NLAFET
Parallel Numerical Linear Algebra for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

D7.6

Batched BLAS Specification

July 2018

Document information

Scheduled delivery	2018-07-31
Actual delivery	(not yet delivered)
Version	1.1
Responsible partner	UNIMAN

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2018-07-25	Sven Hammarling	Final version	1.1	Incorporation of reviewer's comments.
2018-07-01	Sven Hammarling	Draft	0.1	Initial version of document produced.

Authors

Jack Dongarra (UNIMAN)
Sven Hammarling (UNIMAN)
Nicholas J. Higham (UNIMAN)
Mawussi Zounon (UNIMAN)
Iain Duff (STFC)

Internal reviewers

Carl Christian Kjelgaard Mikkelsen (UMU)
Iain Duff (STFC)

External Contributors

Mark Gates (ICL)
Azzam Haidar (ICL)
Piotr Luszczek (ICL)
Stanimire Tomov (ICL)
Jonathan Hogg (Apple)
Pedro Valero Lara (BSC)
Samuel D. Relton (Uni. of Leeds)
Timothy Costa (Intel)
Sarah Knepper (Intel)

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

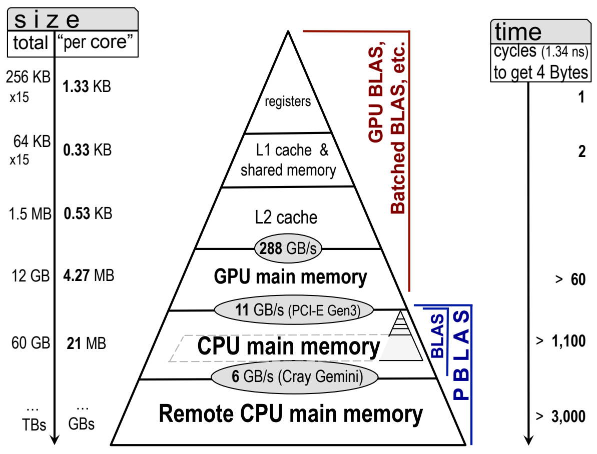


Figure 1: Memory hierarchy of a heterogeneous system from the point of view of a CUDA core of an NVIDIA K40c GPU with 2,880 CUDA cores.

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, 19] and matrix-matrix [17, 18] 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 k 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. It was adopted as a standard and used in a wide range of numerical software, including LINPACK [13]. An extended, Level 2 BLAS, was proposed for matrix-vector operations [20, 19]. 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

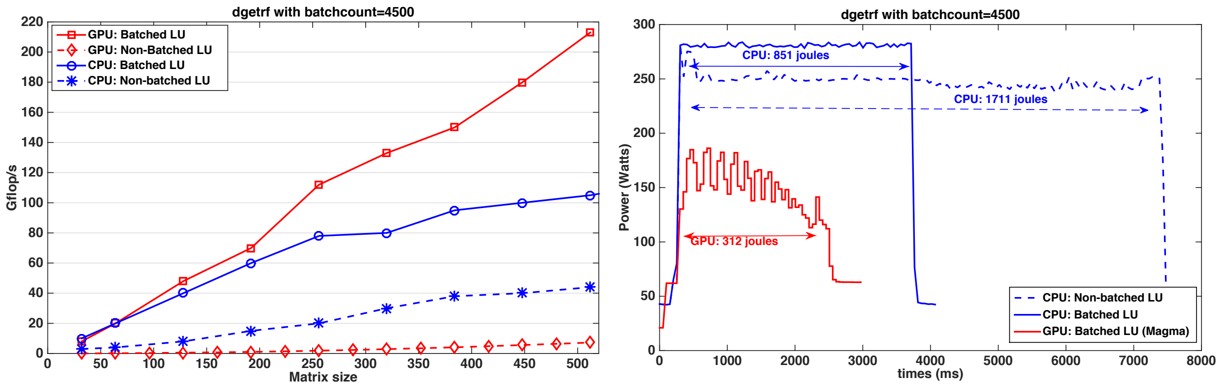


Figure 2: Speedup (Left) and power consumption (Right) achieved by the MAGMA batch LU factorization on NVIDIA K40c GPU vs. 16 cores of Intel Xeon ES-2670 (Sandy Bridge) 2.60GHz CPUs.

