C++ API for Batch BLAS

Ahmad Abdelfattah	ICL[1]
Konstantin Arturov	Intel[2]
Cris Cecka	NVIDIA[3]
Jack Dongarra	ICL
Chip Freitag	AMD[4]
Mark Gates	ICL
Azzam Haidar	ICL
Jakub Kurzak	ICL
Piotr Luszczek	ICL
Stan Tomov	ICL
Panruo Wu	ICL

[1]Innovative Computing Laboratory
[2]Intel Corporation
[3]NVIDIA Corporation
[4]Advanced Micro Devices, Inc.

February 21, 2018

ICL INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY OF TENNESSEE KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

Revision	Notes
06-2017	first publication
01-2018	copy editing, improved artwork, new cover

```
@techreport{luszczek2017cpp,
  author={Abdelfattah, Ahmad and Arturov, Konstantin and Cecka, Cris and
          Dongarra, Jack and Freitag, Chip and Gates, Mark and Haidar, Azzam
          and Kurzak, Jakub and Luszczek, Piotr and Tomov, Stan and Wu, Panruo},
  title={{SLATE} Working Note 4: C++ {API} for Batch {BLAS}},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2017},
  month={December},
  number={ICL-UT-17-12},
  note={revision 01-2018}
}
```

Contents

1 Introduction **1**
- 1.1 What is a Batch Routine? . . . 1
- 1.2 Why are Batch Routines Important? . . . 1
  - 1.2.1 High Demand for Batch Computation in Scientific Applications . . . 1
  - 1.2.2 Increasing Parallelism in Hardware Architectures . . . 2
  - 1.2.3 Fair Performance of Existing Numerical Software on Batch Workloads . . . 3
- 1.3 Standardization Effort for Batch BLAS . . . 6
  - 1.3.1 Naming Conventions . . . 6
  - 1.3.2 Argument Conventions . . . 6
  - 1.3.3 Error Handling . . . 7
  - 1.3.4 Sample APIs . . . 8
  - 1.3.5 Discussion and Critique . . . 9
- 1.4 Summary . . . 9

2 Existing Solutions **11**
- 2.1 NVIDIA cuBLAS . . . 11
  - 2.1.1 Interface with Array of Pointers . . . 12
  - 2.1.2 Interface with Single Pointer and Stride . . . 13
  - 2.1.3 Routines from LAPACK with Pointer Array Interface . . . 14
  - 2.1.4 Routines with LAPACK-like Functionality with a Pointer Array Interface . . . 16
  - 2.1.5 Summary . . . 16
- 2.2 Intel® MKL . . . 17
  - 2.2.1 GEMM . . . 17
  - 2.2.2 TRSM . . . 18
- 2.3 AMD hipBLAS/rocBLAS . . . 18
- 2.4 ICL MAGMA . . . 20
  - 2.4.1 Error Handling in MAGMA . . . 21
  - 2.4.2 Advanced MAGMA APIs . . . 21
  - 2.4.3 Discussion and Critique . . . 22

2.5 Summary of Existing APIs 23

3 Proposed APIs **25**
3.1 The Objective 25
3.2 Compliance with the C++ BLAS API 26
3.3 General Design Principles 27
3.4 BLAS Error Checking 30
3.5 Size Error Checking 33
3.6 Reference Implementation 33
3.7 Extension for Group-Based APIs 35
3.8 Extension for Stride-Based APIs 36
3.9 Discussion and Critique 36
3.10 Summary 37

CHAPTER 1

Introduction

This chapter introduces the definition of batch routines, and their impact on today's scientific applications. The chapter highlights the ongoing efforts to establish standardization for Batch Basic Linear Algebra Subroutines.

1.1 What is a Batch Routine?

A batch routine is defined as a piece of software that applies the same operation on many independent problems, potentially in a parallel fashion. In the context of dense linear algebra, a batch routine applies a basic linear algebra subprogram (BLAS) or Linear Algebra PACKage (LAPACK) operation to an ideally large number of relatively small independent problems. The batch can have problems of the same size (fixed size) or different sizes (variable size).

1.2 Why are Batch Routines Important?

The answer to this question is three-fold. Each reason is discussed in detail below.

1.2.1 High Demand for Batch Computation in Scientific Applications

The first, and most important, reason is that many higher-level solvers and scientific applications require high-performance batch dense linear algebra software. In fact, the absence of a mature software for such workloads has sparked some in-house developments of batch routines for

specific purposes. For example, batch LU factorization has been used in subsurface transport simulation [16], where many chemical and microbiological reactions in a flow path are simulated in parallel [17]. A batch Cholesky factorization and the triangular solve have also been used to accelerate an alternating least square (ALS) solver that generates product recommendations on the basis of implicit feedback datasets [7, 11]. Batch matrix-matrix multiplication (GEMM) is at the core of many tensor contraction problems [1, 14]. Sparse direct solvers, such as SuiteSparse,[1] have huge dependencies on many batch BLAS and LAPACK routines [13, 18], including matrix multiplication and one-sided factorization (LU, QR, and Cholesky). The generation of block-Jacobi preconditioners has also been accelerated using batch matrix inversion [4].

The workload pattern of small independent problems is also very important to computations on hierarchical matrices (H-matrices) [9]. Very large dense matrices (e.g., covariance matrices) are often beyond the storage capabilities of a single modern node, and applying LAPACK algorithms (with cubic complexities) directly on them is very time consuming. H-matrices represent a dense matrix using a much smaller memory footprint through hierarchical representations, which exploit and compress low-rank, off-diagonal blocks of the original matrix. Such a compression can be accomplished using batch QR factorization and singular value decompositions (SVDs) [5]. In addition to the compression phase, computations on H-matrices can benefit from batch BLAS and LAPACK routines as well. The work by Akbudak et al. [3] shows the huge potential for the development of a batch routine for tile low-rank Cholesky factorization, which is used to solve a climate modeling problem on a hierarchically compressed dense matrix. To conclude, there is an obvious and growing research interest in high-performance batch dense linear algebra operations.

1.2.2 Increasing Parallelism in Hardware Architectures

The second reason to consider batch routines is the increased level of parallelism in hardware architectures. There has been a trend to move from *multi-core* architectures to *many-core* architectures. Many-core architectures possess considerably more processing cores than today's multi-core architectures, but they are relatively small and simple compared to the processing cores of today's multi-core CPUs. Both graphics processing units (GPUs) and the Intel Xeon Phi architecture [2] represent a move towards this many-core paradigm of *more but simpler* cores. The simpler architecture of each processing core means that many-core architectures are more throughput oriented, and it is therefore the responsibility of the software—rather than large caches or deep pipelines—to take advantage of massively parallel architectures to achieve high performance.

In batch workloads, the size of a single problem is often too small to provide enough work for the underlying hardware. Without batch routines, batch workloads can be processed by sequentially looping over the problems and invoking a *standard*, non-batch routine at each iteration on a certain problem. This approach does not yield any good performance on batch workloads unless a single problem is large enough to fill up the resources of the hardware. On the other hand, batch routines take advantage of such *data-parallel* workloads and provide enough work for the underlying hardware—even if the problems are individually small. In fact,

[1] http://faculty.cse.tamu.edu/davis/suitesparse.html
[2] http://www.intel.com/XeonPhi/Processors

the performance gap between batch solutions and standard, non-batch solutions is inversely proportional to the size of the individual problem. The smaller the problem size, the more speedup is scored by batch routines over standard approaches.

As an example, consider performing a batch `GEMM` operation in double precision arithmetic using two examples of a many-core architecture. Figure 1.1 shows a performance comparison between the standard Intel Math Kernel Library's (MKL's) `DGEMM` routine and its batch variant using an Intel Xeon Phi Knights Landing (KNL) coprocessor, while Figure 1.2 shows a similar comparison for NVIDIA's cuBLAS running on an NVIDIA Volta V100 GPU. As mentioned before, small problems cannot provide enough parallel work for the hardware. This is why we observe huge speedups scored by the batch routines over the standard approaches, as the former takes advantage of the workload's data-parallel property. As the problem sizes become larger, the gap shrinks consistently until no further benefit is observed for using the batch routines. This is where a single problem has enough parallelism for the underlying hardware, and where the use of batch routines is unnecessary or not even recommended.

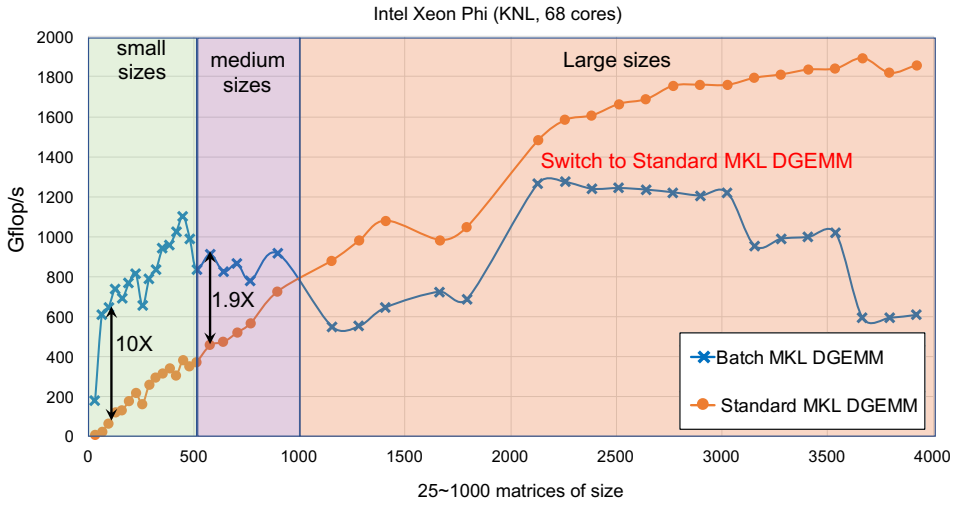

Figure 1.1: Performance comparison between batch `DGEMM` and standard `DGEMM` on a batch workload. For every size, the batch size is adjusted to fill the hardware resources. MKL (version 2017.0.2) is configured with 68 threads.

