User's Guide
Batched BLAS
Version 1.0

i) Flagship 2020 project, Architecture Development Team,
ii) Large-scale Parallel Numerical Computing Technology Research Team

RIKEN Center for Computational Science (R-CCS) [1]

February 2021

[1]Corresponding author: Toshiyuki Imamura (imamura.toshiyuki@riken.jp)

Contents

Chapter 1

Introduction

The emerging computer systems are primarily empowered by many-core processors or similar accelerator units. In order to boost linear algebra operations on such many-core processors, we must adopt a more strategic way to assign a large number of small-size BLAS tasks or their internal sub-tasks into each physical core, taking up all of the core resources rather than 'single-job on single-core' scheduling. The Batched BLAS would offer a new scheduling form of linear algebra operations based on such a mechanism.

Here, it is necessary to note the relationship between the process of the Batched BLAS standard and our implementation. In 2014, RIKEN initiated the flagship 2020 project to inherit the spirit of the K computer to the supercomputer Fugaku, and also kicked off several software developments, such as for operation system, runtime systems, compilers, programming languages, HPC middleware like MPI, DSL, numerical libraries, deep-learning framework, and nine specific target application codes, which we accelerate on Fugaku more than 100 fold with a superior potential of both hardware and software enhancements compared to the K computer. The heart of the supercomputer Fugaku is a Fujitsu A64FX processor, which facilitates 48 computing cores and 2 or 4 assistant cores, and it has categorizing features such as homogeneous, many-core, NUMA, and so on.

From the previous discussions, proper scheduling of physical cores and threads is important to fully use up 48 cores on a single processor. The practical application of experimental Batched BLAS, a part of which has been already applied in AI and other applications, was eagerly awaited. Since the Batched BLAS was not stored in the Fujitsu numerical library (SSL-II), the RIKEN development team developed a full version of the Batched BLAS based on a design policy that could be easily applied to the not-yet-standardized specifications in 2017.

We have made a poster presentation regarding the preliminary implementation and evaluation of Batched BLAS at ISC18 [3] to accelerate AI applications running on Fugaku. Although our implementation did not support a current standard API, we mentioned the possibility of a methodology and some scheduling techniques that would support all APIs before the standard specified all the APIs from level 1 to level 3 kernels. In July 2018, the formal standard [2] was issued, and we have also responded to adjust the APIs to the formal standard (the reference implementation of NLAFET [5]), resulting in the publication of the

package. In addition, we partially support an FP16 enhancement, which has been adopted in IEEE754-2008 as half-precision or binary16.

This user's guide for Batched BLAS version 1.0 covers almost the whole spectrum of the Batched BLAS, from A to Z, with particular consideration given to installation and compiling, a quick tutorial, the API list, and our original extensions. It is written and provided with hope of all the developer team that it will assist many users in achieving efficient parallel simulations.

Chapter 2

Installation and Licensing

This chapter describes the prerequisites and environment settings necessary to execute the Batched BLAS.

2.1 Minimally Required Environment

2.1.1 Computing Environment

The target computer is assumed as a general Linux OS-based environment.[1]

2.1.2 Software Environment

1. On an Intel x86 platform, the Intel compiler icc is supposed to run as a primal compiler (see the note below regarding the case of GNU C compiler gcc).

2. It is necessary to link with the cblas library when compiling the executable objects. When the Intel compiler is used, the MKL library is supposed to be linked as default.

3. Python generates the intermediate source codes for the Batched BLAS. If you already have the generated codes or archived libraries, python is not necessary.

(*) Note that gcc is available to use on both the building and linking phase. To do so, you need to switch the header file and use the Makefile to change the `-D_CBLAS_` option. In addition, the user can also set up references include files and libraries for gcc and cblas appropriately (See Appendix A.3).

2.2 Installation

Copy the downloaded BatchedBLAS-1.0.tar.gz onto your disk storage, and unzip it. After that, a collection of programs is created.

[1] For Fugaku and the Fujitsu HPC platforms, details are noted in Appendix A.1.

```
$ tar xzvf BatchedBLAS-1.0.tar.gz
```

The directory `batched_blas_tool` is created, and it contains all the necessary files.

2.2.1 Generation of Batched BLAS Codes

The Batched BLAS program generates the source codes automatically by running a Python script file. Next, change to the directory `batched_blas_tool` and enter the following commands.

```
$ cd BatchedBLAS-1.0
$ python bblas.py data/bblas_data.csv
$ make lib
```

Or, proceed as follows.

```
$ cd BatchedBLAS-1.0
$ /bin/sh ./make_x86.sh
```

These commands create the directory `bblas_src`, and automatically generate the Batched BLAS source codes in it. The `make` command also generates the Batched BLAS library files, `libbblas.a` (static library) and `libbblas.so` (shared library), and the associated header files `bblas.h` (see others below) are included in `bblas_src` directory. When you install the Batched BLAS header files, following six files must be copied on the system directory as well as `libbblas.a` and `libbblas.so`.

1. `bblas.h`,
2. `bblas_types.h` defining the type Macro variables,
3. `bblas_error.h` defining the Error handling Macros and function prototypes,
4. `batched_blas.h` defining the prototype of the functions,
5. `batched_blas_common.h` defining general interface with `cblass` and others, and
6. `batched_blas_fp16.h` defining the common data type for the FP16 format.

For enhancement or tuning the scheduling policy in the Batched BLAS, copy the following three headers as well.

1. `batched_blas_consts.h` defining the translation routine from single character constants to the CBLAS constants.
2. `batched_blas_cost.h` defining the cost-query interfaces, and
3. `batched_blas_schedule.h` defining the scheduler interfaces.

2.2.2 Use in Applications

Applications can use Batched BLAS by referring to the generated `bblas.h` and linking `libbblas.a` or `libbblas.so`.
Note that `libbblas.a` and `libbblas.so` call cblas linked to the executable as well.

2.3 Copyright and Licencing

Chapter 3

Quick run of Sample Programs

This section explains how to make and execute a sample program. This sample program assists your programming, and you may learn more about how to call Batched BLAS.

3.1 Build of the Sample Programs

The two sample programs can be found in the `bathed_blas_tool` directory in the sample directory (`batched_blas_dgemm_sample.c` , `batched_blas_zgemm_sample.c`). By running `make` in the `batched_blas_tool` directory, you will have executable files in the `sample` directory.

```
$ cd batched_blas_tool
$ make sample_make
```

3.2 Execution of the Sample Programs

The run of the sample programs requires you to move to the sample directory and enter a command. The sample programs are designed to set and execute `dgemm` and `zgemm` of the Batched BLAS, compare the results with those of the normal cblas routines, and print the maximum error and the execution time.

The first argument to the command is the number of groups (`group_count`). The second is the number of jobs in each group. The third is the dimension of the matrix.

```
$ cd sample
$ ./batched_blas_dgemm_sample 100 10 100
```

Though the number of jobs in the group and the order of the matrix can be set to variable in the Batched BLAS function, this sample is set to a fixed value.

Chapter 4

APIs

This chapter describes the interfaces of the Batched BLAS, corresponding to the Level 1 to Level 3 BLAS routines.