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, the Level 3 BLAS were proposed [17, 18], 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 [33, 34] 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 [21], which has the potential to automatically adapt to a memory hierarchy.

For the emerging multicore architectures of the 2000's, the PLASMA library [4] 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 the quantum Hall effect), astrophysics [40], metabolic networks [36], CFD and resulting PDEs through direct and multifrontal solvers [47], high-order FEM schemes for hydrodynamics [11],

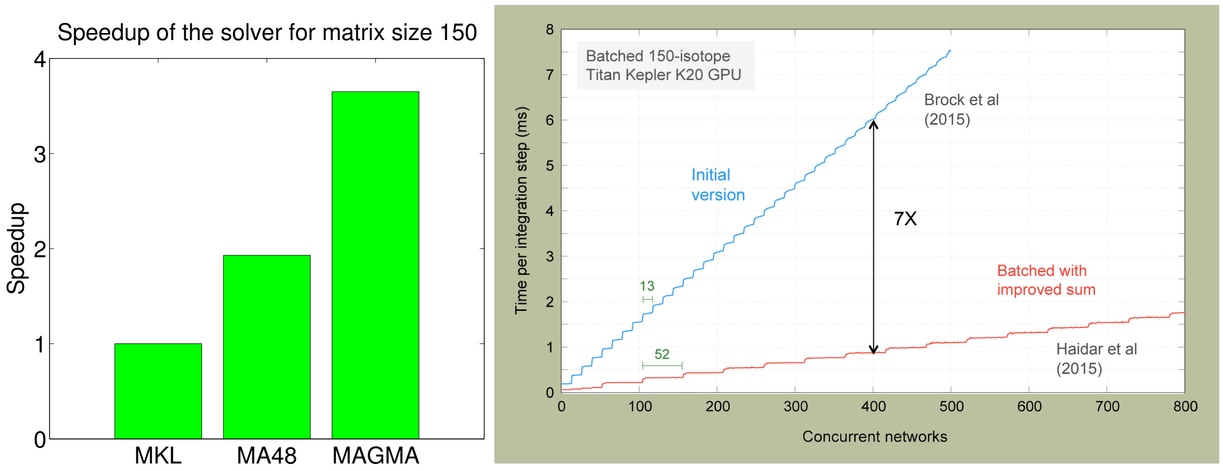


Figure 3: Acceleration of various applications using batched approach.

direct-iterative preconditioned solvers [31], quantum chemistry [7], image [41], and signal processing [6]. Batch LU factorization was used in subsurface transport simulation [45], whereby many chemical and microbiological reactions in a flow path are simulated in parallel [46]. Finally, small independent problems also occur as a very important aspect of computations on hierarchical matrices (H-matrices) [24].

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 [44, 3, 25]. 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, 26]. 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 2, Left illustrates this for the case of many small LU factorizations – even in a multicore setting the batch processing approach outperforms its non-batch counterpart by a factor of approximately two, while the batch approach in MAGMA [1] on a K40c GPU outperforms by about 2× the highly optimized CPU batch version running on 16 Intel Sandy Bridge cores [27]. 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 be 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 [10], and Intel has also included a batch matrix-matrix product (GEMM BATCH) in MKL [23]. Subsequently, batch factorizations, and the underlying Batched BLAS, can be used in applications. For example, the batch LU 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

[1]icl.utk.edu/magma

to 2× speedup over the MA48 factorization from the Harwell Subroutine Library [30], by solving hundreds of matrices of size 150 × 150 on the Titan supercomputer at ORNL [12]. Another example shown in Figure 3(b) is the astrophysical thermonuclear networks coupled to hydrodynamical simulations in explosive burning scenarios [9] that was accelerated 7× by using the batch approach.

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 readily be 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 Fortran interfaces.

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 [28, 29], Birds of a Feather sessions were held at SC17 in Denver, Colorado and at ISC 2018 in Frankfurt am Main, Germany and BBLAS talks have been given in a number of conferences, including WSSSPE4 in Manchester, SIAM CSE 17 in Atlanta, and SIAM PP18 in Tokyo. A previous proposal was presented in [14], see also [15] and [16].