1.2.3 Fair Performance of Existing Numerical Software on Batch Workloads

The majority of numerical linear algebra libraries are specifically designed and tuned to perform well on large problem sizes. The previous section shows that standard, non-batch approaches, while eligible, are not suitable for batch workloads. A question now arises: can we benefit from parallelizing existing numerical software using standard programming models like Open Multi-Processing (OpenMP)? It might be a candidate for adding a parallelization layer over an existing non-batch routine. This approach addresses the lack of parallelism imposed by the small size of the individual problems. By launching parallel runs of a non-batch routine, the hardware is better utilized, and higher performance is expected. The parallelization layer

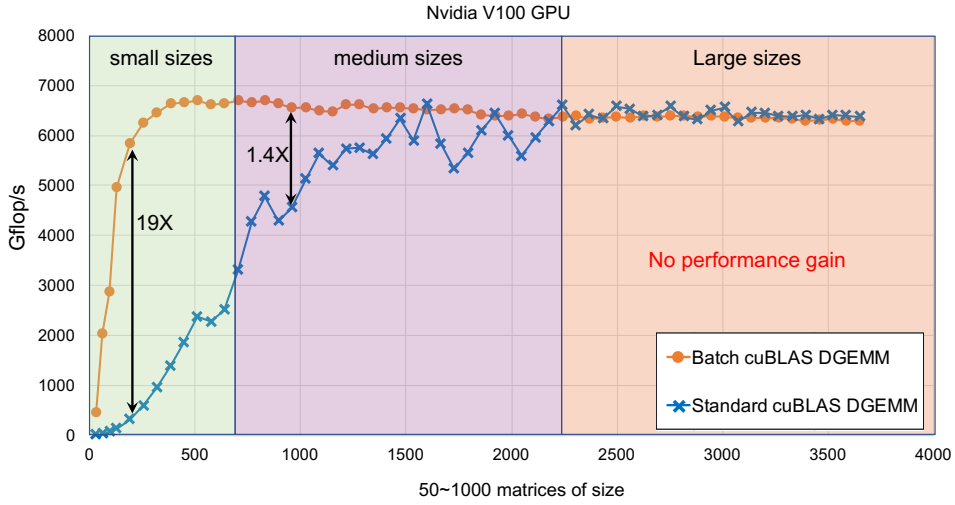

Figure 1.2: Performance comparison between batch `DGEMM` and standard `DGEMM` on a batch workload using cuBLAS (CUDA 9.0.102). For every size, the batch size is adjusted to fill the GPU resources.

can be realized on both CPU-based architectures (e.g., through OpenMP parallel loops) and GPU-based architectures (e.g., through concurrent execution queues). While the approach is a better alternative than the standard sequential method, it still trails behind batch routines in terms of performance. As an example, consider the same experiment shown in Figures 1.1 and 1.2, where we now add a parallelization layer to the standard routine (through OpenMP on the Intel KNL and through CUDA streams on the NVIDIA V100 GPU). Figures 1.3 and 1.4 show much smaller gaps between the two graphs. On the Intel KNL hardware, the batch routine still scores decent speedups against the MKL+OpenMP combination. For most sizes larger than 40×40, the OpenMP solution is very similar in performance (or even better) than the batch routine. On the V100 GPU, the batch routine is still faster than using concurrent streams, especially on very small sizes, where the speedup remain huge. However, the two graphs now meet much sooner than in Figure 1.2 (at size $\sim 320 \times 320$ instead of $\sim 2,000 \times 2,000$).

Another point in this regard is that some existing numerical solutions, by design, cannot benefit that much from the parallelization layer. For example, the Matrix Algebra on GPU and Multicore Architectures (MAGMA) library [3] uses a hybrid CPU-GPU design for its implementation of most LAPACK algorithms. This is a nearly perfect strategy for large problems. MAGMA is built on the assumption that trailing matrix updates on the GPU can hide both the CPU activity and the CPU-GPU communication [15]. Such an assumption completely goes away for small problems, which means that communication becomes the bottleneck in batch routines rather than the lack of parallel runs. Another example is the blocking technique, for which LAPACK is famous. While it has been shown that blocking is beneficial for medium-sized batch workloads [10], the blocking technique, by nature, imposes a redundant memory traffic that is unaffordable for very small sizes. For example, the cost of writing a factorized panel and reading it back for the update stage remains significant in small problems. This explains why many research efforts adopt a fused design approach, where all the computational steps are performed in the same

[3] http://icl.cs.utk.edu/magma/

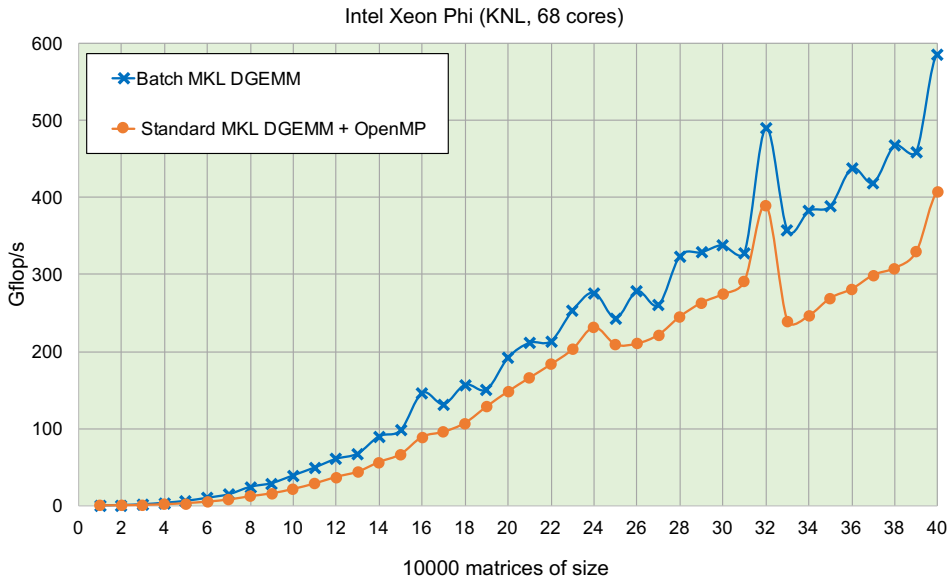

Figure 1.3: Performance comparison between batch `DGEMM` and parallelized standard `DGEMM` on a batch workload. The parallelized standard solution invokes single-threaded MKL (version 2017.0.2) within an OpenMP `parallel for` loop. The number of OpenMP threads is 68.

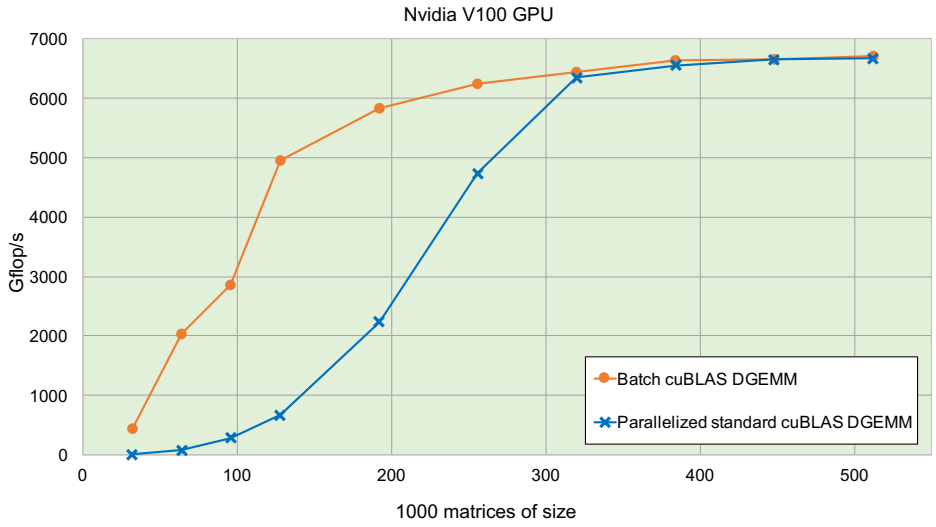

Figure 1.4: Performance comparison between batch `DGEMM` and the parallelized standard `DGEMM` on a batch workload using cuBLAS (CUDA 9.0.102).

context while keeping the data in the fastest memory levels as much as possible [2, 4, 11].

1.3 Standardization Effort for Batch BLAS

Owing to its critical importance and potentially great impact, there have been some community efforts—including efforts from vendors and the research community—to establish a standard application programming interface (API) for batch BLAS.[4] Such efforts have led to a preliminary definition for the batch BLAS API [6], the introduction of numerically reproducible BLAS,[5] and some other efforts that utilize batch BLAS in real-world applications.

The first published standard for batch BLAS [6] defines naming conventions, type definitions, and C interfaces for batch level-3 BLAS routines. 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 enable the entire linear algebra community to collectively attack a wide range of small matrix problems. The published standard is not final, and is, in fact, going through several iterations until the linear algebra community reaches a consensus regarding the specifications of the interface.

1.3.1 Naming Conventions

The first published standard suggests that a batch BLAS routine follows, and extends as needed, the conventions of the corresponding BLAS routine. In particular, the name is composed of five characters specifying the BLAS routine followed by the suffix `_batch`. For example, the `dgemm_batch` routine implements batch general matrix-matrix multiplication in double precision arithmetic, and the `ctrsm_batch` routine implements batch triangular solve in complex precision.

1.3.2 Argument Conventions

The proposed standard also follows a convention for the list of arguments that is similar to the convention used for BLAS, with the necessary adaptations for the batched operations. In general, the standard promotes scalar arguments found in BLAS into array arguments for batch BLAS. As an example, arguments that specify options (e.g., transpositions, conjugation, $\cdots$), sizes, and leading dimensions are promoted into arrays instead of scalars. The input/output matrices are also passed as arrays of pointers. Three additional arguments are also proposed to all routines. The first is an integer that specifies the size of the batch, (i.e., the number of BLAS operations to be performed). The second is an enumerated type called `batch_opts`, which specifies certain options and styles regarding the batch (e.g., fixed size vs. variable size). The third is an array of integers called `info_array`, which is used to report errors to the users.

[4] http://icl.utk.edu/bblas/
[5] http://bebop.cs.berkeley.edu/reproblas/

1.3.3 Error Handling

As mentioned before, the `info_array` argument replaces the legacy `XERBLA()` function, which is used to report errors in the standard BLAS implementation. The use of `XERBLA()` guarantees that errors are reported regardless of whether the caller code checks for errors. The default implementation of `XERBLA()` can also be overridden if another behavior is desired upon detecting errors. However, there are some reasons to reject the use of `XERBLA()` for batch BLAS:

1. The default `XERBLA()` implementation is not sufficiently precise for complex runtime error scenarios: If a BLAS routine is called in a loop, then the input/output (I/O) buffer or the console screen will be flooded with error messages. This would require a custom implementation for `XERBLA()` that suppresses error messages, which is a mode of operation available only in few development workflows.

2. The use of global state: `XERBLA()` requires global variables for non-trivial customization and information passing between the user and the BLAS library.

3. Dependence on platform-specific features: Dynamic libraries often require special features in the binary format of the operating system (OS) to overload a function. This is not hardware specific but also involves the accompanying software stack, including the OS and the compiler-linker tool chain.

4. Limited customization: There can only be one `XERBLA()` per executable, and there is no mechanism for chaining or queueing its invocations in case two different call sites would like to install different error handlers. Furthermore, there is no way to establish a protocol between call sites for cooperative error handling, because the only feature available is the linker name replacement system, which is available in Linux and Mac OS X and used when creating Executable and Linkable Format (ELF) files or Mach-O object files.

5. Language-specific behavior is dependent on name mangling: Modern BLAS standards and their implementations expose the Fortran API and the C API. The older CBLAS standard implements functions like `cblas_dgemm()`, and the newer standard uses `BLAS_dgemm()`. The `XERBLA()` mechanism requires resolving the coexistence of both language bindings (Fortran and C), sometimes in the same binary. Neither of these languages necessarily share the same I/O streams, and—in a mixed programming language environment—it is not obvious which `XERBLA()` binding needs to be reimplemented to take over the BLAS error handling.

6. Mixing computational capabilities with I/O facilities: According to the standard's definition of `XERBLA()`, the use of I/O streams is required by the default implementation inside the BLAS library. This obviously causes issues for headless mode operation when the access to the I/O facilities is restricted to accommodate custom environments. For example, on an embedded system or on a cloud platform, the only available I/O steam might occur during limited system logging or in extra overheads generated by system-wide synchronization.