In the description, the prefix letter indicates `x` in the function name, where the first single (or two) letters `s`, `d`, `c`, and `z` correspond to single-precision floating-point number, double-precision floating-point number, single-precision-complex number, and double-precision-complex number, respectively.

4.1 General Format of the API

This section describes the general rules of the Batched BLAS routines following the batched BLAS specification [1, 2] and reference API implementation [5].

4.1.1 Fundamentals

The Batched BLAS routine assumes to call cblas as a BLAS kernel intrinsically. This implementation follows the specification and the eference code [5]. In fact, the Batched BLAS routine invokes the underlying cblas routines. Therefore, other than Batch executions, the behavior of underlying kernels etc. is identical to that of cblas [4]. Please refer to the literature explaining cblas for more details.

4.1.2 Group

Batched BLAS routines have common calling parameters that are effective within a group. For example, `xGEMM`, the matrix-matrix multiplication routine, has the following call formats.

In the routine, `transa`, `tranb`, `m`, `n`, `k`, `lda`, `ldb`, `ldc`, `alpha`, `beta`, and `group_size` correspond to such parameters. These parameters are referred by an array with the `group_counts` described below. On the other hand, the layout parameter is the same across the board, regardless of the group.

```
void blas_dgemm_batch(
    const int group_count, const int* group_size,
```

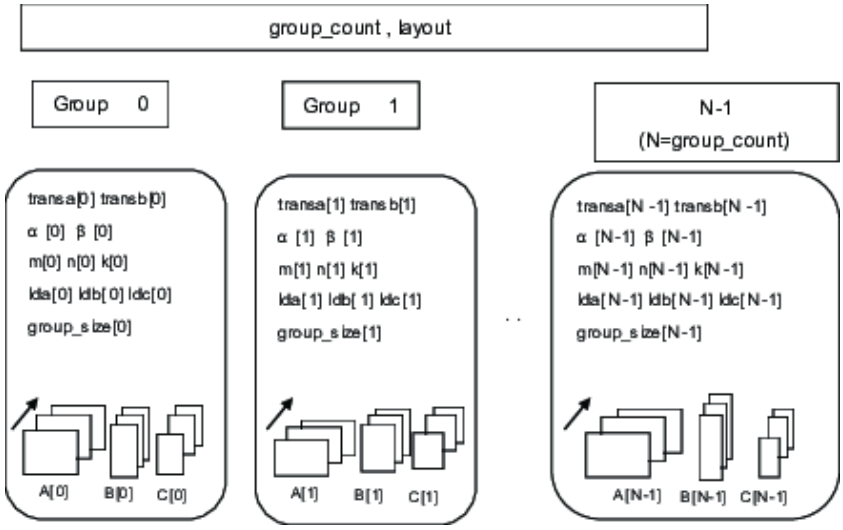

Figure 4.1: Relation of the arguments when calling `blas_dgemm_batch`

```
const bblas_enum_t layout,
const bblas_enum_t* transa, const bblas_enum_t* transb,
const int* m,const int* n, const int* k,
const double* alpha, const double** a, const int* lda,
const double** b, const int* ldb,
const double* beta, double** c, const int* ldc,
int *info)
```

The number of groups is specified in `group_count`. In addition, `group_size` indicates an array consisting `group_count` elements, which specifies the number of jobs to be executed by each group. Therefore, the number of matrices (`a,b,c`) used in `dgemm` should equal the total sum of "`group_size[]`"(See Figure 4.1).

The descriptions, in the list of routines in 4.2 to 4.4, provide the classification of this argument scope as an 'all', as a 'group', and as 'each job', respectively.

4.1.3 Handling functions that return values

Because the Batched BLAS routines generally have multiple return values, they cannot take the form of normal functions. Therefore, the result of executing the function will be returned in an additional array. For example, take the `blas_ddot_batch` routine. The underlying routine is `cblas_ddot`, a function that returns a value. The output results are returned in the array `ret_cblas_ddot[]`. The variable must be an array having a total size equaling to the sum of the `group_size[]`.

```
void blas_ddot_batch(
    const int group_count, int* group_size,
    const int* n,
```

```
    const double** x, const int* incx,
    const double** y, const int* incy,
    double* ret_cblas_ddot,
    int* info)
```

4.1.4 Treatment of complex number format

For complex numbers, as well as cblas, the real part and the imaginary part are arranged in a paired fashion. For example, the interface of `blas_zgemm_batch` is as follows

The complex numbers taken by this routine are `a`, `b`, `c`, `alpha` and `beta`. The common data type is `bblas_complex64_t`, where `a`, `b`, `c` are `bblas_complex64_t**` while `alpha` and `beta` are `bblas_complex64_t*`.

```
void blas_zgemm_batch(
    const int group_count, const int* group_size,
    const bblas_enum_t layout,
    const bblas_enum_t* transa, const bblas_enum_t* transb,
    const int* m, const int* n, const int* k,
    const bblas_complex64_t* alpha,const bblas_complex64_t** a, const int* lda,
    const bblas_complex64_t** b, const int* ldb,
    const bblas_complex64_t* beta, bblas_complex64_t** c, const int* ldc,
    int* info)
```

4.1.5 Info argument

The Batched BLAS routine requires the info argument for error handling. The user can specify an error level on the entry of `info[0]`. The available values of the error level are defined as follows in Table 4.1

Table 4.1: Error level of the Batched BLAS

Value	Error level
`BblasErrorsReportAll`	Outputs integer type numbers corresponding to all the batchwise errors. The array size must be greater than the total number of batches $+ 1$
`BblasErrorsReportGroup`	Outputs integer type numbers corresponding to the groupwise errors. The array size must be greater than the total number of groups $+ 1$
`BblasErrorsReportAny`	Outputs a single integer type number corresponding to the error. The array size must be greater than 1.
`BblasErrorsReportNone`	No error output. The array size must be greater than or equal to 1.

4.1.6 Fixed API

The fixed API is one of the special APIs calling the Batched BLAS routine that corresponds to `group_count=1`. The functions suffixed with "`batchf`" are functions of the fixed API.

4.2 Specification of APIs

4.2.1 Level 1 BLAS Routines

The interfaces of the Batched BLAS Level 1 BLAS routines are as follows.