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 is composed of 5 or 6 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 (denoted as a type template `fp_t`), as follows:
  - S indicates `float`
  - D indicates `double`
  - C indicates `complex`
  - Z indicates `double complex`
- 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 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

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

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

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

2.2.1 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:

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

- side has two possible values which are used by the routines as follows:
  - `BlasLeft`: Specifies to multiply a general matrix by symmetric, Hermitian, or triangular matrix on the left;
  - `BlasRight`: Specifies to multiply general matrix by symmetric, Hermitian, or triangular matrix on the right.
- trans_A, trans_B, 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, the values `BlasTrans` and `BlasConjTrans` have the same effect.
- 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.

2.2.2 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 3). 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 $k = 0$, 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 by n if working with rectangular A, and n by n if A is a square matrix. If there is only a single group of matrices of the same sizes (see Section 2.3.2), the m, n, and k values for all matrices are specified by the `m[0]`, `n[0]`, and `k[0]` values, respectively.

2.2.3 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 (`ld_A`, `ld_B`, or `ld_C`). 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. Similarly, the values of `ld_A[i]`, `ld_B[i]`, and `ld_C[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 2.3.2), the leading dimensions are specified by `ld_A[0]`, `ld_A[0]`, and `ld_A[0]` for all corresponding $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$ matrices.

2.2.4 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 2.3.2), the scalars are given by `alpha[0]` and `beta[0]`.

2.3 Groups of Same-Size Batched BLAS Routines

During the past standardization meetings [28, 29] 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. 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:

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

2.3.1 Specification of the number of matrices

The total number of matrices involved in a single call may be derived from two arguments: `group_count` and `group_sizes`. Formerly, this was known as the `batch_count` argument [14] – an integer that indicated the number of matrices to be processed. If there is more than one group of matrices, Eq. (1) may be used for calculating `batch_count`.

2.3.2 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 group-varying sizes (including operation options, sizes, matrix leading dimensions, and scalars), respectively. This was superseded by the group interface.

Note that through the group interface 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 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.

2.4 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:

- `BBLAS_ERRORS_REPORT_ALL`, 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_count[*]`.
- `BBLAS_ERRORS_REPORT_GROUP`, 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`.
- `BBLAS_ERRORS_REPORT_ANY`, 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.
- `BBLAS_ERRORS_REPORT_NONE`, 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`, `trans_A`, `trans_B`, `trans`, `uplo`, or `diag` whose meaning is not specified;
- If any of `m`, `n`, `k`, `ld_A`, `ld_B`, or `ld_C` is less than zero.

If no errors are detected, or `info[0]` is `BBLAS_ERRORS_REPORT_NONE` on input, then `info[0]` will be returned as zero.

The behavior of the error handling is determined by the input value of `info` as described above, but with full reporting, if a routine is called with an invalid value for arguments: `group_count` and `group_sizes` then the routine will return an error in `info[0]`. Errors related to other arguments are signaled with the number of the group in which the invalid argument was encountered (counting from one because the value of 0 is reserved for the return without an error). In other words, if a routine is called with an invalid value for any of its other arguments for a Batched BLAS operation in group g for matrix i, the routine will return an error in position `info[1+p+i]` that refers to the number of the first invalid argument (counting from one with number 0 reserved for successful completion) where p is the total number of matrices in groups 1 through $g-1$.

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 Specification of Batched BLAS Routines

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

3.1.1 General matrix-matrix products GEMM

Depending on the values of `trans_A` and `trans_B`, this routine performs a batch of one of the matrix-matrix operations described below, for which C is always an $m \times n$ matrix:

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

The calling routine is described as follows:

```
  BLAS_<*>gemm_batch(
   group_count : int                       : In ,
   group_sizes : int[group_count]          : In ,
   layout      : enum Layout               : In ,
   trans_A     : enum Transpose[count]     : In ,
   trans_B     : enum Transpose[count]     : In ,
   m           : int[count]                : In ,
   n           : int[count]                : In ,
   k           : int[count]                : In ,
   alpha       : <float,double,..>[count] : In ,
   A           : <float,double,..>[count] : In ,
   ld_A        : int[count]                : In ,
   B           : <float,double,..>[count] : In ,
   ld_B        : int[count]                : In ,
   beta        : <float,double,..>[count] : In ,
   C           : <float,double,..>[count] : InOut ,
   ld_C        : int[count]                : In ,
   info        : int[count]                : InOut
)
```