7. Lack of support for the asynchronous interface: The `XERBLA()` error handling mechanism is not meant for the asynchronous and event-based processing that has become prevalent

on modern high-performance computing (HPC) architectures. Modern computing hardware features multiple execution streams that add flexibility to scheduling at the hardware level but do not guarantee a specific order of completion for independent subroutine calls. This means that the XERBLA()-based library cannot be wrapped inside such an interface because error delivery is independent of the error-causing invocation. Connecting the two would also add unnecessary complexity and synchronization and thus diminish the potential benefits of asynchronous execution.

8. Lack of support for multithreading: The BLAS interface with XERBLA() is inherently single threaded. Multiple threads that asynchronously call BLAS and cause invocation of XERBLA() must be synchronized to provide coherent error reporting. The behavior under such circumstances is unspecified, and extra care has to be devoted to recognize the calling thread (e.g., with calls to pthread_self() or omp_get_num_threads()) and contextualize the error.

9. Software for Linear Algebra Targeting Exascale (SLATE): The C++ API design for BLAS and LAPACK in SLATE [8] aims to use C++ exceptions for error reporting rather than XERBLA().

The reasons listed above have encouraged the investigation and use of alternate error handling mechanisms, either by returning error codes (e.g., in C and Fortran interfaces) or by throwing exceptions (e.g., in C++ interfaces).

1.3.4 Sample APIs

According to the first published standard for the batch BLAS API [6], a batch general matrix-matrix multiplication in double precision should look like:

```
void dgemm_batch(enum trans_t *transA, enum trans_t *transB,
                 int *m, int *n, int *k,
                 double *alpha, double **A_array, int *lda,
                                double **B_array, int *ldb,
                 double *beta,  double **C_array, int *ldc,
                 int  batch_count,
                 enum batch_opts_t batch_opts,
                 int *info_array)
```

The transA and transB arrays can be of size one for batches of the same size, or of size batch_count for the variable sizes case. For the latter, each value defines the operation on the corresponding matrix. The m, n, and k arrays of integers are of size 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 (recall the standard GEMM operation: $C = \alpha AB + \beta C$). The arrays of pointers A_array, B_array, and C_array are of size at least batch_count and point to the matrices {Ai}, {Bi}, and {Ci}. The size of matrix {Ci} is m[i]×n[i]. The sizes of the matrices {Ai} and {Bi} depend on transA[i] and transB[i]. The arrays of leading dimensions lda, ldb, and ldc define the leading dimension of each of the matrices {Ai(lda,*)}, {Bi(ldb,*)}, and {Ci(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 info_array

argument 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 Ai, Bi, and Ci. The definition of other BLAS routines in the standard is quite similar to GEMM in terms of notation and argument types.

1.3.5 Discussion and Critique

The APIs of the proposed standard uses a flag mechanism to distinguish between batches of fixed size and batches of variable size. The user has to set batch_opts properly to reflect the nature of the input batch. According to the standard, setting batch_opts to BATCH_FIXED means that the batch has matrices of the same size, while setting it to BATCH_VARIABLE means that the batch has matrices of different sizes. However, the definitions of BATCH_FIXED and BATCH_VARIABLE seem inaccurate and lack flexibility for some use cases, where they unnecessarily restrict some arguments based on the sizes being fixed or variable. The following examples are for the GEMM API, but they are generally applicable to any BLAS routine.

1. In the BATCH_FIXED mode, the values of alpha and beta are assumed to be the same across the batch. This is an unnecessary restriction. In fact, it is unrelated to the sizes of the input matrices. For example, with the current standard, if the sizes are the same but the values of alpha and beta differ, the user has to configure the call as BATCH_VARIBALE and replicate the sizes across the arrays m, n, and k.
2. Similarly, the interface assumes the same leading dimension if BATCH_FIXED is defined. In general, matrices of the same size can have different leading dimensions.
3. On the other hand, the BATCH_VARIABLE mode must accept array arguments for alpha and beta. A variable size batch can, in general, have the same alpha and beta, but this is not supported in the current standard.

A more accurate definition for the BATCH_FIXED mode is one where all non-data arguments are the same across the batch. For GEMM, this includes transpositions, scaling arguments, sizes, and leading dimensions. The BATCH_VARIABLE mode assumes that all arguments vary from one problem to another. Despite being generic, such an API can be quite cumbersome in some use cases where a certain argument has the same value across the batch (e.g., same k in a variable size batch GEMM). The user has to create an array whose entries have that exact same value. Such a lack of flexibility can be addressed by expanding the batch_opts argument to include more fine-grained options. However, this expansion makes it more difficult for the user to correctly use the API. The use of more advanced languages like C++ can address the same concerns in a more elegant and easy-to-use way.

1.4 Summary

This chapter introduced a definition of batch routines with an emphasis on its importance for several scientific applications. The need to develop dedicated routines for batch workloads has been discussed and justified. The chapter also highlights the ongoing standardization efforts

for batch BLAS APIs, with some discussion about its advantages and shortcomings. The next chapter provides a survey of the available APIs for batch BLAS routines in many software libraries that are widely used by the community.

CHAPTER 2

Existing Solutions

This chapter presents an overview of existing batch BLAS solutions, and their functionalities, from vendors and library developers. It also discusses solutions that are available for both CPUs and GPU accelerators.

2.1 NVIDIA cuBLAS

NVIDIA provides a rich set of batch BLAS calls in the cuBLAS library.[1] This library is available as part of the CUDA toolkit, which is comprised of compilers, libraries, debugging tools, and performance analysis tools that support NVIDIA's GeForce, Quadro, and Tesla GPU cards.[2] Despite a wide range of supported accelerator devices, the API looks exactly the same for all hardware, and the choice of an optimal code path is taken care of internally. The installation requires "superuser" access to install kernel drivers, and there must be at least one supported device/card present in order for any of the calls to succeed. In other words, the functionality cannot be provided on CPU-only systems, nor on systems with GPUs that are not CUDA-enabled.

The interface has multiple variants that depend on data types and on the way in which the matrices' location is encoded on input and output. Also, different groups of routines are available: level-3 BLAS, LAPACK, and LAPACK-like routines. At the same time, however, not all possible combinations are provided, and Section 2.1.5 provides a quick overview of the functionality, which should help the user discover whether or not a given combination is supported. The following sections try to provide an intuitive layout of the available functionality and might seem lacking at first reading due to the uneven coverage of the possible combinations.

[1] https://developer.nvidia.com/cublas
[2] https://developer.nvidia.com/cuda-toolkit

2.1.1 Interface with Array of Pointers

The "array of pointers" interface requires that the matrices be identified by pointers that are then stored in arrays of the appropriate type. For example, for the 64-bit floating-point data type, `double`, the pointers to matrices `A` will be stored in array `double *Aarray[]`. Complex numbers are represented by CUDA-specific `cuComplex` and `cuDoubleComplex` data types for 32-bit and 64-bit floating-point numbers, respectively.

GEMM

The batch GEMM routine in cuBLAS is available in five precisions.

Half precision: `cublasHgemmBatched()`

```
cublasStatus_t
cublasHgemmBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const __half *alpha, const __half *Aarray[], int lda,
                         const __half *Barray[], int ldb,
    const __half *beta,        __half *Carray[], int ldc,
    int batchCount);
```

Single precision: `cublasSgemmBatched()`

```
cublasStatus_t
cublasSgemmBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const float *alpha, const float *Aarray[], int lda,
                        const float *Barray[], int ldb,
    const float *beta,        float *Carray[], int ldc,
    nt batchCount);
```

Double precision: `cublasDgemmBatched()`

```
cublasStatus_t
cublasDgemmBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const double *alpha, const double *Aarray[], int lda,
                         const double *Barray[], int ldb,
    const double *beta,        double *Carray[], int ldc,
    int batchCount);
```

Complex precision: `cublasCgemmBatched()`

```
cublasStatus_t
cublasCgemmBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const cuComplex *alpha, const cuComplex *Aarray[], int lda,
                            const cuComplex *Barray[], int ldb,
    const cuComplex *beta,        cuComplex *Carray[], int ldc,
    int batchCount);
```

Double complex precision: `cublasZgemmBatched()`

```
cublasStatus_t
cublasZgemmBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const cuDoubleComplex *alpha, const cuDoubleComplex *Aarray[], int lda,
                                  const cuDoubleComplex *Barray[], int ldb,
    const cuDoubleComplex *beta,        cuDoubleComplex *Carray[], int ldc,
    int batchCount);
```

Apart from the pointer array arguments, the cuBLAS interface assumes that all arguments have the same value across the batch. The scalars `alpha` and `beta` are passed by their addresses, which can be in the host memory or in the GPU memory.

The cuBLAS library also provides other batch BLAS functions (e.g., triangular solves [TRSM]). However, this routine is not available in half precision. The signature of the batch TRSM routine in double precision looks like:

`cublasDtrsmBatched()`

```
cublasStatus_t
cublasDtrsmBatched(
    cublasHandle_t handle,
    cublasSideMode_t side, cublasFillMode_t uplo, cublasOperation_t trans, cublasDiagType_t diag,
    int m, int n,
    const double *alpha, double *A[], int lda,
                         double *B[], int ldb,
    int batchCount);
```

2.1.2 Interface with Single Pointer and Stride

In some applications, the batches of matrices are laid out in a regular pattern inside a large memory region. For such cases, the strided interface might be a better fit because it allows the user to specify the pointer to the first matrix and the displacement stride for each subsequent matrix in the batch. It is worth noting that only one level 3 BLAS routine is supported for this interface, and the remaining routines either correspond to the LAPACK routines or represent LAPACK-like functionality.

GEMM

Similar to batch GEMM, the strided batch GEMM routine is available in five precisions.

`cublasHgemmStridedBatched()`

```
cublasStatus_t
cublasHgemmStridedBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const __half *alpha, const __half *A, int lda, long long int strideA,
                         const __half *B, int ldb, long long int strideB,
    const __half *beta,        __half *C, int ldc, long long int strideC,
    int batchCount);
```

cublasSgemmStridedBatched()

```
cublasStatus_t
cublasSgemmStridedBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const float *alpha, const float *A, int lda, long long int strideA,
                        const float *B, int ldb, long long int strideB,
    const float *beta,        float *C, int ldc, long long int strideC,
    int batchCount);
```

cublasDgemmStridedBatched()

```
cublasStatus_t
cublasDgemmStridedBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const double *alpha, const double *A, int lda, long long int strideA,
                         const double *B, int ldb, long long int strideB,
    const double *beta,        double *C, int ldc, long long int strideC,
    int batchCount);
```

cublasCgemmStridedBatched()

```
cublasStatus_t
cublasCgemmStridedBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const cuComplex *alpha, const cuComplex *A, int lda, long long int strideA,
                            const cuComplex *B, int ldb, long long int strideB,
    const cuComplex *beta,        cuComplex *C, int ldc, long long int strideC,
    int batchCount);
```

cublasZgemmStridedBatched()

```
cublasStatus_t
cublasZgemmStridedBatched(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const cuDoubleComplex *alpha, const cuDoubleComplex *A, int lda, long long int strideA,
                                  const cuDoubleComplex *B, int ldb, long long int strideB,
    const cuDoubleComplex *beta,        cuDoubleComplex *C, int ldc, long long int strideC,
    int batchCount);
```

2.1.3 Routines from LAPACK with Pointer Array Interface

Surprisingly, cuBLAS provides more batch LAPACK routines than batch BLAS routines. The interface uses an array of pointers for each data argument. No stride interfaces are available for batch LAPACK routines. In addition, there is no interface for these routines that supports half-precision arithmetic. Here are some examples that are shown in double precision only.

