NLAFET
Parallel Numerical Linear Algebra
for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

D7.3

Draft specification for Hybrid (Batched) BLAS

April 2016

Document information

Scheduled delivery	2016-04-30
Actual delivery	2016-04-26
Version	0.1
Responsible partner	UNIMAN

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
21/04/2016	Pedro Valero-Lara	Draft	0.1	-
26/04/2016	Pedro Valero-Lara	Final	-	Revision w.r.t internal reviewers comments

Author(s)

- Jack Dongarra. Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA; Oak Ridge National Laboratory, TN, USA; School of Computer Science and School of Mathematics, The University of Manchester, Manchester, UK.
- Iain Duff, STFC Rutherford Appleton Laboratory, Harwell Oxford, UK.
- Mark Gates, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA.
- Azzam Haidar, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA.
- Sven Hammarling, School of Mathematics, The University of Manchester, Manchester, UK.
- Nicholas J. Higham, School of Mathematics, The University of Manchester, Manchester, UK.
- Jonathan Hogg, STFC Rutherford Appleton Laboratory, Harwell Oxford, UK.
- Pedro Valero-Lara, School of Mathematics, The University of Manchester, Manchester, UK.
- Samuel D. Relton, School of Mathematics, The University of Manchester, Manchester, UK.
- Stanimire Tomov, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA.

- Mawussi Zounon, School of Mathematics, The University of Manchester, Manchester, UK.

Internal reviewers

Bo Kågström and Lars Karlsson, UMU.

Contributors

- The Innovative Computing Laboratory (ICL), the University of Tennessee.
- Intel Corporation.

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633. This material is based upon work supported in part by the National Science Foundation under Grants No. CSR 1514286 and ACI-1339822, NVIDIA, the Department of Energy.

Table of Contents

List of Figures

Abstract

This paper proposes an API for Batched Basic Linear Algebra Subprograms (Batched BLAS). We focus on many independent BLAS operations on small matrices that are grouped together as a single routine, called **Batched BLAS** routine, with the aim of providing more efficient, but portable, implementations of algorithms on high-performance manycore architectures (like multi/manycore CPU processors, GPUs, and coprocessors).

1 INTRODUCTION

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 [1], which indeed turned out to be advantageous and very successful. It was adopted as a standard and used in a wide range of numerical software, including LINPACK [2]. An extended, Level 2 BLAS, was proposed for matrix-vector operations [3]. Unfortunately, while successful for the vector-processing machines at the time, Level 2 BLAS was not a good fit for the cache-based machines that emerged in the 1980's. With these cache based machines, it was preferable to express computations as matrix-matrix operations. Matrices were split into small blocks so that 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 operations to data movement. Subsequently, Level 3 BLAS was proposed [4], covering the main types of matrix-matrix operations, and LINPACK was redesigned into LAPACK [5] to use the new Level 3 BLAS where possible.

The introduction of the GEMM Based Level 3 BLAS [6, 7] showed that it is possible to develop a portable and high performance Level 3 BLAS library mainly relying on a highly optimized GEMM, the routine for the general matrix multiply and add operation. In turn, this promoted the use of a recursive blocked approach [8], which has the potential to automatically adapt to a memory hierarchy.

For the emerging multicore architectures of the 2000's, the PLASMA library [9] 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. Overhead of scheduling 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.

In the 2010's, 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). Unfortunately, numerous modern applications are cast in terms of a solution of *many small matrix operations*; that is, at some point in their execution, such 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 quantum Hall effect), astrophysics [10], metabolic networks [11], CFD and resulting PDEs through direct and multifrontal solvers [12], high-order FEM schemes for hydrodynamics [13], direct-iterative preconditioned solvers [14], quantum chemistry [15], image [16], and signal processing [17].

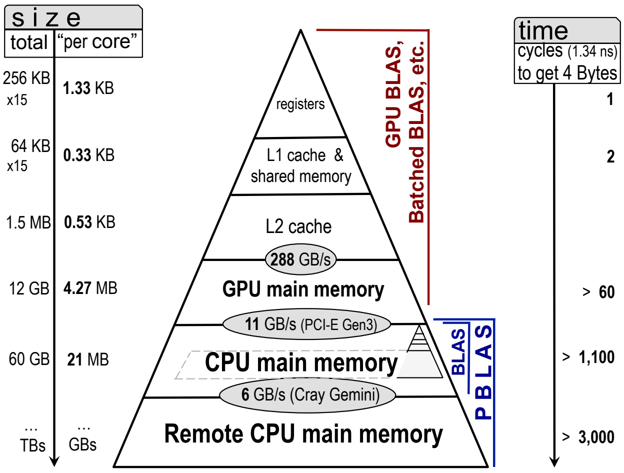


Figure 1: Memory hierarchy of a heterogeneous system from the point of view of a CUDA core of an NVIDIA K40c GPU with 2,880 CUDA cores.

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 [18, 19, 20]. But for several reasons, this turns out not to be the case for applications that involve 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 [21, 22]. By using batched 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 2, Left illustrates this for the case of many small LU factorizations – even in a multicore setting the batched approach outperforms its non-batched counterpart by a factor of approximately 2, while the batched approach in MAGMA on a K40c GPU outperforms by about 2× the highly optimized CPU batched version running on 16 Intel Sandy Bridge cores [21]. The factorizations are organized as a sequence of batched BLAS calls. Note that NVIDIA is already providing some optimized Batched BLAS implementations in CUBLAS [23], and Intel has also included a batched matrix-matrix product (GEMM_BATCH) in MKL [24]. The performance improvement over this batched version from CUBLAS is due to batched BLAS optimizations and some algorithmic improvements [21]. For example, these particular results were used to speed up a nuclear network simulation – the XNet benchmark, as shown in Figure 3(a) – up to 3.6×, *vs.* using the MKL Library, and up to 2× speedup