where "<*>" denotes one of the four standard floating-point arithmetic precisions (`float`, `double`, `complex`, or `double complex`). The `trans_A` and `trans_B` arrays can be of size one for the same size batch and of size at least $\sum$`group_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 `BlasTrans` and `BlasConjTrans` have the same effect. The `m`, `n`, and `k` arrays of integers are of size at least $\sum$`group_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 `A`, `B`, and `C` are of size at least $\sum$`group_count[*]` and point to the matrices $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$. The size of the matrix C_i is `m[i]*n[i]`. The sizes of the matrices A_i and B_i depend on `trans_A[i]` and `trans_B[i]`; their corresponding sizes are mentioned in the equation above. The arrays `ld_A`, `ld_B`, and `ld_C` define the leading dimension of each of the matrices $\{A_i$`[ld_A[i]][*]`$\}$, $\{B_i$`[ld_B[i]][*]`$\}$, $\{C_i$`[ld_C[i]][*]`$\}$, respectively[2].

If there is only one group of matrices (`group_count == 1`) 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 $\sum$`group_count[*]` where a value at position i reflects the argument error for gemm with matrices A_i, B_i, and C_i.

3.1.2 Hermitian and symmetric matrix-matrix products: HEMM and SYMM

These routines performs a batch of matrix-matrix products, each expressed in one of the following forms, for which B and C are $m \times n$ matrices:

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

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

where A is real symmetric (`<s,d>symm_batch`), complex symmetric (`<c,z>symm_batch`), or complex Hermitian (`<c,z>hemm_batch`), and α and β are real or complex scalars.

The calling routines are described as follows:

```
   BLAS_<*>symm_batch(
    group_count : int                     : In ,
    group_sizes : int[group_count]        : In ,
    layout      : enum Layout             : In ,
    side        : enum Side[count]        : In ,
    uplo        : enum UpLo[count]        : In ,
    m           : int[count]              : In ,
    n           : int[count]              : In ,
    alpha       : <float,double>[count] : In ,
    A           : <float,double>[count] : In ,
    ld_A        : int[count]              : In ,
    B           : <float,double>[count] : In ,
    ld_B        : int[count]              : In ,
    beta        : <float,double>[count] : In ,
    C           : <float,double>[count] : InOut ,
    ld_C        : int[count]              : In ,
    info        : int[count]              : InOut
)
   BLAS_<*>hemm_batch(
    group_count : int                              : In ,
    group_sizes : int[group_count]                 : In ,
    layout      : enum Layout                      : In ,
    side        : enum Side[count]                 : In ,
    uplo        : enum UpLo[count]                 : In ,
    m           : int[count]                       : In ,
    n           : int[count]                       : In ,
    alpha       : _Complex<float,double>[count] : In ,
    A           : _Complex<float,double>[count] : In ,
    ld_A        : int[count]                       : In ,
    B           : _Complex<float,double>[count] : In ,
    ld_B        : int[count]                       : In ,
    beta        : _Complex<float,double>[count] : In ,
    C           : _Complex<float,double>[count] : InOut ,
    ld_C        : int[count]                       : In ,
    info        : int[count]                       : InOut
)
```

The side array is of size at least $\sum$group_count[*] and each value defines the operation on each matrix as described in the equations above. The uplo array is of size at least $\sum$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 at least $\sum$group_count[*] and define the dimension of the operation on each matrix. The alpha and beta arrays provide the scalars α_i and β_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 $\sum$group_count[*] that point to the matrices $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$. The size of matrix C_i is m[i]*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 lda, ldb, and ldc define the leading dimension of each of the matrices $\{A_i$[ld_A[i]][*]$\}$, $\{B_i$[ld_B[i]][*]$\}$, $\{C_i$[ld_C[i]][*]$\}$, respectively.

The array `info` defines the error array. It is an output array of integers of size $\sum$`group_count[*]` where a value at position i reflects the argument error for hemm/symm with matrices A_i, B_i, and C_i.