Batch LU Factorization (GETRF)

cublasDgetrfBatched()

```
cublasStatus_t
cublasDgetrfBatched(
    cublasHandle_t handle,
    int n,
    double *Aarray[], int lda,
    int *PivotArray, int *infoArray,
    int batchSize);
```

Batch Linear Solve of LU Factorized Systems (GETRS)

cublasDgetrsBatched()

```
cublasStatus_t
cublasDgetrsBatched(
    cublasHandle_t handle,
    cublasOperation_t trans,
    int n, int nrhs,
    const double *Aarray[], int lda,
    const int *devIpiv,
          double *Barray[], int ldb,
    int *info,
    int batchSize);
```

Batch Inversion of LU Factorized Systems (GETRI)

cublasDgetriBatched()

```
cublasStatus_t
cublasDgetriBatched(
    cublasHandle_t handle,
    int n,
    double *Aarray[], int lda,
    int *PivotArray,
    double *Carray[], int ldc,
    int *infoArray,
    int batchSize);
```

Batch QR Factorization (GEQRF)

cublasDgeqrfBatched()

```
cublasStatus_t
cublasDgeqrfBatched(
    cublasHandle_t handle,
    int m, int n,
    double *Aarray[], int lda,
    double *TauArray[],
    int *info,
    int batchSize);
```

Batch Least Squares Solve (GELS)

cublasDgelsBatched()

```
cublasStatus_t
cublasDgelsBatched(
    cublasHandle_t handle,
    cublasOperation_t trans,
    int m, int n, int nrhs,
    double *Aarray[], int lda,
    double *Carray[], int ldc,
    int *info, int *devInfoArray,
    int batchSize);
```

2.1.4 Routines with LAPACK-like Functionality with a Pointer Array Interface

Some routines in cuBLAS have LAPACK-like functionalities like `cublas<t>matinvBatched`, which computes the explicit inverse of a batch of matrices that have the same size. However, this routine is limited to matrices up to 32×32 in size. The inverse of larger matrices can be computed by calling `cublas<t>getrfBatched` followed by `cublas<t>getriBatched`.

Batch Explicit Matrix Inversion (MATINV)

`cublasDmatinvBatched()`

```
cublasStatus_t
cublasDmatinvBatched(
    cublasHandle_t handle,
    int n,
    const double *A[],    int lda,
          double *Ainv[], int lda_inv,
    int *info,
    int batchSize);
```

2.1.5 Summary

In the table below, we summarize the available interface calls to illustrate the distribution of functionalities among various available routines. In an ideal scenario, each routine would be available in three precisions (16-bit, 32-bit, and 64-bit) for both real and complex elements. Every routine would come in one of three flavors: single large matrices (L), batch of small matrices (B), and a batch of strided matrices (S). As of CUDA 9, only a subset of this functionality is available. The table summarizes batch BLAS routines only.

Real/Complex →	$\mathbb{R}$	$\mathbb{C}$	$\mathbb{R}$	$\mathbb{C}$	$\mathbb{R}$	$\mathbb{C}$
Precision →	16	16	32	32	64	64
Routine ↓						
`gemm`	L/B		L/B/S	L/B/S	L/B/S	L/B/S
`trsm`			L/B	L/B	L/B	L/B

L = interface for large matrices
B = interface for batch of matrices
S = interface for batch of strided matrices

2.2 Intel® MKL

Intel's MKL is a highly optimized mathematical library for Intel processors—including multi-core CPUs and many-core coprocessors (Xeon Phi). As of this writing, Intel MKL 2018 includes batch routines for matrix multiplication (GEMM) and triangular solve (TRSM) for single (S), double (D), single complex (C), and double complex (Z) precisions. MKL also includes routines for specialized matrix multiplication (GEMM3M) for C and Z precisions only. The APIs for the batch routines are quite flexible in that they support both fixed and variable parameters for the operations by introducing the *group* designation. Within a group, the operations must have the same transpose (`trans`, `tranb`), sizes (m, n, k), α, β, and leading dimensions (`lda, ldb, ldc`). These parameters can be different for different groups. By grouping operations with the same aforementioned parameters and specifying multiple groups, a single API call can accommodate many operations without repeating identical parameters. The `layout` (row major or column major), however, must be the same for all the operations.

2.2.1 GEMM

```
void
cblas_sgemm_batch(
    const CBLAS_LAYOUT Layout,
    const CBLAS_TRANSPOSE *transa_array,
    const CBLAS_TRANSPOSE *transb_array,
    const MKL_INT *m_array, const MKL_INT *n_array, const MKL_INT *k_array,
    const float *alpha_array, const float **a_array, const MKL_INT *lda_array,
                              const float **b_array, const MKL_INT *ldb_array,
    const float* beta_array,        float **c_array, const MKL_INT* ldc_array,
    const MKL_INT group_count, const MKL_INT *group_size);

void
cblas_dgemm_batch(
    const CBLAS_LAYOUT Layout,
    const CBLAS_TRANSPOSE *transa_array, const CBLAS_TRANSPOSE *transb_array,
    const MKL_INT *m_array, const MKL_INT *n_array, const MKL_INT *k_array,
    const double *alpha_array, const double **a_array, const MKL_INT *lda_array,
                               const double **b_array, const MKL_INT *ldb_array,
    const double *beta_array,        double **c_array, const MKL_INT *ldc_array,
    const MKL_INT group_count, const MKL_INT *group_size);

void
cblas_cgemm_batch(
    const CBLAS_LAYOUT Layout,
    const CBLAS_TRANSPOSE *transa_array, const CBLAS_TRANSPOSE *transb_array,
    const MKL_INT *m_array, const MKL_INT *n_array, const MKL_INT *k_array,
    const void *alpha_array, const void **a_array, const MKL_INT *lda_array,
                             const void **b_array, const MKL_INT *ldb_array,
    const void *beta_array,        void **c_array, const MKL_INT *ldc_array,
    const MKL_INT group_count, const MKL_INT *group_size);

void
cblas_zgemm_batch(
    const CBLAS_LAYOUT Layout,
    const CBLAS_TRANSPOSE *transa_array, const CBLAS_TRANSPOSE *transb_array,
    const MKL_INT *m_array, const MKL_INT *n_array, const MKL_INT *k_array,
    const void *alpha_array, const void **a_array, const MKL_INT *lda_array,
                             const void **b_array, const MKL_INT *ldb_array,
    const void *beta_array,        void **c_array, const MKL_INT *ldc_array,
    const MKL_INT group_count, const MKL_INT *group_size);
```

2.2.2 TRSM

```
void
cblas_strsm_batch(
    const CBLAS_LAYOUT Layout,
    const CBLAS_SIDE *Side_Array, const CBLAS_UPLO *Uplo_Array,
    const CBLAS_TRANSPOSE *TransA_Array, const CBLAS_DIAG *Diag_Array,
    const MKL_INT *M_Array, const MKL_INT *N_Array,
    const float *alpha_Array, const float **A_Array, const MKL_INT *lda_Array,
                                    float **B_Array, const MKL_INT *ldb_Array,
    const MKL_INT group_count, const MKL_INT *group_size );

void
cblas_dtrsm_batch(
    const CBLAS_LAYOUT Layout,
    const CBLAS_SIDE *Side_Array, const CBLAS_UPLO *Uplo_Array,
    const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG *Diag_Array,
    const MKL_INT *M_Array, const MKL_INT *N_Array,
    const double *alpha_Array, const double **A_Array, const MKL_INT *lda_Array,
                                     double **B_Array, const MKL_INT *ldb_Array,
    const MKL_INT group_count, const MKL_INT *group_size );

void
cblas_ctrsm_batch(
    const CBLAS_LAYOUT Layout,
    const CBLAS_SIDE *Side_Array, const CBLAS_UPLO *Uplo_Array,
    const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG *Diag_Array,
    const MKL_INT *M_Array, const MKL_INT *N_Array,
    const void *alpha_Array, const void **A_Array, const MKL_INT *lda_Array,
                                   void **B_Array, const MKL_INT *ldb_Array,
    const MKL_INT group_count, const MKL_INT *group_size );

void
cblas_ztrsm_batch(
    const CBLAS_LAYOUT Layout,
    const CBLAS_SIDE *Side_Array, const CBLAS_UPLO *Uplo_Array,
    const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG *Diag_Array,
    const MKL_INT *M_Array, const MKL_INT *N_Array,
    const void *alpha_Array, const void **A_Array, const MKL_INT *lda_Array,
                                   void **B_Array, const MKL_INT *ldb_Array,
    const MKL_INT group_count, const MKL_INT *group_size );
```

2.3 AMD hipBLAS/rocBLAS

For their part, AMD provides the hipBLAS and rocBLAS libraries, both of which are being developed in public repositories hosted on GitHub. hipBLAS[3] is a BLAS marshaling library with support for different back ends, including support for the rocBLAS and cuBLAS back ends. hipBLAS also allows the user to change the back end BLAS library without changing the client code (the application). rocBLAS[4] is a BLAS library optimized for AMD's latest discrete GPUs, is implemented on top of AMD's Radeon Open Compute (ROCm) runtime and tool chains, and uses the HIP programming language.

At this point, hipBLAS contains a small number of functions—including a handful of level-1, level-2, and level-3 BLAS functions—and a set of basic functions for managing streams and matrices. In terms of batch BLAS, hipBLAS contains `hipblas?gemmBatched` and `hipblas?gemmStridedBatched` in single and double precision (no complex precision yet).

[3] https://github.com/ROCmSoftwarePlatform/hipBLAS
[4] https://github.com/ROCmSoftwarePlatform/rocBLAS

```
hipblasStatus_t
hipblasSgemmBatched(
    hipblasHandle_t handle,
    hipblasOperation_t transa, hipblasOperation_t transb,
    int m, int n, int k,
    const float *alpha, const float *A[], int lda,
                        const float *B[], int ldb,
    const float *beta,        float *C[], int ldc,
    int batchCount);

hipblasStatus_t
hipblasDgemmBatched(
    hipblasHandle_t handle,
    hipblasOperation_t transa, hipblasOperation_t transb,
    int m, int n, int k,
    const double *alpha, const double *A[], int lda,
                         const double *B[], int ldb,
    const double *beta,        double *C[], int ldc,
    int batchCount);
```

```
hipblasStatus_t
hipblasSgemmStridedBatched(
    hipblasHandle_t handle,
    hipblasOperation_t transa, hipblasOperation_t transb,
    int m, int n, int k,
    const float *alpha, const float *A, int lda, long long bsa,
                        const float *B, int ldb, long long bsb,
    const float *beta,        float *C, int ldc, long long bsc,
    int batchCount);

hipblasStatus_t
hipblasDgemmStridedBatched(
    hipblasHandle_t handle,
    hipblasOperation_t transa, hipblasOperation_t transb,
    int m, int n, int k,
    const double *alpha, const double *A, int lda, long long bsa,
                         const double *B, int ldb, long long bsb,
    const double *beta,        double *C, int ldc, long long bsc,
    int batchCount);
```

Currently, in terms of batch operations, rocBLAS only contains the strided batch ?gemm routine in single and double precisions.

```
rocblas_status
rocblas_sgemm_strided_batched(
    rocblas_handle handle,
    rocblas_operation transa, rocblas_operation transb,
    rocblas_int m, rocblas_int n, rocblas_int k,
    const float *alpha, const float *A, rocblas_int lda, rocblas_int bsa,
                        const float *B, rocblas_int ldb, rocblas_int bsb,
    const float *beta,        float *C, rocblas_int ldc, rocblas_int bsc,
    rocblas_int batch_count);

rocblas_status
rocblas_dgemm_strided_batched(
    rocblas_handle handle,
    rocblas_operation transa, rocblas_operation transb,
    rocblas_int m, rocblas_int n, rocblas_int k,
    const double *alpha, const double *A, rocblas_int lda, rocblas_int bsa,
                         const double *B, rocblas_int ldb, rocblas_int bsb,
    const double *beta,        double *C, rocblas_int ldc, rocblas_int bsc,
    rocblas_int batch_count);
```

hipBLAS can be used with cuBLAS as the back end. When doing so, hipBLAS functions are simple wrappers for the cuBLAS functions. Therefore, the semantics of the hipBLAS functions are

identical to their cuBLAS counterparts, `cublas?gemmBatched` and `cublas?gemmStridedBatched`.