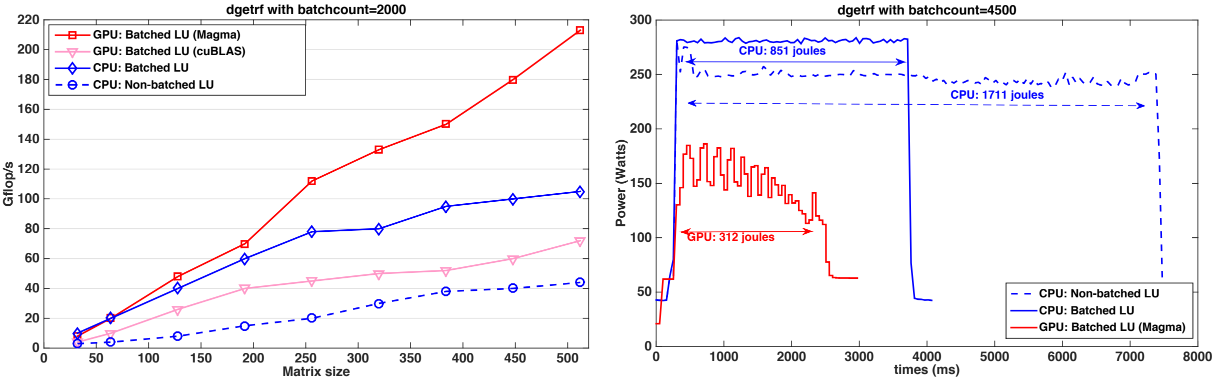
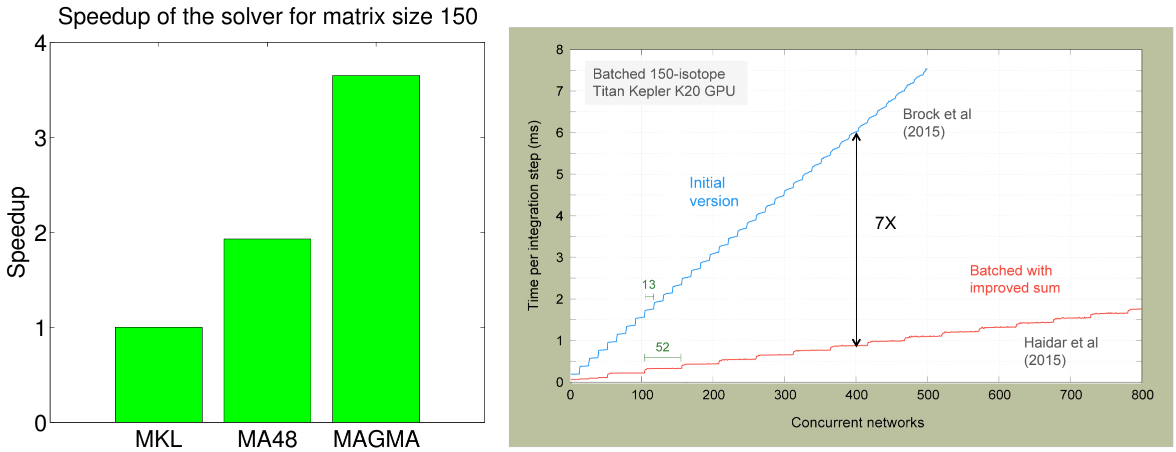


Figure 2: Speedup (Left) and power consumption (Right) achieved by the MAGMA Batched LU factorization on NVIDIA K40c GPU *vs.* 16 cores of Intel Xeon ES-2670 (Sandy Bridge) 2.60GHz CPUs.

over the MA48 factorization from the Harwell Subroutine Library [25], by solving hundreds of matrices of size 150 × 150 on the Titan supercomputer at ORNL [26]. Another example shown (in Figure 3(b)) is the astrophysical thermonuclear networks coupled to hydrodynamical simulations in explosive burning scenarios [27] that was accelerated 7× by using the batched approach.



(a) Xnet speedup

(b) Astrophysics speedup

Figure 3: Acceleration of different applications by using batched approach.

Given the fundamental importance of numerical libraries to science and engineering applications of all types [28], the need for libraries that can perform batched operations on small matrices has clearly become acute. Therefore, to fill this critical gap, we propose standard interfaces for batched BLAS operatons.

The interfaces are intentionally designed to be close to the BLAS standard and **to be hardware independent**. They are given in C for use in C/C++ programs, but extensions/implementations can be called from other languages, e.g., Fortran. 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.

2 NAMING CONVENTIONS

The name of a Batched BLAS routine follows, and extends as needed, the conventions of the corresponding BLAS routine. In particular, the name is composed of 5 characters, specifying the BLAS routine and described below, followed by the suffix _batch:

- The first character in the name denotes the data type of the matrix, as follows:
  - **s** float
  - **d** double
  - **c** complex
  - **z** double complex (if available)

- Characters two and three in the name refer to the kind of matrix involved, as follows:
  - **ge** All matrices are general rectangular
  - **he** One of the matrices is Hermitian
  - **sy** One of the matrices is symmetric
  - **tr** One of the matrices is triangular

- The fourth and fifth, and in one case sixth, characters in the name denote the operation. For example, for the Level 3 Batched BLAS, the operations are given as follows:
  - **mm** Matrix-matrix product
  - **rk** Rank-k update of a symmetric or Hermitian matrix
  - **r2k** Rank-2k update of a symmetric or Hermitian matrix
  - **sm** Solve a system of linear equations for a matrix of right-hand sides

  The Level 1 and Level 2 Batched BLAS operations follow the corresponding Level 1 and Level 2 BLAS operations.