`blas_xrotg_batch`

1. Functionalities:
   Generate plane rotation

2. Prefixes:
   `s, d`

3. Argument scope:
   All: `group_count`
   Group: `group_size`
   Each Job: `a, b, c, s`

4. Interfaces:

```
void blas_srotg_batch(
            const int group_count, const int* group_size,
            float** a, float** b, float** c, float** s,
            int* info);

void blas_drotg_batch(
            const int group_count, const int* group_size,
            double** a, double** b, double** c, double** s,
            int* info);
```

`blas_xrotmg_batch`

1. Functionalities:
   Generate modified plane rotation

2. Prefixes:
   `s, d`

3. Argument scope:
   All: `group_count`
   Group: `y1, group_size`
   Each Job: `d1, d2, x1, param`

4. Interfaces:

```
void blas_srotmg_batch(
            const int group_count, const int* group_size,
            float** d1, float** d2, float** x1, const float* y1,
            float** param,
            int* info);

void blas_drotmg_batch(
            const int group_count, const int* group_size,
            double** d1, double** d2, double** x1, const double* y1,
            double** param,
            int* info);
```

`blas_xrot_batch`

1. Functionalities:
   Apply plane rotation

2. Prefixes:
   `s, d`

3. Argument scope:
   All: `group_count`
   Group: `n, incx, incy , c, s , group_size`
   Each Job: `x, y`

4. Interfaces:

```
void blas_srot_batch(
            const int group_count, const int* group_size,
            const int* n,
            float** x, const int* incx,
            float** y, const int* incy,
            const float* c1, const float* s,
            int* info);

void blas_drot_batch(
            const int group_count, const int* group_size,
            const int* n,
            double** x, const int* incx,
            double** y, const int* incy,
            const double* c1, const double* s,
            int* info);
```

`blas_xrotm_batch`

1. Functionalities:
   Apply modified plane rotation
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `group_count`
   Group: `n, incx, incy, group_size`
   Each Job: `x, y, param`
4. Interfaces:

```
void blas_srotm_batch(
            const int group_count, const int* group_size,
            const int* n,
            float** x, const int* incx, float** y, const int* incy,
            const float** param,
            int* info);

void blas_drotm_batch(
            const int group_count, const int* group_size,
            const int* n,
            double** x, const int* incx, double** y, const int* incy,
            const double** param,
            int* info);
```

`blas_xswap_batch`

1. Functionalities:
   $x \leftrightarrow y$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `group_count`
   Group: `n, incx, incy, group_size`
   Each Job: `x, y`
4. Interfaces:

```
void blas_sswap_batch(
            const int group_count, const int* group_size,
            const int* n,
```

```
                float** x, const int* incx,
                float** y, const int* incy,
                int* info);

    void blas_dswap_batch(
                const int group_count, const int* group_size,
                const int* n,
                double** x, const int* incx,
                double** y, const int* incy,
                int* info);

    void blas_cswap_batch(
                const int group_count, const int* group_size,
                const int* n,
                bblas_complex32_t** x, const int* incx,
                bblas_complex32_t** y, const int* incy,
                int* info);

    void blas_zswap_batch(
                const int group_count, const int* group_size,
                const int* n,
                bblas_complex64_t** x, const int* incx,
                bblas_complex64_t** y, const int* incy,
                int* info);
```

`blas_xscal_batch`

1. Functionalities:
   $x \leftarrow \alpha x$
2. Prefixes:
   `s, d, c, z, cs, zd`
3. Argument scope:
   All: `group_count`
   Group: `n, a, incx, group_size`
   Each Job: `x`
4. Interfaces:

```
    void blas_sscal_batch(
                const int group_count, const int* group_size,
                const int* n,
                const float* a, float** x, const int* incx,
                int* info);
```

```
void blas_dscal_batch(
            const int group_count, const int* group_size,
            const int* n,
            const double* a, double** x, const int* incx,
            int* info);

void blas_cscal_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex32_t* a, bblas_complex32_t** x, const int* incx,
            int* info);

void blas_zscal_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex64_t* a, bblas_complex64_t** x, const int* incx,
            int* info);

void blas_csscal_batch(
            const int group_count, const int* group_size,
            const int* n,
            const float* a, bblas_complex32_t** x, const int* incx,
            int* info);

void blas_zdscal_batch(
            const int group_count, const int* group_size,
            const int* n,
            const double* a, bblas_complex64_t** x, const int* incx,
            int* info);
```

`blas_xcopy_batch`

1. Functionalities:
   $y \leftarrow x$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `group_count`
   Group: `n, incx, incy, group_size`
   Each Job: `x, y`
4. Interfaces:

```
void blas_scopy_batch(
            const int group_count, const int* group_size,
            const int* n,
            const float** x, const int* incx,
            float** y, const int* incy,
            int* info);

void blas_dcopy_batch(
            const int group_count, const int* group_size,
            const int* n,
            const double** x, const int* incx,
            double** y, const int* incy,
            int* info);

void blas_ccopy_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex32_t** x, const int* incx,
            bblas_complex32_t** y, const int* incy,
            int* info);

void blas_zcopy_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex64_t** x, const int* incx,
            bblas_complex64_t** y, const int* incy,
            int* info);
```

`blas_xaxpy_batch`

1. Functionalities:
   $y \leftarrow \alpha x + y$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `group_count`
   Group: `n, a, incx, incy, group_size`
   Each Job: `x, y`
4. Interfaces:

```
void blas_saxpy_batch(
            const int group_count, const int* group_size,
```

```
                const int* n,
                const float* a,
                const float** x, const int* incx,
                float** y, const int* incy,
                int* info);

    void blas_daxpy_batch(
                const int group_count, const int* group_size,
                const int* n,
                const double* a,
                const double** x, const int* incx,
                double** y, const int* incy,
                int* info);

    void blas_caxpy_batch(
                const int group_count, const int* group_size,
                const int* n,
                const bblas_complex32_t* a,
                const bblas_complex32_t** x, const int* incx,
                bblas_complex32_t** y, const int* incy,
                int* info);

    void blas_zaxpy_batch(
                const int group_count, const int* group_size,
                const int* n,
                const bblas_complex64_t* a,
                const bblas_complex64_t** x, const int* incx,
                bblas_complex64_t** y, const int* incy,
                int* info);
```

`blas_xdot_batch`

1. Functionalities:
   `dot` $\leftarrow x^T y$
2. Prefixes:
   `s, d, ds`
3. Argument scope:
   All: `group_count`
   Group: `n, incx, incy, group_size`
   Each Job: `x, y, ret_cblas_xdot`
   NOTE: return values are stored in `ret_cblas_xdot`
4. Interfaces:

```
void blas_sdot_batch(
            const int group_count, const int* group_size,
            const int* n,
            const float** x, const int* incx,
            const float** y, const int* incy,
            float* ret_cblas_sdot,
            int* info);

void blas_ddot_batch(
            const int group_count, const int* group_size,
            const int* n,
            const double** x, const int* incx,
            const double** y, const int* incy,
            double* ret_cblas_ddot,
            int* info);

void blas_dsdot_batch(
            const int group_count, const int* group_size,
            const int* n,
            const float** x, const int* incx,
            const float** y, const int* incy,
            double* ret_cblas_dsdot,
            int* info);
```

`blas_xdotu_batch`

1. Functionalities:
   `dotu` $\leftarrow x^T y$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `group_count`
   Group: `n, incx, incy, group_size`
   Each Job: `x, y, ret_cblas_dotu`
   NOTE: return values are stored in `ret_cblas_dotu`
4. Interfaces:

```
void blas_cdotu_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex32_t** x, const int* incx,
            const bblas_complex32_t** y, const int* incy,
```

```
            bblas_complex32_t** ret_cblas_dotu,
            int* info);

    void blas_zdotu_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex64_t** x, const int* incx,
            const bblas_complex64_t** y, const int* incy,
            bblas_complex64_t** ret_cblas_dotu,
            int* info);
```