3.1.3 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:

- $C \leftarrow \alpha \cdot A \times A^T + \beta \cdot C$ 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 $k \times n$

The calling routines are described as follows:

```
BLAS_<*>syrk_batch(
  group_count : int                       : In ,
  group_sizes : int[group_count]          : In ,
  layout      : enum Layout               : In ,
  uplo        : enum UpLo[count]          : In ,
  trans       : enum Transpose[count]     : In ,
  n           : int[count]                : In ,
  k           : int[count]                : In ,
  alpha       : <float,double>[count]     : In ,
  A           : <float,double>[count]     : In ,
  ld_A        : int[count]                : In ,
  beta        : <float,double>[count]     : In ,
  C           : <float,double>[count]     : InOut ,
  ld_C        : int[count]                : In ,
  info        : int[count]                : InOut
)
```

The `uplo` array is of size at least $\sum$`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 at least $\sum$`group_count[*]` where each value defines the operation on each matrix. In the real precision case, the values `BlasTrans` and `BlasConjTrans` have the same meaning. In the complex case, `trans == BlasConjTrans` is not allowed in syrk case. The `n` and `k` arrays of integers are of size at least $\sum$`group_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_i and C_i . The arrays of pointers `A` and `C` are of size $\sum$`group_count[*]` and point to the matrices $\{A_i\}$ and $\{C_i\}$. The size of matrix C_i is `m[i]*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 `ld_A` and `ld_C` define the leading dimension of each of the matrices $\{A_i$`[ld_A[i]][*]`$\}$, $\{C_i$`[ld_C[i]][*]`$\}$, respectively.

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

```
    BLAS_<*>herk_batch(
      group_count : int                           : In ,
      group_sizes : int[group_count]              : In ,
      layout      : enum Layout                   : In ,
      uplo        : enum UpLo[count]              : In ,
      trans       : enum Transpose[count]         : In ,
      n           : int[count]                    : In ,
      k           : int[count]                    : In ,
      alpha       : _Complex<float,double>[count] : In ,
      A           : _Complex<float,double>[count] : In ,
      ld_A        : int[count]                    : In ,
      beta        : _Complex<float,double>[count] : In ,
      C           : _Complex<float,double>[count] : InOut ,
      ld_C        : int[count]                    : In ,
      info        : int[count]                    : InOut
)
```

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 array info defines the error array. It is an output array of integers of size $\sum$group_count[*] where a value at position i reflects the argument error for herk with matrices A_i and C_i.

3.1.4 Rank-2k updates of a symmetric/Hermitian matrix HER2K and SYR2K

These 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 $k \times n$

The calling routines are described as follows:

```
    BLAS_<*>syr2k_batch(
  group_count : int                     : In ,
  group_sizes : int[group_count]        : In ,
  layout      : enum Layout             : In ,
  uplo        : enum UpLo[count]        : In ,
  trans       : enum Transpose[count]   : In ,
  n           : int[count]              : In ,
  k           : int[count]              : In ,
  alpha       : <float,double>[count]   : In ,
  A           : <float,double>[count]   : In ,
  ld_A        : int[count]              : In ,
  beta        : <float,double>[count]   : In ,
  C           : <float,double>[count]   : InOut ,
  ld_C        : int[count]              : In ,
  info        : int[count]              : InOut
)
```

The uplo array is of size $\sum$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 $\sum$group_count[*] where each value defines the operation on each matrix. In the real precision case, the values BlasTrans and BlasConjTrans have the same meaning. In the complex case, trans == BlasConjTrans is not allowed in syr2k batch. The n and k arrays of integers are of size $\sum$group_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 A, B, and C are the arrays of pointers of size $\sum$group_count[*] that point to the matrices $\{A_i\}$, $\{B_i\}$, and $\{C_i\}$. The size of matrix C_i is m[i]*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 ld_A, ld_B, and ld_C define the leading dimension of the matrices $\{A_i$[ld_A[i]][*]$\}$, $\{B_i$[ld_B[i]][*]$\}$, $\{C_i$[ld_C[i]][*]$\}$, respectively.

