure 5(a), the batched LU results were used to speed up a nuclear network simulation. The XNet benchmark shows up to 3.6× improvement against the MKL library and up to 2× speedup over the MA48 factorization from the Harwell Subroutine Library [32] when solving hundreds of matrices of size $150 \times 150$ on Titan supercomputer at ORNL [12]. Another example shown in Figure 5(b) is the astrophysical thermonuclear networks coupled to hydrodynamical simulations in explosive burning scenarios [9] that was accelerated 7× by using the batch approach.

## 8 FUTURE DIRECTIONS

Defining a Batched BLAS interface is a response to the demand for acceleration of new (batch) linear algebra routines on heterogeneous and manycore architectures used in current applications. While expressing the computations in applications through matrix algebra (e.g., Level 3 BLAS) works well for large matrices, handling small matrices brings new challenges. The goal of the Batched BLAS is to address these challenges on a library level. The proposed API provides a set of routines featuring BLAS-inspired data storage and interfaces. Similarly to the use of BLAS, there are optimization opportunities for batch computing problems that cannot be folded into the Batched BLAS, and therefore must be addressed separately. For example, these are cases where operands $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$ share data, operands are not directly available in the BLAS matrix format, or where expressing a computation through BLAS may just lose application-specific knowledge about data affinity. For instances where the operands originate from multi-dimensional data, which is a common case, in future work, we will look at new interfaces and data abstractions, e.g., tensor-based, where:

[7]See MAGMA's website at https://icl.utk.edu/magma/.

(1) explicit preparation of operands can be replaced by some index operation;
(2) operands do not need to be in matrix form, but instead, can be directly loaded in matrix form in fast memory and proceed with the computation from there;
(3) expressing computations through BLAS will not lead to loss of information, e.g., that can be used to enforce certain memory affinity or other optimization techniques, because the entire data abstraction (tensor/s) will be available to the routine (and to all cores/multiprocessors/etc.) [1, 8].

Finally, we reiterate that the goal is to provide the developers of applications, compilers, and runtime systems with the option of expressing many small BLAS operations as a single call to a routine from the new batch operation standard. Thus, we hope that this standard will help and encourage community efforts to build higher-level algorithms, e.g., not only for dense problems as in LAPACK, but also for sparse problems as in preconditioners for Krylov subspace solvers, sparse direct multifrontal solvers, and so on, using Batched BLAS routines. Some optimized Batched BLAS implementations are already available in the MAGMA library, and moreover, industry leaders such as NVIDIA, Intel, and AMD have also noticed the demand and have started providing some optimized Batched BLAS implementations in their own vendor-optimized libraries.

## ACKNOWLEDGMENTS

We would like to thank the many people who have offered suggestions and attended the Batched BLAS events. In particular, we would like to thank the following for making valuable contributions to this effort: James Demmel, Iain Duff, Murat Guney, Greg Henry, Jonathan Hogg, Hatem Ltaief, Sarah Knepper, Pedro Valero Lara, Srikara Pranesh, Sivasankaran Rajamanickam, Samuel Relton, Jason Riedy, and Shane Story.

## REFERENCES

[1] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack J. Dongarra, C. Earl, J. Falcou, Azzam Haidar, Ian Karlin, Tzanio Kolev, Ian Masliah, and Stanimire Tomov. 2016. *High-performance Tensor Contractions for GPUs.* Technical Report UT-EECS-16-738. University of Tennessee Computer Science.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance, design, and autotuning of batched GEMM for GPUs. In *Proceedings of the ISC High Performance Conference.* Springer, 21–38.

[3] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. 2010. Faster, cheaper, better – A hybridization methodology to develop linear algebra software for GPUs. In *GPU Computing Gems*, Wen mei W. Hwu (Ed.). Vol. 2. Morgan Kaufmann, Burlington, MA.

[4] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Jack Langou, Haitem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Series* 180, 1 (2009). DOI: https://doi.org/10.1088/1742-6596/180/1/012037

[5] Ed Anderson, Z. Bai, C. Bischof, Susan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, Sven Hammarling, A. McKenney, and Danny C. Sorensen. 1999. *LAPACK User's Guide (3 ed.).* Society for Industrial and Applied Mathematics, Philadelphia, PA.

[6] Michael J. Anderson, David Sheffield, and Kurt Keutzer. 2012. A predictive model for solving small linear algebra problems in GPU registers. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium.* IEEE, New York, 2–13. DOI: https://doi.org/10.1109/IPDPS.2012.11

[7] Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. 2006. Automatic code generation for many-body electronic structure methods: The tensor contraction engine. *Molec. Phys.* 104, 2 (Jan. 2006), 211–228.