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 batched operations. The order of arguments is as follows:

1. Array of arguments specifying options
2. Array of arguments defining the sizes of the matrices
3. Array of descriptions of the input-output matrices
4. Array of input scalars (associated with input-output matrices)
5. Integer that specifies the number of matrices in the batch
6. An enumerated value that specifies the style for the batched computation
7. Array of info parameters

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

3.1 Arguments specifying options

The arguments that specify options are of enum type with names *side*, *transa*, *transb*, *trans*, *uplo*, and *diag*. These arguments, along with the values that they can take, are described below:

o *side* has two possible values which are used by the routines as follows:

- **BatchLeft:** Specifies to multiply a general matrix by symmetric, Hermitian, or triangular matrix on the left;
- **BatchRight:** Specifies to multiply general matrix by symmetric, Hermitian, or triangular matrix on the right.

o *transa*, *transb*, and *trans* can have three possible values each, which is used to specify the following:

- **BatchNoTrans:** Operate with the matrix as it is;
- **BatchTrans:** Operate with the transpose of the matrix;
- **BatchConjTrans:** Operate with the conjugate transpose of the matrix.

Note that in the real case, the values '**BatchTrans**' and '**BatchConjTrans**' have the same meaning.

o *uplo* is used by the Hermitian, symmetric, and triangular matrix routines to specify whether the upper or lower triangle is being referenced, as follows:

- **BatchLower:** Lower triangle;
- **BatchUpper:** Upper triangle.

o *diag* is used by the triangular matrix routines to specify whether the matrix is unit triangular, as follows:

- **BatchUnit:** Unit triangular;
- **BatchNonUnit:** Nonunit triangular.

When *diag* is supplied as '**BatchUnit**', the diagonal elements are not referenced.

3.2 Arguments defining the sizes

The sizes of matrices A_i, B_i, and C_i for the i^{th} BLAS operation are determined by the corresponding values of the arrays m, n, and k at position i (see the routine interfaces in Section 4). It is permissible to call the routines with m and/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 and $n > 0$, but $k = 0$, 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 $m \times n$ if working with rectangular A, and $n \times n$ if A is a square matrix. If the *batch_opts* argument specifies Batched BLAS operations on matrices of the same sizes (see Section 3.6), the m, n, and k values for all matrices are specified by the $m[0]$, $n[0]$, and $k[0]$ values, respectively.

3.3 Arguments describing the input-output matrices

The description of the matrix consists of the array name (*arrayA*, *arrayB*, or *arrayC*) followed by an array of the leading dimension as declared in the calling function (*lda*, *ldb*, or *ldc*). The i^{th} values of the *arrayA*, *arrayB*, and *arrayC* are pointers to the arrays of data A_i, B_i, and C_i, respectively. Similarly, the values of $lda[i]$, $ldb[i]$, and $ldc[i]$ correspond to the leading dimensions of the matrices A_i, B_i, and C_i, respectively. For batch style with the same leading dimensions (see Section 3.6), the leading dimensions are specified by $lda[0]$, $ldb[0]$, and $ldc[0]$ for all corresponding $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$ matrices.

3.4 Arguments defining the input scalar

Arrays of scalars are named *alpha* and *beta*, where values at position i correspond to the α and β scalars for the BLAS operation involving matrices A_i, B_i, and C_i. For batch style with the same scalars (see Section 3.6), the scalars are given by $alpha[0]$ and $beta[0]$.

3.5 Specification of the number of matrices

The *batch_count* argument is an integer that indicates the number of matrices to be processed.

3.6 Batch style specifications

The *batch_opts* argument is an enumerated value that specifies the style for the batched computation. Permitted values are either **BATCH_FIXED** or **BATCH_VARIABLE**, which stand for computation of matrices with same or variable sizes (including operation options, sizes, matrix leading dimensions, and scalars), respectively.

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

3.7 Error handling defined by the INFO array

The following values of arguments are invalid:

- o Any value of the character arguments *side*, *transa*, *transb*, *trans*, *uplo*, or *diag* whose meaning is not specified;

- o If any of m, n, k, lda, ldb, or ldc is less than zero.

If a routine is called with an invalid value for *batch_count*, the routine will return an error in $info[0]$ that refers to the number of the *batch_count* argument (counting from one). Otherwise, if a routine is called with an invalid value for any of its other arguments for a BLAS operation at position i, the routine will return an error in $info[i]$ that refers to the number of the first invalid argument (counting from one).

4 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. In real arithmetic, the operations proposed for the Level 3 Batched BLAS have an interface described as follows.

4.1 Matrix-matrix products {s,d,c,z}gemm_batch

This routine performs a batch of one of the matrix-matrix operations described below:

$$
\begin{aligned}
&C_{m\times n} = \alpha \cdot A_{m\times k} \times B_{k\times n} + \beta \cdot C_{m\times n},\\
&C_{m\times n} = \alpha \cdot A^T_{k\times m} \times B_{k\times n} + \beta \cdot C_{m\times n},\\
&C_{m\times n} = \alpha \cdot A^H_{k\times m} \times B_{k\times n} + \beta \cdot C_{m\times n},\\
&C_{m\times n} = \alpha \cdot A_{m\times k} \times B^T_{n\times k} + \beta \cdot C_{m\times n},\\
&C_{m\times n} = \alpha \cdot A_{m\times k} \times B^H_{n\times k} + \beta \cdot C_{m\times n},\\
&C_{m\times n} = \alpha \cdot A^T_{k\times m} \times B^T_{n\times k} + \beta \cdot C_{m\times n},\\
&C_{m\times n} = \alpha \cdot A^H_{k\times m} \times B^H_{n\times k} + \beta \cdot C_{m\times n}.
\end{aligned}
$$