2.4 ICL MAGMA

The MAGMA library[5] is an open-source package that harnesses a GPU's compute power in many BLAS and LAPACK algorithms. The original purpose of MAGMA was to take advantage of heterogeneous CPU-GPU architectures by offloading throughput-sensitive workloads (e.g., GEMM) to the GPU while performing latency-sensitive tasks (e.g., panel factorization) on the CPU [15]. Now, however, MAGMA also provides some routines that work entirely on the GPU. In fact, all batch routines in MAGMA (both BLAS and LAPACK) are designed for GPU-only execution. MAGMA is the only GPU-accelerated library that provides variable-size batch BLAS routines.

Unlike the Intel MKL library, MAGMA provides two separate APIs for each batch routine—one for fixed sizes and the other for variable sizes. All fixed-size routines have the suffix `_batched`, while the variable-size routines have the suffix `_vbatched`. For example, the fixed-size batch DGEMM routine in MAGMA is shown below.

```
void
magmablas_dgemm_batched(
    magma_trans_t transA, magma_trans_t transB,
    magma_int_t m, magma_int_t n, magma_int_t k,
    double alpha,
    double const * const * dA_array, magma_int_t ldda,
    double const * const * dB_array, magma_int_t lddb,
    double beta,
    double **dC_array, magma_int_t lddc,
    magma_int_t batchCount, magma_queue_t queue );
```

Similarly, the variable-size batch DGEMM routine looks like this:

```
void
magmablas_dgemm_vbatched(
    magma_trans_t transA, magma_trans_t transB,
    magma_int_t* m, magma_int_t* n, magma_int_t* k,
    double alpha,
    double const * const * dA_array, magma_int_t* ldda,
    double const * const * dB_array, magma_int_t* lddb,
    double beta,
    double **dC_array, magma_int_t* lddc,
    magma_int_t batchCount, magma_queue_t queue );
```

MAGMA assumes that all array arguments are stored in the GPU's main memory. MAGMA also provides batched one-sided factorization routines. For example, the Cholesky factorization algorithm is available for both fixed and variable-size batch workloads.

```
magma_int_t
magma_dpotrf_batched(
    magma_uplo_t uplo, magma_int_t n,
    double **dA_array, magma_int_t lda,
```

[5] http://icl.cs.utk.edu/magma/

```
    magma_int_t *info_array,
    magma_int_t batchCount, magma_queue_t queue);
```

```
magma_int_t
magma_dpotrf_vbatched(
    magma_uplo_t uplo, magma_int_t *n,
    double **dA_array, magma_int_t *ldda,
    magma_int_t *info_array,  magma_int_t batchCount,
    magma_queue_t queue);
```

2.4.1 Error Handling in MAGMA

The batch BLAS routines in MAGMA have a similar error handling mechanism to traditional BLAS. Numerical errors are not reported, and only errors in the arguments are detected and reported to the user through `XERBLA()`. The variable-size batch BLAS routines do not pinpoint the problem indices where argument errors are found. For example, if a size array has multiple negative entries, one error is reported for the entire array without indicating the positions of the negative values. In batch LAPACK routines, however, MAGMA distinguishes between argument errors and numerical errors through the following mechanisms.

1. Argument errors are checked before launching any computation on the batch. If argument errors are detected, the computation is canceled for the entire batch, and `XERBLA()` is invoked.
2. Numerical errors are specific to batch LAPACK routines and are reported through the `info_array` argument. If a numerical error is detected in a problem, the rest of the computation continues normally for other problems. The user must check the `info_array` argument for numerical errors after the routine is finished.

The argument error checking in fixed-size batch routines is accomplished using the host CPU, since the checking is performed on scalar arguments. However, the argument error checking for variable-size batch routines has to go through arrays that are stored in the GPU memory. This is why the error checking is done on the GPU. If errors are detected, the CPU is informed, and `XERBLA()` is invoked accordingly.

2.4.2 Advanced MAGMA APIs

MAGMA provides a set of advanced APIs for every variable-size batch routine. These APIs help mitigate two different types of overheads for the user.

1. **Checking argument errors.** The argument error checking in the variable-size routines can be a significant overhead, because it involves launching GPU kernels and communication with the CPU.
2. **GPU kernel configuration.** An implementation artifact of the current MAGMA routines requires computing the maximum size(s) of the input problems. Such maximum values are used to configure the computational kernels on the GPU.

The advanced APIs allow the user to skip these overheads if no checking is required (similar to Intel MKL direct calls[6]) and if the maximum sizes are known beforehand. For example, the `magmablas_dgemm_vbatched` has a lower-level API that looks like this:

```
void
magmablas_dgemm_vbatched_max_nocheck(
    magma_trans_t transA, magma_trans_t transB,
    magma_int_t* m, magma_int_t* n, magma_int_t* k,
    double alpha,
    double const * const * dA_array, magma_int_t* ldda,
    double const * const * dB_array, magma_int_t* lddb,
    double beta,
    double **dC_array, magma_int_t* lddc,
    magma_int_t max_m, magma_int_t max_n, magma_int_t max_k,
    magma_int_t batchCount, magma_queue_t queue );
```

The advanced API is usually recommended if the batch contains a relatively small number of problems and/or if the sizes are very small. This helps avoid the overheads of error checking and searching for the maximum values.

2.4.3 Discussion and Critique

Below are some discussion points and critiques of the MAGMA APIs.

1. **Same option arguments.** The MAGMA batch routines restrict the option arguments to be the same across all matrices in the batch. For example, the batch GEMM routines (for both fixed and variable sizes) accept only scalar arguments for transpositions. Similarly, the batch Cholesky factorization routines assume that either the lower or the upper triangular part is read/written for all matrices. While this seems like a lack of flexibility, scalar option arguments work well for MAGMA's purposes (batch BLAS and LAPACK functionality).

2. **Same scaling parameters.** Similar to the option arguments, the scaling parameters are also unified across the batch (e.g., `alpha` and `beta` in the batch GEMM routines). Again, this works well for the purposes of MAGMA but not necessarily for other workloads. However, the API's design in this regard is compatible with the current cuBLAS batch routines and allows for easy integration of cuBLAS batch functionality into MAGMA.

3. **Separate APIs.** MAGMA maintains at least three APIs for every batch routine (fixed size, variable size, and advanced variable size). Keeping too many interfaces creates long-term software maintenance problems.

4. **Cumbersome use of variable size APIs.** In some use cases, the variable size API imposes some difficulties for the users. For example, an array argument with a unified value across the entire batch must still be passed as an array for which entries possess the same value. Having to create an array with identical entries provides a poor user experience.

[6] https://software.intel.com/en-us/articles/improve-intel-mkl-performance-for-small-problems-the-use-of-mkl-direct-call

2.5 Summary of Existing APIs

This section summarizes the main differences among the existing batch BLAS interfaces. In general, BLAS routines have four categories of arguments. Taking matrix multiplication (GEMM) as an example, these categories are:

1. **Option arguments**. These arguments specify different options for performing the BLAS operation. They can be used to define the pattern of accessing the data (e.g., lower vs. upper triangular), the order of performing the computation (e.g., forward vs. backward substitution), and others. The transposition arguments in the GEMM routine (`transA` and `transB`) correspond to this kind of argument.
2. **Scaling arguments**. These arguments specify scalar values that can be used to scale the values of a matrix or a vector. The GEMM routine uses two scaling arguments: (1) `alpha` to scale the product $A \times B$ and (2) `beta` to pre-scale the original values of the output matrix C.
3. **Size arguments**. This category corresponds to the integer arguments that specify the matrix/vector sizes, leading dimensions of matrices, strided access for vectors, etc. The GEMM routine includes many size arguments (`m`, `n`, `k`, `lda`, `ldb`, and `ldc`).
4. **Data pointer arguments**. This category is used to locate the data in memory, usually through a pointer. The `A`, `B`, and `C` arguments in the GEMM routine belong to this category.

Moving from BLAS to batch BLAS, many vendors and library developers took different approaches on which categories of arguments are unified across a batch of problems and which are allowed to be varied. While most solutions assume "flat" interfaces, in the sense that all problems belong to one batch, the Intel MKL library represents the only solution that uses a group interface, where the batch is subdivided into groups such that each group has its own set of arguments. With respect to all the existing solutions discussed in this document, Table 2.1 summarizes the decisions taken to develop each solution.

Type	Library name		Argument type			
			Options	Scaling	Sizes	Data pointers
Flag-based flat	Standard	`BATCH_FIXED`	F	F	F	V
		`BATCH_VARIABLE`	V	V	V	V
Group	MKL	Per group	F	F	F	V
		Across groups	V	V	V	V
Flat	MAGMA	Fixed size	F	F	F	V
		Variable size	F	F	V	V
	cuBLAS		F	F	F	V[7]
	hipBLAS/rocBLAS		F	F	F	V[7]

Table 2.1: A summary of existing interfaces for batch BLAS. An entry with "F" means that the corresponding category of arguments is fixed across the batch. An entry with "V" means that this category of arguments is allowed to be varied across different problems in the batch.

[7] A stride-based interface is also available.

All solutions agree that all data pointer arguments should be passed as pointer arrays. The standard C interface [6] is a flat interface that uses a flag to distinguish between the fixed-size and the variable-size batch workloads. If the value of the flag is `BATCH_FIXED`, then each non-data argument is allowed to have one unique value across the batch. If the flag is set to `BATCH_VARIABLE`, then all arguments can have different values for different problems. The Intel MKL library adopts a group interface, where all non-data arguments are fixed within a group but can be varied across groups. The MAGMA library provides a fixed size interface that is similar to a the standard C interface using the `BATCH_FIXED` mode. It also provides a variable size interface that allows the argument size to be varied but restricts the option arguments and the scaling arguments as fixed. Finally, both the cuBLAS and the rocBLAS libraries adopt interfaces that are similar to the MAGMA fixed size interface. They also provide stride-based interfaces, where a data pointer argument is passed as a combination (pointer+stride) rather than explicitly as a pointer array. The next chapter discusses the shortcomings associated with these interfaces and how they can be resolved with a flexible C++ interface.