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

```
BLAS_<*>her2k_batch(
  group_count : int                              : In ,
  group_sizes : int[group_count]                 : In ,
  layout      : enum Layout                      : In ,
  uplo        : enum UpLo[count]                 : In ,
  trans       : enum Transpose[count]            : In ,
  n           : int[count]                       : In ,
  k           : int[count]                       : In ,
  alpha       : _Complex<float,double>[count]    : In ,
  A           : _Complex<float,double>[count]    : In ,
  ld_A        : int[count]                       : In ,
  B           : _Complex<float,double>[count]    : In ,
  ld_B        : int[count]                       : In ,
  beta        : _Complex<float,double>[count]    : In ,
  C           : _Complex<float,double>[count]    : InOut ,
  ld_C        : int[count]                       : In ,
  info        : int[count]                       : InOut
```

)

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 and that beta is real. The matrices $\{C_i\}$ are complex Hermitian.

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

3.1.5 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 α 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

The calling routines are described as follows:

```
BLAS_<*>trmm_batch(
 group_count : int                       : In ,
 group_sizes : int[group_count]          : In ,
 layout      : enum Layout               : In ,
 side        : enum Side[count]          : In ,
 uplo        : enum UpLo[count]          : In ,
 trans       : enum Transpose[count]     : In ,
 diag        : enum Diagonal[count]      : In ,
 m           : int[count]                : In ,
 n           : int[count]                : In ,
 alpha       : <float,double,..>[count]  : In ,
 A           : <float,double,..>[count]  : In ,
 ld_A        : int[count]                : In ,
 B           : <float,double,..>[count]  : In ,
 ld_B        : int[count]                : In ,
 info        : int[count]                : InOut
)
```

The side array is of size $\sum$group_count[*] and each value defines the operation on each matrix as described in the equations above. The uplo array is of size $\sum$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 $\sum$group_count[*] where each value defines the operation on each matrix. In the real precision case, the values BlasTrans and BlasConjTrans have the same meaning. The diag array is of size $\sum$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 $\sum$group_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 A and B are of size $\sum$group_count[*] and point to the matrices $\{A_i\}$ and $\{B_i\}$. The size of matrix B_i is m[i]*n[i]. The size of matrix A_i depends on side[i]; its corresponding size is mentioned in the equation above. The arrays ld_A and ld_B define the leading dimension of the $\{A_i$[ld_A[i]][*]$\}$, and $\{B_i$[ld_B[i]][*]$\}$ matrices, respective.

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

3.1.6 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 α 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

The calling routines are described as follows:

```
   BLAS_<*>trsm_batch(
    group_count : int                        : In ,
    group_sizes : int[group_count]           : In ,
    layout      : enum Layout                : In ,
    side        : enum Side[count]           : In ,
    uplo        : enum UpLo[count]           : In ,
    trans       : enum Transpose             : In ,
    diag        : enum Diagonal              : In ,
    m           : int[count]                 : In ,
    n           : int[count]                 : In ,
    alpha       : <float,double,..>[count]   : In ,
    A           : <float,double,..>[count]   : In ,
    ld_A        : int[count]                 : In ,
    B           : <float,double,..>[count]   : In ,
    ld_B        : int[count]                 : In ,
    info        : int[count]                 : InOut
)
```

The side array is of size $\sum$group_count[*] where each value defines the operation on each matrix as described in the equation above. The uplo array is of size $\sum$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 $\sum$group_count[*] where each value defines the operation on each matrix. In the real precision case, the values BlasTrans and BlasConjTrans have the same meaning. The diag array is of size $\sum$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 $\sum$group_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 A and B are of size $\sum$group_count[*] and point to the matrices $\{A_i\}$ and $\{B_i\}$. The size of matrix B_i is m[i]*n[i]. The size of the matrix A_i depends on side[i]; its corresponding size is mentioned in the equation above. The arrays ld_A and ld_B define the leading dimension of the matrices $\{A_i$[ld_A[i]][*]$\}$, and $\{B_i$[ld_B[i]][*]$\}$, respectively.

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

3.2 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 AXPY: $y \leftarrow \alpha \cdot x + y$ and the Level 2 GEMV: $y \leftarrow \alpha \cdot A \times x + \beta \cdot y$ BLAS routines.

3.2.1 Scaling a vector and adding another vector AXPY

```
BLAS_<*>axpy_batch(
    group_count : int                        : In ,
    group_sizes : int[group_count]           : In ,
    n           : int[count]                 : In ,
    alpha       : <float,double,..>[count]   : In ,
    X           : <float,double,..>[count]   : In ,
    inc_X       : int[count]                 : In ,
    Y           : <float,double,..>[count]   : In ,
    inc_Y       : int[count]                 : In ,
    info        : int[count]                 : InOut
)
```

Here, inc_X[i] and inc_Y[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.