The calling routine is described as follows:

$$
\begin{aligned}
\{s,d,c,z\}gemm\_batch(&enum & *transa,\\
&enum & *transb,\\
&integer & *m,\\
&integer & *n,\\
&integer & *k,\\
&\{s,d,c,z\}precision & *alpha,\\
&\{s,d,c,z\}precision & **arrayA,\\
&integer & *lda,\\
&\{s,d,c,z\}precision & **arrayB,\\
&integer & *ldb,\\
&\{s,d,c,z\}precision & *beta,\\
&\{s,d,c,z\}precision & **arrayC,\\
&integer & *ldc,\\
&integer & batch\_count,\\
&enum & batch\_opts,\\
&integer & *info)
\end{aligned}
$$

where {s,d,c,z}precision denotes one of the four standard floating-point arithmetic precisions (float, double, complex, or double complex). The *transa* and *transb* arrays can be of size one for the same size batch and of size at least *batch_count* for the variable sizes case. For the latter, each value defines the operation on the corresponding matrix. In the real precision case, the values **BatchTrans** and **BatchConjTrans** have the same meaning. The m, n, and k arrays of integers are of size at least batch_count, where each value defines the dimension of the operation on each corresponding matrix. The *alpha* and *beta* arrays provide the scalars α and β, described in the equation above. They are of the same precision as the arrays A, B, and C. The arrays of pointers *arrayA*, *arrayB*, and *arrayC* are of size at least *batch_count* and point to the matrices $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$. The size of matrix C_i is $m[i] \times n[i]$. The sizes of the matrices A_i and B_i depend on $transa[i]$ and $transb[i]$; their corresponding sizes are mentioned in the equation above. The arrays of leading dimensions *lda*, *ldb*, and *ldc* define the leading dimension of each of the matrices $\{A_i(lda, *)\}$, $\{B_i(ldb, *)\}$, and $\{C_i(ldc, *)\}$, respectively.

If the *batch_opts* argument specifies that the batch is of `BATCH_FIXED` type, only $transa[0]$, $transb[0]$, $m[0]$, $n[0]$, $k[0]$, $alpha[0]$, $lda[0]$, $ldb[0]$, $beta[0]$, and $ldc[0]$ are used to specify the `gemm` parameters for the batch.

The array $info$ defines the error array. It is an output array of integers of size *batch_count* where a value at position i reflects the argument error for the `gemm` with matrices A_i, B_i, and C_i.

4.2 Hermitian matrix-matrix products {c,z}hemm_batch, {s,d,c,z}symm_batch

This routine performs a batch of matrix-matrix products, each expressed in one of the following forms:

$$\begin{array}{lll} C_{m\times n} & = \alpha \cdot A_{m\times m} \times B_{m\times n} + \beta \cdot C_{m\times n} & \text{if } side = \text{BatchLeft} \\ C_{m\times n} & = \alpha \cdot B_{m\times n} \times A_{n\times n} + \beta \cdot C_{m\times n} & \text{if } side = \text{BatchRight,} \end{array}$$

where the matrices A, B, and C are real symmetric ({s,d}symm_batch), complex symmetric ({c,z}symm_batch), or complex Hermitian ({c,z}hemm_batch), and α and β are scalars.

The calling routine is described as follows:

$$\begin{array}{llr} \{s,d,c,z\}symm\_batch(& enum & *side, \\ & enum & *uplo, \\ & integer & *m, \\ & integer & *n, \\ & \{s,d,c,z\}precision & *alpha, \\ & \{s,d,c,z\}precision & **arrayA, \\ & integer & *lda, \\ & \{s,d,c,z\}precision & **arrayB, \\ & integer & *ldb, \\ & \{s,d,c,z\}precision & *beta, \\ & \{s,d,c,z\}precision & **arrayC, \\ & integer & *ldc, \\ & integer & batch\_count, \\ & enum & batch\_opts, \\ & integer & *info) \end{array}$$

The *side* array is of size at least *batch_count* and each value defines the operation on each matrix as described in the equations above. The *uplo* array is of size at least *batch_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 at least *batch_count* and define the dimension of the operation on each matrix. The *alpha* and *beta* arrays provide the scalars α and β described in the equation above. They are of the same precision as the arrays A, B, and C. The arrays *arrayA*, *arrayB*, and *arrayC* are the arrays of pointers of size *batch_count* that point to the matrices $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$. The size of matrix C_i is $m[i] \times n[i]$. The sizes of the matrices A_i and B_i depend on $side[i]$; their corresponding sizes are mentioned in the equations above. The arrays of leading dimensions *lda*, *ldb*, and *ldc* define the leading dimension of each of the matrices $\{A_i(lda,*)\}$, $\{B_i(ldb,*)\}$, and $\{C_i(ldc,*)\}$, respectively.

If the *batch_opts* argument specifies that the batch is of `BATCH_FIXED` type, only $side[0]$, $uplo[0]$, $m[0]$, $n[0]$, $alpha[0]$, $lda[0]$, $ldb[0]$, $beta[0]$, and $ldc[0]$ are used to specify the `symm` parameters for the batch.

The array *info* defines the error array. It is an output array of integers of size *batch_count* where a value at position i reflects the argument error for the `symm` with matrices A_i, B_i, and C_i.