`blas_xdotc_batch`

1. Functionalities:
   `dotc` $\leftarrow x^H y$

2. Prefixes:
   `c, z`

3. Argument scope:
   All: `group_count`
   Group: `n, incx, incy, group_size`
   Each Job: `x, y, ret_cblas_dotc`
   NOTE: return values are stored in `ret_cblas_dotc`

4. Interfaces:

```
    void blas_cdotc_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex32_t** x, const int* incx,
            const bblas_complex32_t** y, const int* incy,
            bblas_complex32_t** ret_cblas_dotc,
            int* info);

    void blas_zdotc_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex64_t** x, const int* incx,
            const bblas_complex64_t** y, const int* incy,
            bblas_complex64_t** ret_cblas_dotc,
            int* info);
```

`blas_xxdot_batch`

1. Functionalities:
   `dot` $\leftarrow \alpha + x^T y$
2. Prefixes:
   `sds`
3. Argument scope:
   All: `group_count`
   Group: `n, a, incx, incy, group_size`
   Each Job: `x, y, ret_cblas_sdsdot`
   NOTE: return values are stored in `ret_cblas_sdsdot`
4. Interfaces:

```
void blas_sdsdot_batch(
            const int group_count, const int* group_size,
            const int* n,
            const float* a, const float** x, const int* incx,
            const float** y, const int* incy,
            float* ret_cblas_sdsdot,
            int* info);
```

`blas_xnrm2_batch`

1. Functionalities:
   `nrm2` $\leftarrow \|x\|_2$
2. Prefixes:
   `s, d, sc, dz`
3. Argument scope:
   All: `group_count`
   Group: `n, incx, group_size`
   Each Job: `x, ret_cblas_xnrm2`
   NOTE: return values are stored in `ret_cblas_xnrm2`
4. Interfaces:

```
void blas_snrm2_batch(
            const int group_count, const int* group_size,
            const int* n,
            const float** x, const int* incx,
            float* ret_cblas_snrm2,
            int* info);
```

```
void blas_dnrm2_batch(
            const int group_count, const int* group_size,
            const int* n,
            const double** x, const int* incx,
            double* ret_cblas_dnrm2,
            int* info);

void blas_scnrm2_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex32_t** x, const int* incx,
            float* ret_cblas_scnrm2,
            int* info);

void blas_dznrm2_batch(
            const int group_count, const int* group_size,
            const int* n,
            const bblas_complex64_t** x, const int* incx,
            double* ret_cblas_dznrm2,
            int* info);
```

`blas_xasum_batch`

1. Functionalities:
   asum $\leftarrow \|\mathrm{Re}(x)\|_1 + \|\mathrm{Im}(x)\|_1$
2. Prefixes:
   `s, d, sc, dz`
3. Argument scope:
   All: `group_count`
   Group: `n, incx, group_size`
   Each Job: `x, ret_cblas_xasum`
   NOTE: return values are stored in `ret_cblas_xasum`
4. Interfaces:

```
void blas_sasum_batch(
            const int group_count, const int* group_size,
            const int* n,
            const float** x, const int* incx,
            float* ret_cblas_sasum,
            int* info);

void blas_dasum_batch(
```

```
                const int group_count, const int* group_size,
                const int* n,
                const double** x, const int* incx,
                double* ret_cblas_dasum,
                int* info);

    void blas_scasum_batch(
                const int group_count, const int* group_size,
                const int* n,
                const bblas_complex32_t** x, const int* incx,
                float* ret_cblas_scasum,
                int* info);

    void blas_dzasum_batch(
                const int group_count, const int* group_size,
                const int* n,
                const bblas_complex64_t** x, const int* incx,
                double* ret_cblas_dzasum,
                int* info);
```

`blas_ixamax_batch`

1. Functionalities:
   amax $\leftarrow$ 1st $k \ni |\mathrm{Re}(x_k)| + |\mathrm{Im}(x_k)| = \max_i(|\mathrm{Re}(x_i)| + |\mathrm{Im}(x_i)|)$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `group_count`
   Group: `n, incx, group_size`
   Each Job: `x, ret_cblas_ixamax`
   NOTE: return values are stored in `ret_cblas_ixamax`
4. Interfaces:

```
    void blas_isamax_batch(
                const int group_count, const int* group_size,
                const int* n,
                const float** x, const int* incx,
                int* ret_cblas_isamax,
                int* info);

    void blas_idamax_batch(
                const int group_count, const int* group_size,
```

```
                const int* n,
                const double** x, const int* incx,
                int* ret_cblas_idamax,
                int* info);

    void blas_icamax_batch(
                const int group_count, const int* group_size,
                const int* n,
                const bblas_complex32_t** x, const int* incx,
                int* ret_cblas_icamax,
                int* info);

    void blas_izamax_batch(
                const int group_count, const int* group_size,
                const int* n,
                const bblas_complex64_t** x, const int* incx,
                int* ret_cblas_izamax,
                int* info);
```

`blas_xrotg_batchf`

1. Functionalities:
   Generate plane rotation

2. Prefixes:
   `s, d`

3. Argument scope:
   Group: `group_size`
   Each Job: `a, b, c, s`

4. Interfaces:

```
    void blas_srotg_batchf(
                const int group_size,
                float** a, float** b, float** c, float** s,
                int* info);

    void blas_drotg_batchf(
                const int group_size,
                double** a, double** b, double** c, double** s,
                int* info);
```

`blas_xrotmg_batchf`

1. Functionalities:
   Generate modified plane rotation

2. Prefixes:
   `s, d`

3. Argument scope:
   Group: `y1, group_size`
   Each Job: `d1, d2, x1, param`

4. Interfaces:

```
void blas_srotmg_batchf(
            const int group_size,
            float** d1, float** d2, float** x1, const float* y1,
            float** param,
            int* info);

void blas_drotmg_batchf(
            const int group_size,
            double** d1, double** d2, double** x1, const double* y1,
            double** param,
            int* info);
```

`blas_xrot_batchf`

1. Functionalities:
   Apply plane rotation

2. Prefixes:
   `s, d`

3. Argument scope:
   Group: `n, incx, incy , c, s , group_size`
   Each Job: `x, y`

4. Interfaces:

```
void blas_srot_batchf(
            const int group_size,
            const int n,
            float** x, const int incx, , float** y, const int incy,
            const float c1, const float s1,
            int* info);
```

```
void blas_drot_batchf(
            const int group_size,
            const int n,
            double** x, const int incx, , double** y, const int incy,
            const double c1, const double s1,
            int* info);
```

`blas_xrotm_batchf`

1. Functionalities:
   Apply modified plane rotation
2. Prefixes:
   `s, d`
3. Argument scope:
   Group: `n, incx, incy, group_size`
   Each Job: `x, y, param`
4. Interfaces:

```
void blas_srotm_batchf(
            const int group_size,
            const int n,
            float** x, const int incx, , float** y, const int incy,
            const float** param,
            int* info);

void blas_drotm_batchf(
            const int group_size,
            const int n,
            double** x, const int incx, , double** y, const int incy,
            const double** param,
            int* info);
```