[8] Marc Baboulin, Veselin Dobrev, Jack J. Dongarra, C. Earl, J. Falcou, Azzam Haidar, Ian Karlin, Tzanio Kolev, Ian Masliah, and Stanimire Tomov. 2015. Towards a high-performance tensor algebra package for accelerators. In *Proceedings of the Smoky Mountains Computational Sciences and Engineering Conference (SMC'15).* HAL Archives. Retrieved from http://computing.ornl.gov/workshops/SMC15/presentations/.

[9] Benjamin Brock, Andrew Belt, Jay J. Billings, and Mike Guidry. 2015. Explicit integration with GPU acceleration for large kinetic networks. *J. Comput. Phys.* 302, C (Jan.–Dec. 2015), 591–602. arXiv:1409.5826 (2015).

[10] J. Demmel, M. Gates, G. Henry, X. S. Li, J. Riedy, and P. T. P. Tang. 2017. *A Proposal for a Next-generation BLAS.* Technical Report. Retrieved from http://goo.gl/D1UKnw.

[11] Tingxing Dong, Veselin Dobrev, Tzanio Kolev, Robert Rieben, Stanimire Tomov, and Jack Dongarra. 2014. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *Proceedings of the IEEE 28th International Parallel Distributed Processing Symposium (IPDPS'14)*. IEEE, New York, 972–981.

[12] Tingxing Dong, Azzam Haidar, Piotr Luszczek, A. Harris, Stanimire Tomov, and Jack J. Dongarra. 2014. LU factorization of small matrices: Accelerating batched DGETRF on the GPU. In *Proceedings of 16th IEEE International Conference on High Performance and Communications (HPCC'14)*. IEEE, 157–160.

[13] Jack J. Dongara, J. R. Bunch Cleve B. Moler, and G. W. Stewart. 1979. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

[14] Jack Dongarra, Iain Duff, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jonathon Hogg, Pedro Valero-Lara, Samuel D. Relton, Stanimire Tomov, and Mawussi Zounon. 2016. *A Proposed API for Batched Basic Linear Algebra Subprograms*. MIMS EPrint 2016.25. Manchester Institute for Mathematical Sciences, The University of Manchester, UK. Retrieved from http://eprints.ma.man.ac.uk/2464/.

[15] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim Yarkhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Šístek, David Stevens, Mawussi Zounon, and Samuel D. Relton. 2019. PLASMA: Parallel linear algebra software for multicore using OpenMP. *ACM Trans. Math. Softw.* 45, 2 (May 2019). DOI: https://doi.org/10.1145/3264491

[16] Jack Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. 2016. Creating a standardised set of batched BLAS routines. In *Proceedings of the 4th Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE'16)*, Gabrielle Allen, Jeffrey Carver, Sou-Cheng T. Choi et al. (Eds.), Vol. 1686. CEUR Workshop Proceedings, WSSSPE, 1–2. Retrieved from http://ceur-ws.org/Vol-1686/WSSSPE4_paper_3.pdf.

[17] Jack Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. 2017. The design and performance of batched BLAS on modern high-performance computing systems. *Proced. Comput. Sci.* 108 (2017), 495–504. DOI: http://dx.doi.org/10.1016/j.procs.2017.05.138

[18] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and Sven Hammarling. 1990. Algorithm 679: A set of Level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16 (Mar. 1990), 1–17.

[19] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and Sven Hammarling. 1990. A set of Level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16 (Mar. 1990), 18–28.

[20] Jack J. Dongarra, J. Du Croz, Sven Hammarling, and R. Hanson. 1988. Algorithm 656: An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.* 14 (Mar. 1988), 18–32.

[21] Jack J. Dongarra, J. Du Croz, Sven Hammarling, and R. Hanson. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.* 14 (Mar. 1988), 1–17.

[22] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a modern distributed and accelerated linear algebra library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'19)*. ACM, New York, NY, 13.

[23] Scott Gray. 2015. A full walk through of the SGEMM implementation. Retrieved from https://github.com/NervanaSystems/maxas/wiki/SGEMM.

[24] Murat Guney, Sarah Knepper, Kazushige Goto, Vamsi Sripathi, Greg Henry, and Shane Story. 2015. Batched Matrix-Matrix Multiplication Operations for Intel Xeon Processor and Intel Xeon Phi Co-Processor. Retrieved from http://meetings.siam.org/sess/dsp talk.cfm?p=72187.

[25] W. Hackbusch. 1999. A sparse matrix arithmetic based on H-matrices. Part I: Introduction to H-matrices. *Computing* 62, 2 (May 1999), 89–108. DOI: https://doi.org/10.1007/s006070050015

[26] Azzam Haidar, Chongxiao Cao, Asim Yarkhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, and Jack Dongarra. 2014. Unified development for mixed multi-GPU and multi-coprocessor environments using a lightweight runtime environment. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*. IEEE Computer Society, Washington, DC, 491–500. DOI: https://doi.org/10.1109/IPDPS.2014.58

[27] Azzam Haidar, TingXing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. 2015. Optimization for performance and energy for batched matrix computations on GPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU'15) co-located with PPOPP'15*. ACM, 59–69.