4.3 Rank-k updates of a symmetric/Hermitian matrix {c,z}syrk_batch, {s,d,c,z}herk_batch

This routine performs a batch of rank-k updates of real symmetric ({s,d}syrk_batch), complex symmetric ({c,z}syrk_batch), or complex Hermitian ({c,z}herk_batch) matrices in the form:

$$\begin{array}{lll}
C_{n\times n} & = \alpha \cdot A_{n\times k} \times A^T_{n\times k} + \beta \cdot C_{n\times n} & \text{for syrk if } trans = \text{BatchNoTrans} \\
C_{n\times n} & = \alpha \cdot A^T_{k\times n} \times A_{k\times n} + \beta \cdot C_{n\times n} & \text{for syrk if } trans = \text{BatchTrans} \\
C_{n\times n} & = \alpha \cdot A_{n\times k} \times A^H_{n\times k} + \beta \cdot C_{n\times n} & \text{for herk if } trans = \text{BatchNoTrans} \\
C_{n\times n} & = \alpha \cdot A^H_{k\times n} \times A_{k\times n} + \beta \cdot C_{n\times n} & \text{for herk if } trans = \text{BatchConjTrans}
\end{array}$$

The calling routine is described as follows:

$$\begin{array}{lll}
\{s,d,c,z\}syrk\_batch(& enum & *uplo, \\
 & enum & *trans, \\
 & integer & *n, \\
 & integer & *k, \\
 & \{s,d,c,z\}precision & *alpha, \\
 & \{s,d,c,z\}precision & **arrayA, \\
 & integer & *lda, \\
 & \{s,d,c,z\}precision & *beta, \\
 & \{s,d,c,z\}precision & **arrayC, \\
 & integer & *ldc, \\
 & integer & batch\_count, \\
 & enum & batch\_opts, \\
 & integer & *info)
\end{array}$$

The *uplo* array is of size at least *batch_count* and defines whether the upper or the lower triangular part of the matrix is to be referenced. The *trans* array is of size at least *batch_count* where each value defines the operation on each matrix. In the real precision case, the values '**BatchTrans**' and '**BatchConjTrans**' have the same meaning. In the complex case, *trans* = **BatchConjTrans** is not allowed in xsyrk_batch. The n and k arrays of integers are of size at least *batch_count* and define the dimensions of the operation on each matrix. The *alpha* and *beta* arrays provide the scalars α and β described in the equation above. They are of the same precision as the arrays A and C. The arrays of pointers *arrayA* and *arrayC* are of size *batch_count* and point to the matrices $\{A_i\}$ and $\{C_i\}$. The size of matrix C_i is $n[i] \times n[i]$. All matrices $\{C_i\}$ are either real or complex symmetric. The size of the matrix A_i depends on $trans[i]$; its corresponding size is mentioned in the equation above. The arrays of leading dimensions *lda* and *ldc* define the leading dimension of each of the matrices $\{A_i(lda,*)\}$ and $\{C_i(ldc,*)\}$, respectively.

If the *batch_opts* field specifies that the batch is of `BATCH_FIXED` type, only $uplo[0]$, $trans[0]$, $n[0]$, $k[0]$, $alpha[0]$, $lda[0]$, $beta[0]$, and $ldc[0]$ are used to specify the `syrk` parameters for the batch.

The array *info* defines the error array. It is an output array of integers of size *batch_count* where a value at position i reflects the argument error for the `syrk` with matrices A_i and C_i.

$$\begin{array}{lll}
\{c,z\}herk\_batch(& enum & *uplo, \\
 & enum & *trans, \\
 & integer & *n, \\
 & integer & *k, \\
 & \{s,d\}precision & *alpha, \\
 & \{c,z\}precision & **arrayA, \\
 & integer & *lda, \\
 & \{s,d\}precision & *beta, \\
 & \{c,z\}precision & **arrayC, \\
 & integer & *ldc, \\
 & integer & batch\_count, \\
 & enum & batch\_opts, \\
 & integer & *info)
\end{array}$$

The {c,z}herk_batch routine is only available for the complex precision. It has the same parameters as the {s,d,c,z}syrk_batch except that the *trans* = **BatchTrans** is not allowed in xherk_batch and that *alpha* and *beta* are real. The matrices $\{C_i\}$ are complex Hermitian.

If the *batch_opts* argument specifies that the batch is of `BATCH_FIXED` type, only $uplo[0]$, $trans[0]$, $n[0]$, $k[0]$, $alpha[0]$, $lda[0]$, $beta[0]$, and $ldc[0]$ are used to specify the `herk` parameters for the batch.

The array *info* defines the error array. It is an output array of integers of size *batch_count* where a value at position i reflects the argument error for the `herk` with matrices A_i and C_i.

4.4 Rank-2k updates of a symmetric/Hermitian matrix {s,d,c,z}syr2k_batch, {c,z}her2k_batch

This routine performs batched rank-2k updates on real symmetric ({s,d}syr2k_batch), complex symmetric ({c,z}syr2k_batch), or complex Hermitian ({c,z}her2k_batch) matrices of the form:

$$\begin{array}{lll}
C_{n\times n} & = \alpha \cdot A_{n\times k} \times B^T_{n\times k} + \alpha \cdot B_{n\times k} \times A^T_{n\times k} + \beta \cdot C_{n\times n} & \text{for syr2k if } trans = \text{BatchNoTrans} \\
C_{n\times n} & = \alpha \cdot A^T_{k\times n} \times B_{k\times n} + \alpha \cdot B^T_{k\times n} \times A_{k\times n} + \beta \cdot C_{n\times n} & \text{for syr2k if } trans = \text{BatchTrans} \\
C_{n\times n} & = \alpha \cdot A_{n\times k} \times B^H_{n\times k} + \bar{\alpha} \cdot B_{n\times k} \times A^H_{n\times k} + \beta \cdot C_{n\times n} & \text{for her2k if } trans = \text{BatchNoTrans} \\
C_{n\times n} & = \alpha \cdot A^H_{k\times n} \times B_{k\times n} + \bar{\alpha} \cdot B^H_{k\times n} \times A_{k\times n} + \beta \cdot C_{n\times n} & \text{for her2k if } trans = \text{BatchConjTrans}
\end{array}$$