`blas_xswap_batchf`

1. Functionalities:
   $x \leftrightarrow y$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   Group: `n, incx, incy, group_size`
   Each Job: `x, y`

4. Interfaces:

```
void blas_sswap_batchf(
            const int group_size,
            const int n,
            float** x, const int incx, float** y, const int incy,
            int* info);

void blas_dswap_batchf(
            const int group_size,
            const int n,
            double** x, const int incx, double** y, const int incy,
            int* info);

void blas_cswap_batchf(
            const int group_size,
            const int n,
            bblas_complex32_t** x, const int incx,
            bblas_complex32_t** y, const int incy,
            int* info);

void blas_zswap_batchf(
            const int group_size,
            const int n,
            bblas_complex64_t** x, const int incx,
            bblas_complex64_t** y, const int incy,
            int* info);
```

`blas_xscal_batchf`

1. Functionalities:
   $x \leftarrow \alpha x$
2. Prefixes:
   `s, d, c, z, cs, zd`
3. Argument scope:
   Group: `n, a, incx, group_size`
   Each Job: `x`
4. Interfaces:

```
void blas_sscal_batchf(
            const int group_size,
            const int n,
```

```
            const float a1, float** x, const int incx,
            int* info);

    void blas_dscal_batchf(
            const int group_size,
            const int n,
            const double a1, double** x, const int incx,
            int* info);

    void blas_cscal_batchf(
            const int group_size,
            const int n,
            const bblas_complex32_t* a, bblas_complex32_t** x, const int incx,
            int* info);

    void blas_zscal_batchf(
            const int group_size,
            const int n,
            const bblas_complex64_t* a, bblas_complex64_t** x, const int incx,
            int* info);

    void blas_csscal_batchf(
            const int group_size,
            const int n,
            const float a1, bblas_complex32_t** x, const int incx,
            int* info);

    void blas_zdscal_batchf(
            const int group_size,
            const int n,
            const double a1, bblas_complex64_t** x, const int incx,
            int* info);
```

`blas_xcopy_batchf`

1. Functionalities:
   $y \leftarrow x$

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   Group: `n, incx, incy, group_size`
   Each Job: `x, y`

4. Interfaces:

```
void blas_scopy_batchf(
            const int group_size,
            const int n,
            const float** x, const int incx,
            float** y, const int incy,
            int* info);

void blas_dcopy_batchf(
            const int group_size,
            const int n,
            const double** x, const int incx,
            double** y, const int incy,
            int* info);

void blas_ccopy_batchf(
            const int group_size,
            const int n,
            const bblas_complex32_t** x, const int incx,
            bblas_complex32_t** y, const int incy,
            int* info);

void blas_zcopy_batchf(
            const int group_size,
            const int n,
            const bblas_complex64_t** x, const int incx,
            bblas_complex64_t** y, const int incy,
            int* info);
```

`blas_xaxpy_batchf`

1. Functionalities:
   $y \leftarrow \alpha x + y$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   Group: `n, a1, incx, incy, group_size`
   Each Job: `x, y`
4. Interfaces:

```
void blas_saxpy_batchf(
```

```
                const int group_size,
                const int n,
                const float a1,
                const float** x, const int incx,
                float** y, const int incy,
                int* info);

    void blas_daxpy_batchf(
                const int group_size,
                const int n,
                const double a1,
                const double** x, const int incx,
                double** y, const int incy,
                int* info);

    void blas_caxpy_batchf(
                const int group_size,
                const int n,
                const bblas_complex32_t a1,
                const bblas_complex32_t** x, const int incx,
                bblas_complex32_t** y, const int incy,
                int* info);

    void blas_zaxpy_batchf(
                const int group_size,
                const int n,
                const bblas_complex64_t a1,
                const bblas_complex64_t** x, const int incx,
                bblas_complex64_t** y, const int incy,
                int* info);
```

`blas_xdot_batchf`

1. Functionalities:
   `dot` $\leftarrow x^T y$
2. Prefixes:
   `s, d, ds`
3. Argument scope:
   Group: `n, incx, incy, group_size`
   Each Job: `x, y, ret_cblas_xdot`
   NOTE: return values are stored in `ret_cblas_xdot`
4. Interfaces:

```
void blas_sdot_batchf(
            const int group_size,
            const int n,
            const float** x, const int incx, const float** y, const int incy,
            float* ret_cblas_sdot,
            int* info);

void blas_ddot_batchf(
            const int group_size,
            const int n,
            const double** x, const int incx, const double** y, const int incy,
            double* ret_cblas_ddot,
            int* info);

void blas_dsdot_batchf(
            const int group_size,
            const int n,
            const float** x, const int incx, const float** y, const int incy,
            double* ret_cblas_dsdot,
            int* info);
```

`blas_xdotu_batchf`

1. Functionalities:
   `dotu` $\leftarrow x^T y$
2. Prefixes:
   `c, z`
3. Argument scope:
   Group: `n, incx, incy, group_size`
   Each Job: `x, y, ret_cblas_dotu`
   NOTE: return values are stored in `ret_cblas_dotu`
4. Interfaces:

```
void blas_cdotu_batchf(
            const int group_size,
            const int n,
            const bblas_complex32_t** x, const int incx,
            const bblas_complex32_t** y, const int incy,
            bblas_complex32_t** ret_cblas_dotu,
            int* info);

void blas_zdotu_batchf(
```

```
                const int group_size,
                const int n,
                const bblas_complex64_t** x, const int incx,
                const bblas_complex64_t** y, const int incy,
                bblas_complex64_t** ret_cblas_dotu,
                int* info);
```

`blas_xdotc_batchf`

1. Functionalities:
   `dotc` $\leftarrow x^H y$
2. Prefixes:
   `c, z`
3. Argument scope:
   Group: `n, incx, incy, group_size`
   Each Job: `x, y, ret_cblas_dotc`
   NOTE: return values are stored in `ret_cblas_dotc`
4. Interfaces:

```
void blas_cdotc_batchf(
            const int group_size,
            const int n,
            const bblas_complex32_t** x, const int incx,
            const bblas_complex32_t** y, const int incy,
            bblas_complex32_t** ret_cblas_dotc,
            int* info);

void blas_zdotc_batchf(
            const int group_size,
            const int n,
            const bblas_complex64_t** x, const int incx,
            const bblas_complex64_t** y, const int incy,
            bblas_complex64_t** ret_cblas_dotc,
            int* info);
```

`blas_xxdot_batchf`

1. Functionalities:
   `dot` $\leftarrow \alpha + x^T y$
2. Prefixes:
   `sds`

3. Argument scope:
   Group: `n, a, incx, incy, group_size`
   Each Job: `x, y, ret_cblas_sdsdot`
   NOTE: return values are stored in `ret_cblas_sdsdot`

4. Interfaces:

```
void blas_sdsdot_batchf(
            const int group_size,
            const int n, const float b,
            const float** x, const int incx,
            const float** y, const int incy,
            float* ret_cblas_sdsdot,
            int* info);
```

`blas_xnrm2_batchf`

1. Functionalities:
   `nrm2` $\leftarrow \|x\|_2$

2. Prefixes:
   `s, d, sc, dz`

3. Argument scope:
   Group: `n, incx, group_size`
   Each Job: `x, ret_cblas_xnrm2`
   NOTE: return values are stored in `ret_cblas_xnrm2`