3.2.2 General matrix-vector products GEMV

```
BLAS_<*>gemv_batch(
    group_count : int                        : In ,
    group_sizes : int[group_count]           : In ,
    layout      : enum Layout                : In ,
    trans_A     : enum Transpose[count]      : In ,
    m           : int[count]                 : In ,
    n           : int[count]                 : In ,
    alpha       : <float,double,..>[count]   : In ,
    A           : <float,double,..>[count]   : In ,
    ld_A        : int[count]                 : In ,
    beta        : <float,double,..>[count]   : In ,
    Y           : <float,double,..>[count]   : InOut ,
    inc_Y       : int[count]                 : In ,
    info        : int[count]                 : InOut
)
```

Array inc_Y[i] at the i^{th} position must not be zero and specifies the increment for the elements of Y[i].

4 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 [17] and Section 4.13 of [5].

5 Specification of Batch 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 are 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, gives the following batch version:

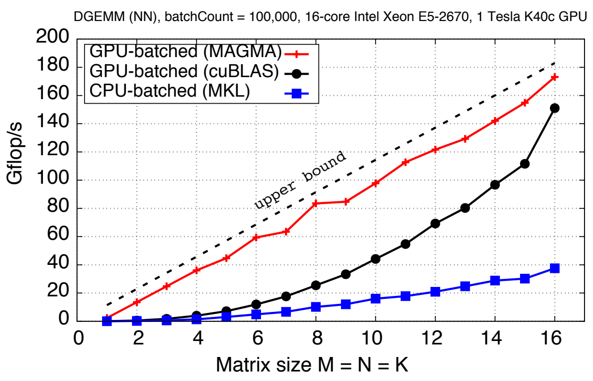
```
  LAPACK_<*>getrf_batch(
    group_count : int                          : In ,
    group_sizes : int[group_count]             : In ,
    layout      : enum Layout                  : In ,
    m           : int[count]                   : In ,
    n           : int[count]                   : In ,
    A           : <float,double,..>[count]     : In ,
    ld_A        : int[count]                   : In ,
    piv         : int[count]                   : In ,
    info        : int[count]                   : InOut
)
```

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 [32].

Thus, splitting an algorithm into hierarchical tasks that block the computation over the available memory hierarchies (in order 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 [2] are released through the MAGMA library, providing a model Batched BLAS 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 4. 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 like grouping several GEMMs to be executed on the same multiprocessor, vectorization across GEMMs, along with data prefetching optimizations, are used in order to reach 90+% of the theoretical peak on either multicore CPUs or GPUs [1, 39] (see Figure 4, 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 4,



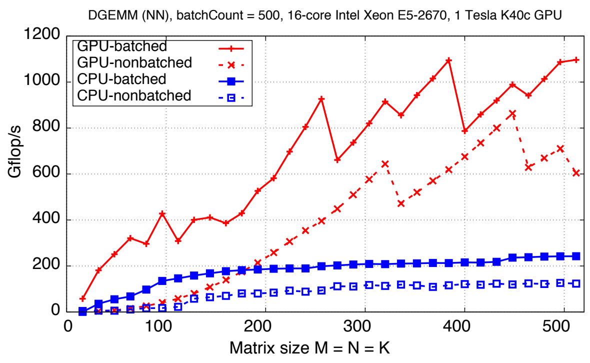


Figure 4: Performance of batch DGEMM versions on matrices of size less than 32 (Left) and larger (Right) on a K40c GPU and 16 cores of Intel Xeon ES-2670 (Sandy Bridge) 2.60 GHz CPUs.

Right). For these sizes, similar to CPUs, coding multilevel blocking types of algorithms on GPUs must be in native machine language in order to overcome some limitations of the CUDA compiler or warp scheduler (or both) [43]. Assembly implementations [37, 22] 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.

7 Future Directions and Final Remarks

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

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, 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.

Acknowledgments

This material is based upon work supported in part by the European Union's Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633, the National Science Foundation under Grants No. CSR 1514286 and ACI-1339822, NVIDIA, the Department of Energy, and in part by the Russian Scientific Foundation, Agreement N14-11-00190.