The calling routine is described as follows:

$$\begin{array}{llr}
\{s,d,c,z\}syr2k\_batch(& enum & *uplo, \\
 & enum & *trans, \\
 & integer & *n, \\
 & integer & *k, \\
 & \{s,d,c,z\}precision & *alpha, \\
 & \{s,d,c,z\}precision & **arrayA, \\
 & integer & *lda, \\
 & \{s,d,c,z\}precision & **arrayB, \\
 & integer & *ldb, \\
 & \{s,d,c,z\}precision & *beta, \\
 & \{s,d,c,z\}precision & **arrayC, \\
 & integer & *ldc, \\
 & integer & batch\_count, \\
 & enum & batch\_opts, \\
 & integer & *info)
\end{array}$$

The *uplo* array is of size *batch_count* and defines whether the upper or the lower triangular part of the matrix is to be referenced. The *trans* array is of size *batch_count* where each value defines the operation on each matrix. In the real precision case, the values '**BatchTrans**' and '**BatchConjTrans**' have the same meaning. In the complex case, *trans* = **BatchConjTrans** is not allowed in xsyr2k_batch. The n and k arrays of integers are of size *batch_count* and define the dimensions of the operation on each matrix. The *alpha* and *beta* arrays provide the scalars α and β described in the equations above. They are of the same precision as the arrays A, B, and C. The arrays *arrayA*, *arrayB*, and *arrayC* are the arrays of pointers of size *batch_count* that point to the matrices $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$. The size of matrix C_i is $n[i] \times n[i]$. All matrices $\{C_i\}$ are either real or complex symmetric. The size of the matrices A_i and B_i depends on $trans[i]$; its corresponding size is mentioned in the equation above. The arrays of leading dimensions *lda*, *ldb*, and *ldc* define the leading dimension of the matrices $\{A_i(lda,*)\}$, $\{B_i(ldb,*)\}$, and $\{C_i(ldc,*)\}$, respectively.

If the *batch_opts* argument specifies that the batch is of `BATCH_FIXED` type, only $uplo[0]$, $trans[0]$, $n[0]$, $k[0]$, $alpha[0]$, $lda[0]$, $ldb[0]$, $beta[0]$, and $ldc[0]$ are used to specify the `syr2k` parameters for the batch.

The array $info$ defines the error array. It is an output array of integers of size $batch\_count$ where a value at position i reflects the argument error for the `syr2k` with matrices A_i, B_i, and C_i.

$$
\begin{array}{lll}
\{c,z\}her2k\_batch(& enum & *uplo, \\
 & enum & *trans, \\
 & integer & *n, \\
 & integer & *k, \\
 & \{c,z\}precision & *alpha, \\
 & \{c,z\}precision & **arrayA, \\
 & integer & *lda, \\
 & \{c,z\}precision & **arrayB, \\
 & integer & *ldb, \\
 & \{s,d\}precision & *beta, \\
 & \{c,z\}precision & **arrayC, \\
 & integer & *ldc, \\
 & integer & batch\_count, \\
 & enum & batch\_opts, \\
 & integer & *info)
\end{array}
$$

The {c,z}her2k_batch routine is only available for the complex precision. It has the same parameters as the {s,d,c,z}syr2k_batch routine except that the $trans$ = **BatchTrans** is not allowed in xher2k_batch and that $beta$ is real. The matrices $\{C_i\}$ are complex Hermitian.

If the $batch\_opts$ argument specifies that the batch is of `BATCH_FIXED` type, only $uplo[0]$, $trans[0]$, $n[0]$, $k[0]$, $alpha[0]$, $lda[0]$, $ldb[0]$, $beta[0]$, and $ldc[0]$ are used to specify the `her2k` parameters for the batch.

The array $info$ defines the error array. It is an output array of integers of size $batch\_count$ where a value at position i reflects the argument error for the `her2k` with matrices A_i, B_i, and C_i.

4.5 Multiplying a matrix by a triangular matrix {s,d,c,z}trmm_batch

This routine performs a batch of one of the following matrix-matrix products, where the matrix A is an upper or lower triangular matrix, and α is scalar:

$$
\begin{array}{lll}
B_{m\times n} & = \alpha \cdot A_{m\times m} \times B_{m\times n} & \text{if } side = \text{BatchLeft and } trans = \text{BatchNoTrans} \\
B_{m\times n} & = \alpha \cdot A^T_{m\times m} \times B_{m\times n} & \text{if } side = \text{BatchLeft and } trans = \text{BatchTrans} \\
B_{m\times n} & = \alpha \cdot A^H_{m\times m} \times B_{m\times n} & \text{if } side = \text{BatchLeft and } trans = \text{BatchConjTrans} \\
B_{m\times n} & = \alpha \cdot B_{m\times n} \times A_{m\times m} & \text{if } side = \text{BatchRight and } trans = \text{BatchNoTrans} \\
B_{m\times n} & = \alpha \cdot B_{m\times n} \times A^T_{m\times m} & \text{if } side = \text{BatchRight and } trans = \text{BatchTrans} \\
B_{m\times n} & = \alpha \cdot B_{m\times n} \times A^H_{m\times m} & \text{if } side = \text{BatchRight and } trans = \text{BatchConjTrans}
\end{array}
$$

$$\begin{array}{lll}
\{s,d,c,z\}trmm\_batch(& enum & *side, \\
 & enum & *uplo, \\
 & enum & *trans, \\
 & enum & *diag, \\
 & integer & *m, \\
 & integer & *n, \\
 & \{s,d,c,z\}precision & *alpha, \\
 & \{s,d,c,z\}precision & **arrayA, \\
 & integer & *lda, \\
 & \{s,d,c,z\}precision & **arrayB, \\
 & integer & *ldb, \\
 & integer & batch\_count, \\
 & enum & batch\_opts, \\
 & integer & *info)
\end{array}$$

The *side* array is of size *batch_count* and each value defines the operation on each matrix as described in the equations above. The *uplo* array is of size *batch_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 *batch_count* where each value defines the operation on each matrix. In the real precision case, the values '**BatchTrans**' and '**BatchConjTrans**' have the same meaning. The *diag* array is of size *batch_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 *batch_count* and define the dimension of the operation on each matrix. The *alpha* array provides the scalars α described in the equation above. It is of the same precision as the arrays A and B. The arrays of pointer *arrayA* and *arrayB* are of size *batch_count* and point to the matrices $\{A_i\}$ and $\{B_i\}$. The size of matrix B_i is $m[i] \times n[i]$. The size of matrix A_i depends on $side[i]$; its corresponding size is mentioned in the equation above. The arrays of leading dimensions lda and ldb define the leading dimension of the $\{A_i(lda,*)\}$ and $\{B_i(ldb,*)\}$ matrices, respectively.