4. Interfaces:

```
void blas_snrm2_batchf(
            const int group_size,
            const int n,
            const float** x, const int incx,
            float* ret_cblas_snrm2,
            int* info);

void blas_dnrm2_batchf(
            const int group_size,
            const int n,
            const double** x, const int incx,
            double* ret_cblas_dnrm2,
            int* info);

void blas_scnrm2_batchf(
            const int group_size,
```

```
                const int n, const bblas_complex32_t** x, const int incx,
                float* ret_cblas_scnrm2,
                int* info);

    void blas_dznrm2_batchf(
                const int group_size,
                const int n,
                const bblas_complex64_t** x, const int incx,
                double* ret_cblas_dznrm2,
                int* info);
```

`blas_xasum_batchf`

1. Functionalities:
   asum $\leftarrow \|\mathrm{Re}(x)\|_1 + \|\mathrm{Im}(x)\|_1$
2. Prefixes:
   `s, d, sc, dz`
3. Argument scope:
   Group: `n, incx, group_size`
   Each Job: `x, ret_cblas_xasum`
   NOTE: return values are stored in `ret_cblas_xasum`
4. Interfaces:

```
    void blas_sasum_batchf(
                const int group_size,
                const int n,
                const float** x, const int incx,
                float* ret_cblas_sasum,
                int* info);

    void blas_dasum_batchf(
                const int group_size,
                const int n,
                const double** x, const int incx,
                double* ret_cblas_dasum,
                int* info);

    void blas_scasum_batchf(
                const int group_size,
                const int n,
                const bblas_complex32_t** x, const int incx,
                float* ret_cblas_scasum,
```

```
                int* info);

    void blas_dzasum_batchf(
                const int group_size,
                const int n,
                const bblas_complex64_t** x, const int incx,
                double* ret_cblas_dzasum,
                int* info);
```

`blas_ixamax_batchf`

1. Functionalities:
   `amax` $\leftarrow 1\text{st } k \ni |\mathrm{Re}(x_k)| + |\mathrm{Im}(x_k)| = \max_i(|\mathrm{Re}(x_i)| + |\mathrm{Im}(x_i)|)$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   Group: `n, incx, group_size`
   Each Job: `x, ret_cblas_ixamax`
   NOTE: return values are stored in `ret_cblas_ixamax`
4. Interfaces:

```
    void blas_isamax_batchf(
                const int group_size,
                const int n,
                const float** x, const int incx,
                int* ret_cblas_isamax,
                int* info);

    void blas_idamax_batchf(
                const int group_size,
                const int n,
                const double** x, const int incx,
                int* ret_cblas_idamax,
                int* info);

    void blas_icamax_batchf(
                const int group_size,
                const int n,
                const bblas_complex32_t** x, const int incx,
                int* ret_cblas_icamax,
                int* info);
```

```
void blas_izamax_batchf(
            const int group_size,
            const int n,
            const bblas_complex64_t** x, const int incx,
            int* ret_cblas_izamax,
            int* info);
```

4.2.2 Level 2 BLAS Routines

The interfaces of the Batched BLAS Level 2 BLAS routines are as follows.

`blas_xgemv_batch`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,\ y \leftarrow \alpha A^H x + \beta y,\ A \in \mathrm{K}^{m \times n}$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `transa, m, n, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`
4. Interfaces:

```
void blas_sgemv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* transa,
            const int* m, const int* n,
            const float* alpha,
            const float** a, const int* lda,
            const float** x, const int* incx,
            const float* beta,
            float** y, const int* incy,
            int* info);

void blas_dgemv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* transa,
            const int* m, const int* n,
            const double* alpha,
            const double** a, const int* lda,
            const double** x, const int* incx,
            const double* beta,
            double** y, const int* incy,
            int* info);

void blas_cgemv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* transa,
            const int* m, const int* n,
```

```
                const bblas_complex32_t* alpha,
                const bblas_complex32_t** a, const int* lda,
                const bblas_complex32_t** x, const int* incx,
                const bblas_complex32_t* beta,
                bblas_complex32_t** y, const int* incy,
                int* info);

    void blas_zgemv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* transa,
                const int* m, const int* n,
                const bblas_complex64_t* alpha,
                const bblas_complex64_t** a, const int* lda,
                const bblas_complex64_t** x, const int* incx,
                const bblas_complex64_t* beta,
                bblas_complex64_t** y, const int* incy,
                int* info);
```

`blas_xgbmv_batch`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,\ y \leftarrow \alpha A^H x + \beta y,\ A \in \mathrm{Band}(\mathrm{kl}, \mathrm{ku})$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `trans, m, n, kl, ku, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`
4. Interfaces:

```
    void blas_sgbmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* trans,
                const int* m, const int* n, const int* kl, const int* ku,
                const float* alpha,
                const float** a, const int* lda,
                const float** x, const int* incx,
                const float* beta,
                float** y, const int* incy,
                int* info);

    void blas_dgbmv_batch(
```

```
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* trans,
                const int* m, const int* n, const int* kl, const int* ku,
                const double* alpha,
                const double** a, const int* lda,
                const double** x, const int* incx,
                const double* beta, double** y, const int* incy,
                int* info);

    void blas_cgbmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* trans,
                const int* m, const int* n, const int* kl, const int* ku,
                const bblas_complex32_t* alpha,
                const bblas_complex32_t** a, const int* lda,
                const bblas_complex32_t** x, const int* incx,
                const bblas_complex32_t* beta,
                bblas_complex32_t** y, const int* incy,
                int* info);

    void blas_zgbmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* trans,
                const int* m, const int* n, const int* kl, const int* ku,
                const bblas_complex64_t* alpha,
                const bblas_complex64_t** a, const int* lda,
                const bblas_complex64_t** x, const int* incx,
                const bblas_complex64_t* beta,
                bblas_complex64_t** y, const int* incy,
                int* info);
```

`blas_xhemv_batch`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Hermite}$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`
4. Interfaces:

```
void blas_chemv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** a, const int* lda,
            const bblas_complex32_t** x, const int* incx,
            const bblas_complex32_t* beta,
            bblas_complex32_t** y, const int* incy,
            int* info);

void blas_zhemv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo, const int* n,
            const bblas_complex64_t* alpha,
            const bblas_complex64_t** a, const int* lda,
            const bblas_complex64_t** x, const int* incx,
            const bblas_complex64_t* beta,
            bblas_complex64_t** y, const int* incy,
            int* info);
```

`blas_xhbmv_batch`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Hermit\&Band}(\text{band width} = k)$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, k, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`
4. Interfaces:

```
void blas_chbmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n, const int* k,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** a, const int* lda,
            const bblas_complex32_t** x, const int* incx,
            const bblas_complex32_t* beta,
            bblas_complex32_t** y, const int* incy,
```

```
                int* info);

    void blas_zhbmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const int* n, const int* k,
                const bblas_complex64_t* alpha,
                const bblas_complex64_t** a, const int* lda,
                const bblas_complex64_t** x, const int* incx,
                const bblas_complex64_t* beta,
                bblas_complex64_t** y, const int* incy,
                int* info);
```