CHAPTER 3

Proposed APIs

3.1 The Objective

The main objective of the C++ API is to address some of the shortcomings in the existing APIs. Every solution discussed in Chapter 2 imposes some constraints on certain categories of arguments. For example, cuBLAS, rocBLAS, and MAGMA all assume that the scaling arguments are fixed across the batch. The same restriction applies to the leading dimensions. These are unnecessary constraints. In general, a batch can have matrices of the same size but with different scaling arguments and leading dimensions. Similar constraints appear in the MKL interface (within a group) and in the standard C interface when the `BATCH_FIXED` mode is enabled.

On another level, the standard C interface is the only flat API that allows all argument categories to be varied across the batch. The MKL interface supports the same variation across different groups. While these interfaces have a high degree of flexibility, some scenarios are poorly supported, which results in the user having to duplicate one or more arguments. For example, consider the batch rank-k update in the LU factorization ($C_{m\times n} = C_{m\times n} - A_{m\times k} \times B_{k\times n}$) to be applied on matrices of different sizes $\{C1_{m_1\times n_1}, C2_{m_2\times n_2}$, etc.$\}$. Assuming that $k \ll n_i$, the value of k is unified across all matrices during the batch update. Unfortunately, such a scenario is poorly supported through the existing interfaces. The user must invoke a variable size API, where k is unnecessarily promoted to an array of duplicate values. This applies to all variable-size flat interfaces. For the MKL group interface, different size matrices have to go to different groups, and an array of duplicate values for replacing the scalar k is still required.

The same unnecessary duplication can appear in fixed size interfaces. For example, consider a fixed-size batch GEMM, where the matrix A is the same for all problems. Since most existing solutions assume a pointer array for each data argument, the pointer of A has to be duplicated

in a pointer array, because there is no way to tell the batch routine that it is the same matrix.

The objective of the C++ interface is to flexibly support many different scenarios through one unified API. Such an API should meet the following requirements.

1. Support both fixed-size and variable-size batches (no separate APIs).
2. No unnecessary duplication of any argument category. Basically, any argument can be independently fixed or varied across the batch.
3. Feasible implementation on CPUs and accelerators.
4. Flexible error checking mechanisms.
5. Support for different modes of operations (e.g., flat, grouped, and stride-based).

An API that meets these requirements will allow easy integration into several applications and ensure portable codes on different hardware configurations.

3.2 Compliance with the C++ BLAS API

The proposed C++ API for batch BLAS will be introduced in the context of the C++ API for BLAS and LAPACK [8], which represents the foundation of the SLATE library [12]. The proposed API shares the following design decisions from the C++ BLAS API. For more details about these design decisions and the justifications supporting them, the reader should refer to the original design document for the BLAS/LAPACK C++ API [8].

1. **Stateless interface**. The proposed batch BLAS API will be stateless.
2. **Templated routines**. The proposed APIs use templated routines with respect to precision, which allows for a unified name across different precisions and future mixed precision routines. Similar to BLAS [8], the templated routines will have overloaded signatures to support the existing solutions from vendors and library developers.
3. **C++ language standard**. The C++11 standard sufficiently covers all of the features required in the proposed APIs.
4. **Naming convention**. The batch BLAS routines will have names similar to those used in the traditional BLAS routines, excluding the precision prefix. The interfaces shall be declared in the `batch` namespace, which is in turn declared inside the `blas` namespace. Therefore, both BLAS and batch BLAS interfaces are reachable by including the `blas.hh` header and using the namespace `blas`. As an example, the name of the batch GEMM routine will be `blas::batch::gemm`.
5. **Unified syntax**. Some routines have different names for real and complex precisions (e.g., `syrk` vs. `herk`). Following the footsteps of the C++ BLAS API, both names will be extended to all precisions.

6. **const**. The `const` specifier will be used where applicable.

7. **Enumberated constants**. The proposed APIs use the same *enum* constants defined in the `blas` namespace. These constants mostly correspond to the option arguments in BLAS operations (e.g., transposition, upper/lower triangular).

8. **Errors**. The proposed API uses C++ exceptions to report errors. However, there are different modes of operation for error reporting. More details are presented in Section 3.4.

9. **Return values**. All batch BLAS interfaces will be void. Although some BLAS routines have return values (e.g., `nrm2`), the corresponding batch routines will have an extra output argument to hold the results instead.

10. **Integer type**. The proposed API will use the `int64_t` type to specify sizes and dimensions. Although batch routines usually operate on relatively small sizes, the use of `int64_t` unifies the object dimensions between BLAS and batch BLAS and avoids any confusion about the size of integer type.

11. **Matrix layout**. For now, the proposed API shall focus only on column-major layouts. While the C++ BLAS API supports row-major layouts by calling column-major routines using different options (e.g., changing transpositions), the same approach has technical issues for some routines if applied to the batch BLAS—especially when explicit transpositions are required in extra allocated workspaces.

3.3 General Design Principles

The requirements mentioned in Section 3.1 state that the API should be flexible enough to accomodate several scenarios (e.g., fixed size vs. variable size and fixed arguments vs. array arguments). Basically, there are two ways to accomplish such a requirement. The first is to use overloaded routine names such that a single routine name accepts different sets of arguments. While this is a legitimate approach, it is very cumbersome for vendors and library developers to provide many signatures for the same routine. For example, the GEMM routine accepts 13 arguments. Excluding the output argument and ruling out some ineligible combinations, the batch GEMM routine can have more than a thousand sets of different signatures. The overloaded routine names are, therefore, impractical as a primary solution.

The second approach, which is the solution proposed in this document, is to use the `std::vector` container in C++. Basically, almost every argument in a BLAS routine will be promoted to a `std::vector` argument of the same type. As an example, the code below shows the declarations of the GEMM and the batch GEMM routines within the `blas` and `batch` namespaces, respectively.

```
1  using namespace std;
2
3  namespace blas {
4
5  /* other declarations */
6
7  template<typename FloatType>
8  void gemm( Op transA, Op transB,
9             int64_t m, int64_t n, int64_t k,
10            FloatType alpha, FloatType* A, int64_t lda,
```

```
                            FloatType* B, int64_t ldb,
            FloatType beta, FloatType* C, int64_t ldc
        );

namespace batch {

/* other declarations */

template<typename FloatType>
void gemm(
        vector<Op> const &transA, vector<Op> const &transB,
        vector<int64_t> const &m, vector<int64_t> const &n, vector<int64_t> const &k,
        vector<FloatType>  const &alpha,
        vector<FloatType*> const &A, vector<int64_t> const &lda,
        vector<FloatType*> const &B, vector<int64_t> const &ldb,
        vector<FloatType>  const &beta,
        vector<FloatType*> const &C, vector<int64_t> const &ldc,
        const int64_t batchCount);

} // namespace batch
} // namespace blas
```

As mentioned before, the namespace `batch` is included within the `blas` namespace, so that `blas::gemm` is the traditional non-batch instance of the operation, and `blas::batch::gemm` corresponds to its batch variant. The interface shown does not provide error checking, which will be discussed later in Section 3.4. The basic idea is that a vector argument should have its size equal to either one or `batchCount`. If the size of the vector is one, then the value of the corresponding argument is unified across the batch. If the size is equal to `batchCount`, then the corresponding argument has a value for each problem. This does not include the vector(s) of the output argument(s) (e.g., `C` in the batch GEMM routine), which must obviously be of size `batchCount`. While the `batchCount` argument can be removed completely, since the size of the batch can be deduced from the size of the output vector, we decided to preserve it for potential extensions, as explained later in Sections 3.7 and 3.8.

Below are some advantages of this interface.

1. The batch GEMM interface shown above covers over a thousand different combinations of fixed vs. varied arguments. It eliminates the borders between fixed size and variable size batch routines, as it naturally combines both. It also goes beyond the fixed vs. variable size distinction and efficiently supports significantly more calling scenarios than any other interface.

2. No duplication of any argument is required. If the value of an argument is unified across the batch, then it is passed in a `std::vector` of size one.

3. The interface uses a standard C++ container, which is widely supported in almost all software environments. It also comes with built-in functions and capabilities such as iterators, maximum/minimum functions, and others. The `vector` type is also supported on accelerators (e.g., NVIDIA's Thrust library[1]).

4. A reference implementation is feasible on both CPUs and accelerators. Vendors and library developers can choose to provide more optimized routines for specific combinations of vector sizes. Such optimized routine(s) can be internally invoked instead of the reference

[1] http://docs.nvidia.com/cuda/thrust/index.html

implementation if the vector sizes match the corresponding combination of the optimized routine.

The reference implementation using this interface can be developed through OpenMP on CPUs, and through execution queues on GPU accelerators (e.g., CUDA streams). We first introduce a very simple *extract* function that, given the index of a problem, extracts the corresponding value of an argument. Because the function is templated, it works for any type of argument (e.g., option arguments, scaling arguments, sizes, and pointers). The function exists in the `blas::batch` namespace.

```
1
2  template<typename T>
3  T blas::batch::extract(vector<T> const &ivector, const int64_t index)
4  {
5      if(ivector.size() == 1)
6          return ivector[0];
7      else
8          return ivector[index];
9  }
```

A reference implementation for the batch GEMM using OpenMP is shown below. Recall that, at this stage, we assume no error checking. Ideally, the routine `blas::gemm` should provide an instance that does not throw any exceptions. The reference implementation is a simple `parallel for` loop that calls a non-batch GEMM routine. Before the invokation takes place, the `extract` function is called as many times as necessary to provide the list of arguments for a specific operation.

```
1  using namespace std;
2  using namespace blas;
3
4  template<typename FloatType>
5  void blas::batch::gemm(
6          vector<Op> const &transA, vector<Op> const &transB,
7          vector<int64_t> const &m, vector<int64_t> const &n, vector<int64_t> const &k,
8          vector<FloatType>  const &alpha,
9          vector<FloatType*> const &A, vector<int64_t> const &lda,
10         vector<FloatType*> const &B, vector<int64_t> const &ldb,
11         vector<FloatType>  const &beta,
12         vector<FloatType*> const &C, vector<int64_t> const &ldc,
13         const int64_t batchCount)
14 {
15     #pragma omp parallel for schedule(dynamic)
16     for(int64_t i = 0; i < batchCount; i++){
17         // extract arguments
18         Op transA_ = extract<Op>(transA, i);
19         Op transB_ = extract<Op>(transB, i);
20
21         int64_t m_ = extract<int64_t>(m, i);
22         int64_t n_ = extract<int64_t>(n, i);
23         int64_t k_ = extract<int64_t>(k, i);
24
25         FloatType alpha_ = extract<FloatType>(alpha, i);
26         FloatType  beta_ = extract<FloatType>(beta,  i);
27
28         FloatType* A_ = extract<FloatType*>(A,  i);
29         FloatType* B_ = extract<FloatType*>(B,  i);
30         FloatType* C_ = C[i];
31
32         int64_t lda_ = extract<int64_t>(lda, i);
33         int64_t ldb_ = extract<int64_t>(ldb, i);
34         int64_t ldc_ = extract<int64_t>(ldc, i);
```

```
35
36          // call non-batch gemm
37          blas::gemm<FloatType>(
38                  transA_, transB_,
39                  m_, n_, k_,
40                  alpha_, A_, lda_,
41                          B_, ldb_,
42                  beta_ , C_, ldc_);
43      }
44  }
```

3.4 BLAS Error Checking

The error checking in BLAS is responsible for catching errors in the arguments' values (e.g., if the leading dimension of a matrix is less than its number of rows). In batch BLAS, we propose to have a similar error checking behavior, except that the user can specify different methods of error reporting. We also choose to cancel the entire computation on the batch if any error is detected in any problem. The batch GEMM interface with error checking looks like:

```
1  namespace blas{
2
3  /* other declarations */
4
5  namespace batch{
6
7  /* other declarations */
8
9  template<typename FloatType>
10 void gemm(
11         vector<Op> const &transA, vector<Op> const &transB,
12         vector<int64_t> const &m, vector<int64_t> const &n, vector<int64_t> const &k,
13         vector<FloatType>  const &alpha,
14         vector<FloatType*> const &A, vector<int64_t> const &lda,
15         vector<FloatType*> const &B, vector<int64_t> const &ldb,
16         vector<FloatType>  const &beta,
17         vector<FloatType*> const &C, vector<int64_t> const &ldc,
18         const int64_t batchCount, vector<int64_t> const &info
19         );
20 } // namespace batch
21 } // namespace blas
```

The interface is now overloaded to accept an extra vector argument, `info`. The size of the `info` vector specifies how the errors are reported, the general procedure of which is outlined below.