If the *batch_opts* argument specifies that the batch is of `BATCH_FIXED` type, only $side[0]$, $uplo[0]$, $trans[0]$, $diag[0]$, $m[0]$, $n[0]$, $alpha[0]$, $lda[0]$, and $ldb[0]$ are used to specify the `trmm` parameters for the batch.

The array $info$ defines the error array. It is an output array of integers of size *batch_count* where a value at position i reflects the argument error for the `trmm` with matrices A_i and B_i.

4.6 Solving triangular systems of equations with multiple right-hand {s,d,c,z}trsm_batch

This routine solves a batch of matrix equations. Each equation is described below, where the matrix A is an upper or lower triangular matrix, and α is scalar:

$$
\begin{array}{lll}
B_{m\times n} & = \alpha \cdot A^{-1}_{m\times m} \times B_{m\times n} & \text{if } side = \text{BatchLeft and } trans = \text{BatchNoTrans} \\
B_{m\times n} & = \alpha \cdot A^{-T}_{m\times m} \times B_{m\times n} & \text{if } side = \text{BatchLeft and } trans = \text{BatchTrans} \\
B_{m\times n} & = \alpha \cdot A^{-H}_{m\times m} \times B_{m\times n} & \text{if } side = \text{BatchLeft and } trans = \text{BatchConjTrans} \\
B_{m\times n} & = \alpha \cdot B_{m\times n} \times A^{-1}_{m\times m} & \text{if } side = \text{BatchRight and } trans = \text{BatchNoTrans} \\
B_{m\times n} & = \alpha \cdot B_{m\times n} \times A^{-T}_{m\times m} & \text{if } side = \text{BatchRight and } trans = \text{BatchTrans} \\
B_{m\times n} & = \alpha \cdot B_{m\times n} \times A^{-H}_{m\times m} & \text{if } side = \text{BatchRight and } trans = \text{BatchConjTrans}
\end{array}
$$

$$
\begin{array}{llr}
\{s,d,c,z\}trsm\_batch(& enum & *side, \\
 & enum & *uplo, \\
 & enum & *trans, \\
 & enum & *diag, \\
 & integer & *m, \\
 & integer & *n, \\
 & \{s,d,c,z\}precision & *alpha, \\
 & \{s,d,c,z\}precision & **arrayA, \\
 & integer & *lda, \\
 & \{s,d,c,z\}precision & **arrayB, \\
 & integer & *ldb, \\
 & integer & batch\_count, \\
 & enum & batch\_opts, \\
 & integer & *info)
\end{array}
$$

The *side* array is of size *batch_count* where each value defines the operation on each matrix as described in the equation above. The *uplo* array is of size *batch_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 *batch_count* where each value defines the operation on each matrix. In the real precision case, the values '**BatchTrans**' and '**BatchConjTrans**' have the same meaning. The *diag* array is of size *batch_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 *batch_count* and define the dimension of the operation on each matrix. The *alpha* array provides the scalars α described in the equation above. It is of the same precision as the arrays A and B. The arrays of pointers *arrayA* and *arrayB* are of size *batch_count* and point to the matrices $\{A_i\}$ and $\{B_i\}$. The size of matrix B_i is $m[i] \times n[i]$. The size of the matrix A_i depends on $side[i]$; its corresponding size is mentioned in the equation above. The arrays of leading dimension lda and ldb define the leading dimension of the matrices $\{A_i(lda, *)\}$ and $\{B_i(ldb, *)\}$, respectively.

If the *batch_opts* argument specifies that the batch is of `BATCH_FIXED` type, only $side[0]$, $uplo[0]$, $trans[0]$, $diag[0]$, $m[0]$, $n[0]$, $alpha[0]$, $lda[0]$, and $ldb[0]$ are used to specify the `trmm` parameters for the batch.

The array $info$ defines the error array. It is an output array of integers of size *batch_count* where a value at position i reflects the argument error for the `trsm` with matrices A_i and B_i.

5 SCOPE and SPECIFICATIONS OF THE LEVEL 1 and LEVEL 2 BATCHED BLAS

Similarly to the derivation of a Level 3 Batched BLAS form the Level 3 BLAS, we derive Level 1 and Level 2 Batched BLAS from the corresponding Level 1 and Level 2 BLAS routines. Examples are given below for the Level 1 $y = \alpha x + y$ (`axpy`) and the Level 2 $y = \alpha Ax + \beta y$ (`gemv`) BLAS routines.

$$
\begin{array}{lll}
\{s,d,c,z\}axpy\_batch(& integer & *n, \\
 & \{s,d,c,z\}precision & *alpha, \\
 & \{s,d,c,z\}precision & **x, \\
 & integer & *incx, \\
 & \{s,d,c,z\}precision & **y, \\
 & integer & *incy, \\
 & integer & batch\_count, \\
 & enum & batch\_opts, \\
 & integer & *info).
\end{array}
$$

$$
\begin{array}{lll}
\{s,d,c,z\}gemv\_batch(& enum & *trans, \\
 & integer & *m, \\
 & integer & *n, \\
 & \{s,d,c,z\}precision & *alpha, \\
 & \{s,d,c,z\}precision & **arrayA, \\
 & integer & *lda, \\
 & \{s,d,c,z\}precision & **x, \\
 & integer & *incx, \\
 & \{s,d,c,z\}precision & *beta, \\
 & \{s,d,c,z\}precision & **y, \\
 & integer & *incy, \\
 & integer & batch\_count, \\
 & enum & batch\_opts, \\
 & integer & *info).
\end{array}
$$

Here $incx[i]$ and $incy[i]$ from the i^{th} BLAS operation must not be zero and specify the increments for the elements of $x[i]$ and $y[i]$, respectively.