`blas_xhpmv_batch`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Packed\&Hermite}$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, incx, beta, incy, group_size`
   Each Job: `ap, x, y`
4. Interfaces:

```
    void blas_chpmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const int* n,
                const bblas_complex32_t* alpha,
                const bblas_complex32_t** ap,
                const bblas_complex32_t** x, const int* incx,
                const bblas_complex32_t* beta,
                bblas_complex32_t** y, const int* incy,
                int* info);

    void blas_zhpmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const int* n,
                const bblas_complex64_t* alpha, const bblas_complex64_t** ap,
                const bblas_complex64_t** x, const int* incx,
```

```
                const bblas_complex64_t* beta,
                bblas_complex64_t** y, const int* incy,
                int* info);
```

`blas_xsymv_batch`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Symmetric}$
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`
4. Interfaces:

```
void blas_ssymv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const float* alpha,
            const float** a, const int* lda,
            const float** x, const int* incx,
            const float* beta,
            float** y, const int* incy,
            int* info);

void blas_dsymv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const double* alpha,
            const double** a, const int* lda,
            const double** x, const int* incx,
            const double* beta,
            double** y, const int* incy,
            int* info);
```

`blas_xsbmv_batch`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Symmetric\&Band}(\text{band width} = k)$

2. Prefixes:
   `s, d`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, k, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`

4. Interfaces:

```
void blas_ssbmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n, const int* k,
            const float* alpha,
            const float** a, const int* lda,
            const float** x, const int* incx,
            const float* beta,
            float** y, const int* incy,
            int* info);

void blas_dsbmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n, const int* k,
            const double* alpha,
            const double** a, const int* lda,
            const double** x, const int* incx,
            const double* beta,
            double** y, const int* incy,
            int* info);
```

`blas_xspmv_batch`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Packed\&Symmetric}$

2. Prefixes:
   `s, d`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, incx, beta, incy, group_size`
   Each Job: `ap, x, y`

4. Interfaces:

```
void blas_sspmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const float* alpha, const float** ap,
            const float** x, const int* incx,
            const float* beta,
            float** y, const int* incy,
            int* info);

void blas_dspmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const double* alpha, const double** ap,
            const double** x, const int* incx,
            const double* beta,
            double** y, const int* incy,
            int* info);
```

`blas_xtrmv_batch`

1. Functionalities:
   $x \leftarrow Ax,\ x \leftarrow A^T x,\ x \leftarrow A^H x,\ A \in \mathrm{K}^{n \times n},\ A$: Triangular
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, trans, diag, n, lda, incx, group_size`
   Each Job: `a, x`
4. Interfaces:

```
void blas_strmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const float** a, const int* lda,
            float** x, const int* incx,
            int* info);

void blas_dtrmv_batch(
```

```
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const bblas_enum_t* diag,
                const int* n,
                const double** a, const int* lda,
                double** x, const int* incx,
                int* info);

    void blas_ctrmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const bblas_enum_t* diag,
                const int* n,
                const bblas_complex32_t** a, const int* lda,
                bblas_complex32_t** x, const int* incx,
                int* info);

    void blas_ztrmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const bblas_enum_t* diag,
                const int* n,
                const bblas_complex64_t** a, const int* lda,
                bblas_complex64_t** x, const int* incx,
                int* info);
```

`blas_xtbmv_batch`

1. Functionalities:
   $x \leftarrow Ax$, $x \leftarrow A^T x$, $x \leftarrow A^H x$, $A \in \mathrm{K}^{n \times n}$, A : Band&Triangular(band width $= k$)
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, trans, diag, n, k, lda, incx, group_size`
   Each Job: `a, x`
4. Interfaces:

```
    void blas_stbmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const bblas_enum_t* diag,
```

```
                const int* n, const int* k,
                const float** a, const int* lda,
                float** x, const int* incx,
                int* info);

    void blas_dtbmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const bblas_enum_t* diag,
                const int* n, const int* k,
                const double** a, const int* lda,
                double** x, const int* incx,
                int* info);

    void blas_ctbmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const bblas_enum_t* diag,
                const int* n, const int* k,
                const bblas_complex32_t** a, const int* lda,
                bblas_complex32_t** x, const int* incx,
                int* info);

    void blas_ztbmv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const bblas_enum_t* diag,
                const int* n, const int* k,
                const bblas_complex64_t** a, const int* lda,
                bblas_complex64_t** x, const int* incx,
                int* info);
```

`blas_xtpmv_batch`

1. Functionalities:
   $x \leftarrow Ax,\ x \leftarrow A^T x,\ x \leftarrow A^H x,\ A \in \mathrm{K}^{n \times n},\ A$: Packed&Triangular

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, trans, diag, n, incx, group_size`
   Each Job: `ap, x`

4. Interfaces:

```
void blas_stpmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const float** ap,
            float** x, const int* incx,
            int* info);

void blas_dtpmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const double** ap,
            double** x, const int* incx,
            int* info);

void blas_ctpmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const bblas_complex32_t** ap,
            bblas_complex32_t** x, const int* incx,
            int* info);

void blas_ztpmv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const bblas_complex64_t** ap,
            bblas_complex64_t** x, const int* incx,
            int* info);
```

`blas_xtrsv_batch`

1. Functionalities:
   $x \leftarrow A^{-1}x,\ x \leftarrow A^{-T}x,\ x \leftarrow A^{-H}x,\ A \in \mathrm{K}^{n\times n},\ A$: Triangular

2. Prefixes:

`s, d, c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, trans, diag, n, lda, incx, group_size`
   Each Job: `a, x`

4. Interfaces:

```
void blas_strsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n, const float** a, const int* lda,
            float** x, const int* incx,
            int* info);

void blas_dtrsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const double** a, const int* lda,
            double** x, const int* incx,
            int* info);

void blas_ctrsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const bblas_complex32_t** a, const int* lda,
            bblas_complex32_t** x, const int* incx,
            int* info);

void blas_ztrsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const bblas_complex64_t** a, const int* lda,
            bblas_complex64_t** x, const int* incx,
            int* info);
```

`blas_xtbsv_batch`

1. Functionalities:
   $x \leftarrow A^{-1}x$, $x \leftarrow A^{-T}x$, $x \leftarrow A^{-H}x$, $A \in \mathrm{K}^{n\times n}$, A : Band&Triangular(band width = k

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, trans, diag, n, k, lda, incx, group_size`
   Each Job: `a, x`

4. Interfaces:

```
void blas_stbsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n, const int* k,
            const float** a, const int* lda,
            float** x, const int* incx,
            int* info);

void blas_dtbsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n, const int* k,
            const double** a, const int* lda,
            double** x, const int* incx,
            int* info);

void blas_ctbsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n, const int* k,
            const bblas_complex32_t** a, const int* lda,
            bblas_complex32_t** x, const int* incx,
            int* info);

void blas_ztbsv_batch(
            const int group_count, const int* group_size,
```

```
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const bblas_enum_t* diag,
                const int* n, const int* k,
                const bblas_complex64_t** a, const int* lda,
                bblas_complex64_t** x, const int* incx,
                int* info);
```