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. High-performance tensor contractions for GPUs. Technical Report UT-EECS-16-738, University of Tennessee Computer Science, 01-2016 2016.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance, design, and autotuning of batched GEMM for GPUs. In *ISC High Performance 2016*, Frankfurt, Germany, 06-2016 2016.

[3] 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.

[4] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Jack Langou, Haitem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):(5 pages), 2009.

[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. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[6] Michael J. Anderson, David Sheffield, and Kurt Keutzer. A predictive model for solving small linear algebra problems in GPU registers. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 2–13, 2012.

[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. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, January 2006.

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

[9] Benjamin Brock, Andrew Belt, Jay J. Billings, and Mike Guidry. Explicit Integration with GPU Acceleration for Large Kinetic Networks. *J. Comput. Phys.*, 302(C):591–602, January December 2015. Preprint: http://arxiv.org/abs/1409.5826.

[10] cuBLAS 7.5, 2016. Available at http://docs.nvidia.com/cuda/cublas/.

[11] 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.

[12] Tingxing Dong, Azzam Haidar, Piotr Luszczek, A. Harris, Stanimire Tomov, and Jack 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.

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

[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. A proposed API for Batched Basic Linear Algebra Subprograms. MIMS EPrint 2016.25, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, April 2016. http://eprints.ma.man.ac.uk/2464/.

[15] Jack Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. Creating a standardised set of batched BLAS routines. In Gabrielle Allen, Jeffrey Carver, Sou-Cheng T. Choi, et al., editors, *Proceedings of the Fourth Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE4)*, volume 1686. CEUR Workshop Proceedings, 2016. http://ceur-ws.org/Vol-1686/WSSSPE4_paper_3.pdf.

[16] Jack Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. The design and performance of batched BLAS on modern high-performance computing systems. *Procedia Computer Science*, 108:495–504, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, http://dx.doi.org/10.1016/j.procs.2017.05.138.

[17] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and Sven Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990.

[18] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and Sven Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:18–28, March 1990.

[19] Jack J. Dongarra, J. Du Croz, Sven Hammarling, and R. Hanson. Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, March 1988.

[20] Jack J. Dongarra, J. Du Croz, Sven Hammarling, and R. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, March 1988.

[21] 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.

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

[23] 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.

[24] W. Hackbusch. A Sparse Matrix Arithmetic Based on H-matrices. Part I: Introduction to H-matrices. *Computing*, 62(2):89–108, May 1999.

[25] 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.

[26] 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.

[27] 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.

[28] Sven Hammarling. Workshop on batched, reproducible, and reduced precision BLAS. Technical report, The University of Manchester, Manchester, UK, 2016. eprints.maths.manchester.ac.uk/2494/.

[29] Sven Hammarling. Second workshop on batched, reproducible, and reduced precision BLAS. Technical report, The University of Manchester, Manchester, UK, 2017. eprints.maths.manchester.ac.uk/2543/.

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

[31] 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.

[32] Khairul Kabir, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. On the design, development, and analysis of optimized matrix-vector multiplication routines for coprocessors. In *ISC High Performance 2015*, Frankfurt, Germany, 07-2015 2015.

[33] 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.

[34] 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.

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

[36] 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.

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

[38] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[39] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, J. Falcou, and Jack J. Dongarra. High-performance matrix-matrix multiplications of very small matrices. Technical Report UT-EECS-16-740, University of Tennessee Computer Science, 03-2016 2016.

[40] 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.

[41] 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.

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

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

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

[45] Oreste Villa, Massimiliano Fatica, Nitin Gawande, and Antonino Tumeo. *Power/Performance Trade-Offs of Small Batched LU Based Solvers on GPUs*, pages 813–825. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[46] Oreste Villa, Nitin Gawande, and Antonino Tumeo. Accelerating subsurface transport simulation on heterogeneous clusters. In *2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, IN, USA, September 23-27, 2013*, pages 1–8, 2013.

[47] Sencer Nuri Yeralan, Timothy A. Davis, Wissam M. Sid-Lakhdar, and Sanjay Ranka. Algorithm 980: Sparse QR Factorization on the GPU. *ACM Trans. Math. Softw.*, 44(2):17:1–17:29, August 2017.