1. If the vector size is zero, this means "disable any error checking." In fact, this is mostly equivalent to calling the interface that does not accept the `info` argument.

2. If the vector size is one, this enables the argument-based error checking (ABER) mode. This mode is similar to the error checking in BLAS except that it executes on vector arguments, and it does not pinpoint the indices of the entries that caused the errors. If an error is detected in a vector argument, the value of `info` is set to `-i`, where `i` is the order of the vector argument in the interface. For example, if any error is detected in the `lda` argument, the value of `info` is set to -8. An exception is thrown to cancel the computation and notify the user. This mode informs the user that "something is wrong with a certain argument." The exception is thrown when the first error is detected.

3. If the vector size is batchCount, this enables the problem-based error reporting (PBER) mode, where an error code is generated for every problem in the batch. This mode provides more information about each problem. If an operation encountered an error, the respective entry in info is set to a negative value that indicates the error-causing argument. An exception is thrown upon the detection of any error in any problem. Such an exception informs the user that "something went wrong in at least one problem." The user should then inspect the info vector to determine which problems encountered errors.

The following code shows a prototype implementation for the function gemm_check.

```
1
2  template<typename FloatType>
3  void blas::batch::gemm_check(
4          vector<Op> const &transA, vector<Op> const &transB,
5          vector<int64_t> const &m, vector<int64_t> const &n, vector<int64_t> const &k,
6          vector<FloatType>  const &alpha,
7          vector<FloatType*> const &A, vector<int64_t> const &lda,
8          vector<FloatType*> const &B, vector<int64_t> const &ldb,
9          vector<FloatType>  const &beta,
10         vector<FloatType*> const &C, vector<int64_t> const &ldc,
11         const int64_t batchCount, vector<int64_t> const &info)
12 {
13     if(info.size == 1){
14         /* argument based error reporting */
15         int64_t linfo;
16
17         // transA
18         linfo = 0;
19         #pragma omp parallel for reduction(+:linfo)
20         for(int64_t i = 0; i < transA.size(); i++){
21             linfo += (transA[i] != Op::NoTrans &&
22                       transA[i] != Op::Trans   &&
23                       transA[i] != Op::ConjTrans
24                       ) ? 1 : 0;
25         }
26         info[0] = (linfo > 0) ? -1 : 0;
27         throw_if_( info[0] == -1 );
28
29         // transB
30         linfo = 0;
31         #pragma omp parallel for reduction(+:linfo)
32         for(int64_t i = 0; i < transB.size(); i++){
33             linfo += (transB[i] != Op::NoTrans &&
34                       transB[i] != Op::Trans   &&
35                       transB[i] != Op::ConjTrans
36                      ) ? 1 : 0;
37         }
38         info[0] = (linfo > 0) ? -2 : 0;
39         throw_if_( info[0] == -2 );
40
41         // m
42         linfo = 0;
43         #pragma omp parallel for reduction(+:linfo)
44         for(int64_t i = 0; i < m.size(); i++){
45             linfo += (m[i] < 0) ? 1 : 0;
46         }
47         info[0] = (linfo > 0) ? -3 : 0;
48         throw_if_( info[0] == -3 );
49
50         // n
51         linfo = 0;
52         #pragma omp parallel for reduction(+:linfo)
53         for(int64_t i = 0; i < n.size(); i++){
54             linfo += (n[i] < 0) ? 1 : 0;
```

```
55          }
56          info[0] = (linfo > 0) ? -4 : 0;
57          throw_if_( info[0] == -4 );
58
59          // k
60          linfo = 0;
61          #pragma omp parallel for reduction(+:linfo)
62          for(int64_t i = 0; i < k.size(); i++){
63              linfo += (k[i] < 0) ? 1 : 0;
64          }
65          info[0] = (linfo > 0) ? -5 : 0;
66          throw_if_( info[0] == -5 );
67
68          // lda
69          linfo = 0;
70          #pragma omp parallel for reduction(+:linfo)
71          for(int64_t i = 0; i < batchCount; i++){
72              Op trans_  = extract<Op>(transA, i);
73              int64_t nrowA_ = (trans_ == Op::NoTrans) ?
74                               extract<int64_t>(m, i) : extract<int64_t>(k, i);
75              int64_t lda_   = extract<int64_t>(lda, i);
76              linfo += (lda_ < nrowA_) ? 1 : 0;
77          }
78          info[0] = (linfo > 0) ? -8 : 0;
79          throw_if_( info[0] == -8 );
80
81          // ldb
82          linfo = 0;
83          #pragma omp parallel for reduction(+:linfo)
84          for(int64_t i = 0; i < batchCount; i++){
85              Op trans_  = extract<Op>(transB, i);
86              int64_t nrowB_ = (trans_ == Op::NoTrans) ?
87                               extract<int64_t>(k, i) : extract<int64_t>(n, i);
88              int64_t ldb_   = extract<int64_t>(ldb, i);
89              linfo += (ldb_ < nrowB_) ? 1 : 0;
90          }
91          info[0] = (linfo > 0) ? -10 : 0;
92          throw_if_( info[0] == -10 );
93
94          // ldc
95          linfo = 0;
96          #pragma omp parallel for reduction(+:linfo)
97          for(int64_t i = 0; i < batchCount; i++){
98              int64_t m_   = extract<int64_t>(m, i);
99              int64_t ldc_ = extract<int64_t>(ldc, i);
100             linfo += (ldc_ < m_) ? 1 : 0;
101         }
102         info[0] = (linfo > 0) ? -13 : 0;
103         throw_if_( info[0] == -13 );
104
105     }
106     else{
107         /* problem based eror reporting */
108         #pragma omp parallel for schedule(dynamic)
109         for(int64_t i = 0; i < batchCount; i++){
110             Op transA_ = extract<Op>(transA, i);
111             Op transB_ = extract<Op>(transB, i);
112
113             int64_t m_ = extract<int64_t>(m, i);
114             int64_t n_ = extract<int64_t>(n, i);
115             int64_t k_ = extract<int64_t>(k, i);
116
117             int64_t lda_ = extract<int64_t>(lda, i);
118             int64_t ldb_ = extract<int64_t>(ldb, i);
119             int64_t ldc_ = extract<int64_t>(ldc, i);
120
```

```
121             int64_t norwA_ = (tarnsA_ == Op::NoTrans) ? m_ : k_;
122             int64_t norwB_ = (tarnsB_ == Op::NoTrans) ? k_ : n_;
123
124             if(transA_ != Op::NoTrans &&
125                transA_ != Op::Trans   &&
126                transA_ != Op::ConjTrans) {
127                 info[i] = -1;
128             }
129             else if(transB_ != Op::NoTrans &&
130                     transB_ != Op::Trans   &&
131                     transB_ != Op::ConjTrans) {
132                 info[i] = -2;
133             }
134             else if(m_ < 0) info[i] = -3;
135             else if(n_ < 0) info[i] = -4;
136             else if(k_ < 0) info[i] = -5;
137             else if(lda_ < nrowA_) info[i] = -8;
138             else if(ldb_ < nrowB_) info[i] = -10;
139             else if(ldc_ < m_    ) info[i] = -13;
140         }
141
142         int64_t info_ = 0;
143         #pragma omp parallel for reduction(+:info_)
144         for(int64_t i = 0; i < batchCount; i++){
145             info_ += info[i];
146         }
147         throw_if_( info_ != 0 );
148     }
149 }
```

3.5 Size Error Checking

The use of the `std::vector` container enforces another kind of checking related to vector sizes. With few exceptions, most vector arguments must have sizes that are either 1 or `batchCount`. If any vector argument has a size that is not allowed, an exception is thrown to notify the user about the error. This is a completely new type of error that did not exist before in BLAS. In addition, there are some cases where the vector sizes become inconsistent across a group of arguments. For example, if the matrix A in a batch GEMM operation is the same for all multiplications (i.e., `A.size()` returns 1), this means that the sizes of `m`, `k`, and `lda` should all be equal to 1. Otherwise, an exception is thrown to notify the user about the error in the vector sizes. The size error checking layer precedes the BLAS error checking layer.

3.6 Complete Prototype for a Reference Implementation

Now we bring all the pieces together to provide a prototype reference implementation for a batch GEMM. Figure 3.1 shows the composition of the different layers. The figure shows a sugessted hierarchy for a reference implementation that invokes a non-batch BLAS routine. Routines that are natively optimized for batch workloads do not have to follow the same composition. However, the behavior of error reporting should adhere to the specifications mentioned in Sections 3.4 and 3.5.

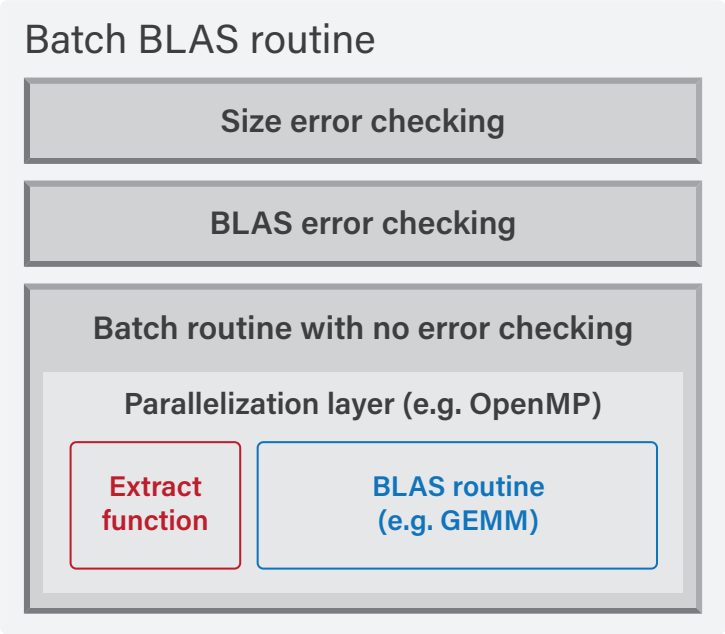

Figure 3.1: The hierarchy of the proposed reference implementation of a batch BLAS routine.