6 BATCHED LAPACK

The batched approach to BLAS can be applied to higher-level libraries, and in particular to LAPACK. In this extension, the Batched LAPACK routines are derived from the interfaces of their corresponding LAPACK routines, similarly to the derivation of Level 3

Batched BLAS from BLAS. For example, the specification for the batched LU factorizations (using partial pivoting with row interchanges) of general M-by-N matrices specified through arrayA, is derived from the LAPACK's `getrf` routine as follows:

$$\begin{array}{lll}
\{s,d,c,z\}getrf\_batch(& integer & *m, \\
 & integer & *n, \\
 & \{s,d,c,z\}precision & **arrayA, \\
 & integer & *lda, \\
 & integer & **ipiv, \\
 & integer & batch\_count, \\
 & enum & batch\_opts, \\
 & integer & *info).
\end{array}$$

7 FUTURE DIRECTIONS AND FINAL REMARKS

Defining a Batched BLAS interface is a response to the demand for acceleration of new (batched) linear algebra routines on heterogeneous and manycore architectures used in current applications. While flattening the computations in applications to linear algebra on matrices (e.g., Level 3 BLAS) works well for large matrices, handling small matrices brings new challenges. The batched approach works, but there are cases, where for example, operands $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$ share data, operands are not directly available in the BLAS matrix format, or where the flattening may just lose application-specific knowledge about data affinity, etc., in which cases there would be overheads that could possibly be avoided. For instances where the operands originate from multi-dimensional data, which is a common case, we are looking at new interfaces and data abstractions, e.g., tensor-based, where **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)** flattening 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.) [29, 30].

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, etc., using Batched BLAS routines. Some optimized Batched BLAS implementations are already available in the MAGMA library, and moreover, industry leaders like NVIDIA, Intel, and AMD, have also noticed the demand and have started providing some optimized Batched BLAS implementations in their own vendor-optimized libraries.

References

[1] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[2] J. J. Dongara, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, PA, 1979.

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

[4] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[5] Edward Anderson, Zhaojun Bai, Christian Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven J. Hammarling, Alan McKenney, and Danny C. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[6] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. *ACM Trans. Math. Software*, 24(3):268–302, 1998.

[7] B. Kågström, P. Ling, and C. Van Loan. Algorithm 784: GEMM-Based Level 3 BLAS: Portability and Optimization Issues. *ACM Trans. Math. Software*, 24(3):303–316, 1998.

[8] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, 46(1):3–45, 2004.

[9] Emmanuel Agullo, James Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

[10] O.E.B. Messer, J.A. Harris, S. Parete-Koon, and M.A. Chertkow. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.

[11] J.C. Liao Khodayari A., A.R. Zomorrodi and C.D. Maranas. A kinetic model of escherichia coli core metabolism satisfying multiple sets of mutant flux data. *Metabolic engineering*, 25C:50–62, 2014.

[12] Sencer N. Yeralan, Tim A. Davis, and Sanjay Ranka. Sparse mulitfrontal QR on the GPU. Technical report, University of Florida Technical Report, 2013.

[13] Tingxing Dong, Veselin Dobrev, Tzanio Kolev, Robert Rieben, Stanimire Tomov, and Jack Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

[14] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, February 2004.

[15] 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 Luc, Marcel Nooijene, Russell Pitzerf, J Ramanujamg, P Sadayappanc, and Alexander Sibiryakovc. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[16] J.M. Molero, E.M. Garzón, I. García, E.S. Quintana-Ortí, and A. Plaza. Poster: A batched Cholesky solver for local RX anomaly detection on GPUs, 2013. PUMPS.

[17] M.J. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, 2012.

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

[19] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

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

[21] Azzam Haidar, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Towards batched linear solvers on accelerated hardware platforms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, 02/2015 2015. ACM, ACM.

[22] Azzam Haidar, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Optimization for performance and energy for batched matrix computations on GPUs. In *8th Workshop on General Purpose Processing Using GPUs (GPGPU 8) co-located with PPOPP 2015*, PPoPP 2015, San Francisco, CA, 02/2015 2015. ACM, ACM.

[23] CUBLAS 7.5, 2016. Available at `http://docs.nvidia.com/cuda/cublas/`.

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

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

[26] T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, and J. Dongarra. LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In *Proceedings of 16th IEEE International Conference on High Performance and Communications (HPCC 2014)*, August 2014.

[27] Benjamin Brock, Andrew Belt, Jay Jay Billings, and Mike Guidry. Explicit Integration with GPU Acceleration for Large Kinetic Networks. *J. Comput. Phys.*, 302(C):591–602, December 2015.

[28] David Keyes and Valerie Taylor. NSF-ACCI task force on software for science and engineering. https://www.nsf.gov/cise/aci/taskforces/TaskForceReport_Software.pdf, March 2011.

[29] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. High-Performance Tensor Contractions for GPUs. *University of Tennessee Computer Science Technical Report*, (UT-EECS-16-738), 01-2016 2016.

[30] M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. Towards a high-performance tensor algebra package for accelerators. http://computing.ornl.gov/workshops/SMC15/presentations/, 2015. Smoky Mountains Computational Sciences and Engineering Conference (SMC'15), Gatlinburg, TN, Sep 2015.