[28] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. 2015. Towards batched linear solvers on accelerated hardware platforms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. ACM, 261–262.

[29] Sven Hammarling. 2016. *Workshop on Batched, Reproducible, and Reduced Precision BLAS*. Technical Report. The University of Manchester, Manchester, UK. Retrieved from eprints.maths.manchester.ac.uk/2494/.

[30] Sven Hammarling. 2017. *Second Workshop on Batched, Reproducible, and Reduced Precision BLAS*. Technical Report. The University of Manchester, Manchester, UK. Retrieved from eprints.maths.manchester.ac.uk/2543/.

[31] Nicholas J. Higham and Theo Mary. 2018. *A New Approach to Probabilistic Rounding Error Analysis*. Technical Report MIMS EPrint:2018.33. Manchester Institute for Mathematical Sciences, School of Mathematics, The University of Manchester. Retrieved from eprints.maths.manchester.ac.uk/2679/.

[32] HSL. 2013. A collection of Fortran codes for large scale scientific computation. Retrieved from http://www.hsl.rl.ac.uk.

[33] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perf. Comput. Appl.* 18, 1 (Feb. 2004), 135–158. DOI: https://doi.org/10.1177/1094342004041296

[34] Khairul Kabir, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2015. On the design, development, and analysis of optimized matrix-vector multiplication routines for coprocessors. In *Proceedings of the ISC High Performance Conference*. Springer, 58–73.

[35] David Keyes and Valerie Taylor. March 2011. NSF-ACCI task force on software for science and engineering. Retrieved from https://www.nsf.gov/cise/aci/taskforces/TaskForceReport Software.pdf.

[36] J. C. Liao Khodayari A., A. R. Zomorrodi, and C. D. Maranas. 2014. A kinetic model of Escherichia coli core metabolism satisfying multiple sets of mutant flux data. *Metab. Eng.* 25C (2014), 50–62.

[37] Junjie Lai and Andre Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. IEEE Computer Society, Washington, DC, 1–10.

[38] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. 1979. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.* 5 (1979), 308–323.

[39] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, J. Falcou, and Jack J. Dongarra. 2016. *High-performance Matrix-matrix Multiplications of Very Small Matrices*. Technical Report UT-EECS-16-740. University of Tennessee Computer Science.

[40] O. E. B. Messer, J. A. Harris, S. Parete-Koon, and M. A. Chertkow. 2012. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of the State-of-the-Art in Scientific and Parallel Computing Conference (PARA'12)*. Springer-Verlag, Berlin, Germany, 92–106.

[41] J. M. Molero, E. M. Garzón, I. García, E. S. Quintana-Ortí, and A. Plaza. 2013. Poster: A Batched Cholesky Solver for Local RX Anomaly Detection on GPUs. PUMPS. Retrieved from https://dl.acm.org/doi/10.1145/2716282.2716288.

[42] Rajib Nath, Stanimire Tomov, and Jack J. Dongarra. 2010. An improved MAGMA GEMM for Fermi GPUs. *Int. J. High Perf. Comput. Appl.* 24, 4 (Nov. 2010), 511–515.

[43] NVIDIA. 2016. cuBLAS 7.5. Available at Retrieved from http://docs.nvidia.com/cuda/cublas/.

[44] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sũn. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM, New York, NY, 35:1–35:11.

[45] S. Tomov, J. Dongarra, and M. Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput. Syst. Appl.* 36, 5–6 (2010), 232–240.

[46] Oreste Villa, Massimiliano Fatica, Nitin Gawande, and Antonino Tumeo. 2013. Power/performance trade-offs of small batched LU based solvers on GPUs. In *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26–30, 2013. Proceedings*, Felix Wolf, Bernd Mohr, and Dieter an Mey (Eds.). Springer Berlin, 813–825. DOI: https://doi.org/10.1007/978-3-642-40047-6_81

[47] Oreste Villa, Nitin Gawande, and Antonino Tumeo. 2013. Accelerating subsurface transport simulation on heterogeneous clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, New York, 1–8. DOI: https://doi.org/10.1109/CLUSTER.2013.6702656

[48] Sencer Nuri Yeralan, Timothy A. Davis, Wissam M. Sid-Lakhdar, and Sanjay Ranka. 2017. Algorithm 980: Sparse QR factorization on the GPU. *ACM Trans. Math. Softw.* 44, 2 (Aug. 2017). DOI: https://doi.org/10.1145/3065870

Received October 2019; revised July 2020; accepted October 2020