The code for the prototype reference implementation is shown below.

```
using namespace std;
using namespace blas;

template<typename FloatType>
void blas::batch::gemm(
        vector<Op> const &transA, vector<Op> const &transB,
        vector<int64_t> const &m, vector<int64_t> const &n, vector<int64_t> const &k,
        vector<FloatType> const &alpha,
        vector<FloatType*> const &A, vector<int64_t> const &lda,
        vector<FloatType*> const &B, vector<int64_t> const &ldb,
        vector<FloatType> const &beta,
        vector<FloatType*> const &C, vector<int64_t> const &ldc,
        const int64_t batchCount, vector<int64_t> const &info
        )
{
    throw_if_( batchCount < 0 );
    throw_if_( !(info.size() == 0 || info.size() == 1 || info.size() == batchCount) );
    if(info.size() > 0){
        // size error checking
        throw_if_( (transA.size() != 1 && transA.size() != batchCount) );
        throw_if_( (transB.size() != 1 && transB.size() != batchCount) );

        throw_if_( (m.size() != 1 && m.size() != batchCount) );
        throw_if_( (n.size() != 1 && n.size() != batchCount) );
        throw_if_( (k.size() != 1 && k.size() != batchCount) );

        throw_if_( (alpha.size() != 1 && alpha.size() != batchCount) );
        throw_if_( (beta.size()  != 1 && beta.size()  != batchCount) );

        throw_if_( (lda.size() != 1 && lda.size() != batchCount) );
        throw_if_( (ldb.size() != 1 && ldb.size() != batchCount) );
        throw_if_( (ldc.size() != 1 && ldc.size() != batchCount) );

        throw_if_( (A.size() != 1 && A.size() != batchCount) );
        throw_if_( (B.size() != 1 && B.size() != batchCount) );
```

```
36          throw_if_( (C.size() != batchCount) );
37
38          throw_if_( A.size() == 1 && (m.size() > 1 || k.size() > 1 || lda.size() > 1) );
39          throw_if_( B.size() == 1 && (k.size() > 1 || n.size() > 1 || ldb.size() > 1) );
40          throw_if_( C.size() == 1 &&
41                     (transA.size() > 1 || transB.size() > 1 ||
42                      m.size()      > 1 || n.size()      > 1 || k.size()   > 1 ||
43                      alpha.size()  > 1 || beta.size()   > 1 ||
44                      lda.size()    > 1 || ldb.size()    > 1 || ldc.size() > 1 ||
45                      A.size()      > 1 || B.size()      > 1
46                     )
47                   );
48
49          // blas error checking
50          blas::batch::gemm_check<FloatType>( transA, transB,
51                                              m, n, k,
52                                              alpha, A, lda,
53                                                     B, ldb,
54                                              beta,  C, ldc,
55                                              batchCount, info);
56      }
57
58      blas::batch::gemm<FloatType>( transA, transB,
59                                    m, n, k,
60                                    alpha, A, lda,
61                                           B, ldb,
62                                    beta,  C, ldc);
63  }
```

So far we have focused on a flat interface that assumes no groups. The following sections explain how group-based and stride-based interfaces can be supported.

3.7 Extension for Group-Based APIs

The proposed interface is easily extendable to support groups of matrices—where all of the argument values are the same across a group—by promoting the `batchCount` argument from a constant integer to `std::vector<int64_t>`. In this case, the routine behaves as a flat routine only if the size of the vector `batchCount` is 1. Otherwise, a group-based routine is provided. The size error checking layer in a group implementation should limit the size of a vector argument to either 1 or the size of the `batchCount` vector. A variant of the `extract` function should also be provided in order to extract the correct argument value based on the group sizes.

The extension to a group interface is questionable, though. If the group interface is originally intended to achieve a higher performance than a flat interface on groups of fixed arguments, then the proposed API for batch BLAS can be extended to support groups and maintain the same performance advantage. However, if the group interface is originally intended only to provide a unified C interface that supports fixed-size and variable-size batches, then it is not necessary to extend the proposed API to support groups. The proposed C++ API naturally supports significantly more scenarios than the existing group interface in this regard.

3.8 Extension for Stride-Based APIs

Both cuBLAS and rocBLAS enable the user to implicitly pass a pointer array through a pointer and a fixed stride. If the matrices are equidistant from each other, this is simpler than explicitly providing a pointer array. There are two solutions for supporting stride-based interfaces. The first is to provide an auxiliary function to populate a `std::vector<FloatType*>` based on a pointer and a stride. Otherwise, the proposed C++ API can be overloaded to accept a pointer and a stride instead of `std::vector<FloatType*>`. However, it is unclear if both input and output data pointers should be strided. This adds yet another level of complexity to the interface. For example, considering batch GEMM, it is unclear if there is a scenario where one matrix argument is strided, while the others are not.

3.9 Discussion and Critique

The proposed API covers many scenarios that are impractical to cover in other languages like C and FORTRAN. The use of a standard container in C++ allows for easy adoption in many software frameworks. However, the high degree of flexibility is not without some shortcomings. Such shortcomings, along with suggested remedies, are summarized below.

1. The ease of use provided by the API comes at the cost of complications for vendors and library developers to provide high-performance implementations of the same routine for different calling scenarios. Fortunately, there is a relatively simple reference implementation that can cover all cases. Vendors and library developers should focus on the optimization of specific scenarios that appear frequently in today's applications. The optimization of other scenarios can be carried out over time based on demand from application developers.

2. Indeed, the implementation complexity is hidden from the user by a simple unified interface. However, the user has to be very careful about the sizes of all the vector arguments. The size error checking layer is responsible of informing the user about errors and inconsistencies in the vector sizes. This can be enabled by using an `info` vector of non-zero length.

3. Since a lot of semantics are hidden within the vector sizes, calling a batch routine might not be as self-explanatory as regular BLAS calls. This is the price paid to have such a flexible API. Since the user is responsible for providing correct vector sizes, it is relatively easy to understand the call by inspecting the argument setup before launching the batch routine.

4. The error checking layer, for both size and BLAS errors, is synchronous. No computation is allowed if any error is detected. This brings with it a considerable overhead for batch routines. An asynchronous error checking layer can amortize such an overhead, as the checking of some problems can overlap with the computation of other problems. However, the asynchronicity of error checking will probably only allow for PBER mode. In addition, if the batch is large, it makes more sense to check all arguments first—especially if the

error is detected only at the end of the batch. Finally, best practice would be to enable error checking only for development runs. Production runs should, ideally, disable error checking to maximize the performance, which can be accomplished with a simple change in the application developer's code (i.e., setting the `info` vector size to 0).

3.10 Summary

This chapter outlined the details of the proposed C++ API for batch BLAS routines. The API uses the standard vector C++ container (`std::vector`), which allows for a very flexible interface that can cover numerous calling scenarios. A detailed reference prototype implementation has been introduced for batch matrix multiplication. While a comprehensive, high-performance batch routine is a complicated thing to implement, such complexity can be mitigated over time by focusing on the current demands for batch workloads from application developers. As new demands arise for new patterns of batch workloads, the internals of the batch routine can be optimized specifically for such workloads while keeping the same unified interface.

Bibliography

[1] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack J. Dongarra, Christopher W. Earl, Joel Falcou, Azzam Haidar, Ian Karlin, Tzanio V. Kolev, Ian Masliah, and Stanimire Tomov. High-Performance Tensor Contractions for GPUs. In *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, pages 108–118, 2016. doi: 10.1016/j.procs.2016.05.302. URL https://doi.org/10.1016/j.procs.2016.05.302.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures. In *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, pages 606–615, 2017. doi: 10.1016/j.procs.2017.05.250. URL https://doi.org/10.1016/j.procs.2017.05.250.

[3] Kadir Akbudak, Hatem Ltaief, Aleksandr Mikhalev, and David Keyes. *Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures*, pages 22–40. Springer International Publishing, Cham, 2017. ISBN 978-3-319-58667-0. doi: 10.1007/978-3-319-58667-0_2. URL https://doi.org/10.1007/978-3-319-58667-0_2.

[4] Hartwig Anzt, Jack J. Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2017, Austin, TX, USA, February 5, 2017*, pages 1–10, 2017. doi: 10.1145/3026937.3026940. URL http://doi.acm.org/10.1145/3026937.3026940.

[5] Wajih Halim Boukaram, George Turkiyyah, Hatem Ltaief, and David E. Keyes. Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression. *Parallel Computing*, 2017. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2017.09.001. URL http://www.sciencedirect.com/science/article/pii/S0167819117301461.

[6] 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. Technical report, Manchester Institute for Mathematical Sciences, April 2016. URL http://eprints.maths.manchester.ac.uk/id/eprint/2464. [MIMS Preprint].

[7] Mark Gates, Hartwig Anzt, Jakub Kurzak, and Jack J. Dongarra. Accelerating collaborative filtering using concepts from high performance computing. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 667–676, 2015. doi: 10.1109/BigData.2015.7363811. URL https://doi.org/10.1109/BigData.2015.7363811.

[8] Mark Gates, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. C++ API for BLAS and LAPACK. Technical Report 2, ICL-UT-17-03, 06-2017 2017. revision 06-2017.

[9] W. Hackbusch. A Sparse Matrix Arithmetic Based on H-matrices. Part I: Introduction to H-matrices. *Computing*, 62(2):89–108, May 1999. ISSN 0010-485X. doi: 10.1007/s006070050015. URL http://dx.doi.org/10.1007/s006070050015.

[10] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack J. Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *IJHPCA*, 29(2):193–208, 2015. doi: 10.1177/1094342014567546. URL https://doi.org/10.1177/1094342014567546.

[11] Jakub Kurzak, Hartwig Anzt, Mark Gates, and Jack J. Dongarra. Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 27(7):2036–2048, 2016. doi: 10.1109/TPDS.2015.2481890. URL https://doi.org/10.1109/TPDS.2015.2481890.

[12] Jakub Kurzak, Panruo Wu, Mark Gates, Ichitaro Yamazaki, Piotr Luszczek, Gerald Ragghianti, and Jack Dongarra. Designing SLATE: Software for Linear Algebra Targeting Exascale. SLATE Working Notes 3, ICL-UT-17-06, 10-2017 2017.

[13] Steven C. Rennich, Darko Stosic, and Timothy A. Davis. Accelerating sparse Cholesky factorization on GPUs. *Parallel Computing*, 59:140–150, 2016. doi: 10.1016/j.parco.2016.06.004. URL https://doi.org/10.1016/j.parco.2016.06.004.

[14] Yang Shi, U. N. Niranjan, Animashree Anandkumar, and Cris Cecka. Tensor Contractions with Extended BLAS Kernels on CPU and GPU. In *23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India, December 19-22, 2016*, pages 193–202, 2016. doi: 10.1109/HiPC.2016.031. URL https://doi.org/10.1109/HiPC.2016.031.

[15] Stanimire Tomov, Jack J. Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010. doi: 10.1016/j.parco.2009.12.005. URL https://doi.org/10.1016/j.parco.2009.12.005.

[16] 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. ISBN 978-3-642-40047-6. doi: 10.1007/978-3-642-40047-6_81. URL https://doi.org/10.1007/978-3-642-40047-6_81.

[17] 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. doi: 10.1109/CLUSTER.2013.6702656. URL https://doi.org/10.1109/CLUSTER.2013.6702656.

[18] 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. ISSN 0098-3500. doi: 10.1145/3065870. URL http://doi.acm.org/10.1145/3065870.