`blas_xtpsv_batch`

1. Functionalities:
   $x \leftarrow A^{-1}x,\ x \leftarrow A^{-T}x,\ x \leftarrow A^{-H}x,\ A \in \mathrm{K}^{n\times n},\ A : \mathrm{Packed\&Triangular}$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, trans, diag, n, incx, group_size`
   Each Job: `ap, x`
4. Interfaces:

```
void blas_stpsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const float** ap,
            float** x, const int* incx,
            int* info);

void blas_dtpsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
            const double** ap,
            double** x, const int* incx,
            int* info);

void blas_ctpsv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const bblas_enum_t* diag,
            const int* n,
```

```
                const bblas_complex32_t** ap,
                bblas_complex32_t** x, const int* incx,
                int* info);

    void blas_ztpsv_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const bblas_enum_t* diag,
                const int* n,
                const bblas_complex64_t** ap,
                bblas_complex64_t** x, const int* incx,
                int* info);
```

`blas_xger_batch`

1. Functionalities:
   $A \leftarrow \alpha x y^T + \beta A,\ A \in \mathrm{R}^{m \times n}$
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout, group_count`
   Group: `m, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`
4. Interfaces:

```
    void blas_sger_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout,
                const int* m, const int* n,
                const float* alpha,
                const float** x, const int* incx,
                const float** y, const int* incy,
                float** a, const int* lda,
                int* info);

    void blas_dger_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout,
                const int* m, const int* n,
                const double* alpha,
                const double** x, const int* incx,
                const double** y, const int* incy,
```

```
                double** a, const int* lda,
                int* info);
```

`blas_xgeru_batch`

1. Functionalities:
   $A \leftarrow \alpha x y^T + \beta A,\ A \in \mathrm{C}^{m \times n}$

2. Prefixes:
   `c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `m, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`

4. Interfaces:

```
void blas_cgeru_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const int* m, const int* n,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** x, const int* incx,
            const bblas_complex32_t** y, const int* incy,
            bblas_complex32_t** a, const int* lda,
            int* info);

void blas_zgeru_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const int* m, const int* n,
            const bblas_complex64_t* alpha,
            const bblas_complex64_t** x, const int* incx,
            const bblas_complex64_t** y, const int* incy,
            bblas_complex64_t** a, const int* lda,
            int* info);
```

`blas_xgerc_batch`

1. Functionalities:
   $A \leftarrow \alpha x y^H + \beta A,\ A \in \mathrm{C}^{m \times n}$

2. Prefixes:
   `c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `m, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`

4. Interfaces:

```
void blas_cgerc_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const int* m, const int* n,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** x, const int* incx,
            const bblas_complex32_t** y, const int* incy,
            bblas_complex32_t** a, const int* lda,
            int* info);

void blas_zgerc_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const int* m, const int* n,
            const bblas_complex64_t* alpha,
            const bblas_complex64_t** x, const int* incx,
            const bblas_complex64_t** y, const int* incy,
            bblas_complex64_t** a, const int* lda,
            int* info);
```

`blas_xher_batch`

1. Functionalities:
   $A \leftarrow \alpha x x^H + \beta A,\ A \in \mathrm{C}^{n \times n},\ A : \mathrm{Hermite}$

2. Prefixes:
   `c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, incx, lda, group_size`
   Each Job: `x, a`

4. Interfaces:

```
void blas_cher_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const float* alpha,
            const bblas_complex32_t** x, const int* incx,
```

```
                bblas_complex32_t** a, const int* lda,
                int* info);

    void blas_zher_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const int* n,
                const double* alpha,
                const bblas_complex64_t** x, const int* incx,
                bblas_complex64_t** a, const int* lda,
                int* info);
```

`blas_xhpr_batch`

1. Functionalities:
   $A \leftarrow \alpha x x^H + \beta A,\ A \in \mathrm{C}^{n \times n},\ A : \mathrm{Packed\&Hermite}$

2. Prefixes:
   `c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, incx, group_size`
   Each Job: `x, ap`

4. Interfaces:

```
    void blas_chpr_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const int* n,
                const float* alpha,
                const bblas_complex32_t** x, const int* incx,
                bblas_complex32_t** ap,
                int* info);

    void blas_zhpr_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const int* n,
                const double* alpha,
                const bblas_complex64_t** x, const int* incx,
                bblas_complex64_t** ap,
                int* info);
```

`blas_xher2_batch`

1. Functionalities:
   $A \leftarrow \alpha x y^H + y(\alpha x)^H + \beta A,\ A \in \mathrm{C}^{n \times n},\ A : \mathrm{Hermite}$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`
4. Interfaces:

```
void blas_cher2_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** x, const int* incx,
            const bblas_complex32_t** y, const int* incy,
            bblas_complex32_t** a, const int* lda,
            int* info);

void blas_zher2_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const bblas_complex64_t* alpha,
            const bblas_complex64_t** x, const int* incx,
            const bblas_complex64_t** y, const int* incy,
            bblas_complex64_t** a, const int* lda,
            int* info);
```

`blas_xhpr2_batch`

1. Functionalities:
   $A \leftarrow \alpha x y^H + \bar{\alpha} y x^H + \beta A,\ A \in \mathrm{C}^{n \times n},\ A : \mathrm{Packed\&Hermite}$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, incx, incy, group_size`
   Each Job: `x, y, ap`

4. Interfaces:

```
void blas_chpr2_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** x, const int* incx,
            const bblas_complex32_t** y, const int* incy,
            bblas_complex32_t** ap,
            int* info);

void blas_zhpr2_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const bblas_complex64_t* alpha,
            const bblas_complex64_t** x, const int* incx,
            const bblas_complex64_t** y, const int* incy,
            bblas_complex64_t** ap,
            int* info);
```

`blas_xsyr_batch`

1. Functionalities:
   $A \leftarrow \alpha x x^T + \beta A,\ A \in \mathrm{R}^{n \times n},\ A : \mathrm{Symmetric}$

2. Prefixes:
   `s, d`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, incx, lda, group_size`
   Each Job: `x, a`

4. Interfaces:

```
void blas_ssyr_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const float* alpha,
            const float** x, const int* incx,
            float** a, const int* lda,
            int* info);
```

```
void blas_dsyr_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const double* alpha,
            const double** x, const int* incx,
            double** a, const int* lda,
            int* info);
```

`blas_xspr_batch`

1. Functionalities:
   $A \leftarrow \alpha x x^T + \beta A,\ A \in \mathrm{R}^{n \times n},\ A$: Packed&Symmetric
2. Prefixes:
   `s, d`
3. Argument scope:

All:	`layout, group_count`
Group:	`uplo, n, alpha, incx, group_size`
Each Job:	`x, ap`

4. Interfaces:

```
void blas_sspr_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const float* alpha,
            const float** x, const int* incx, float** ap,
            int* info);

void blas_dspr_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const double* alpha,
            const double** x, const int* incx, double** ap,
            int* info);
```

`blas_xsyr2_batch`

1. Functionalities:
   $A \leftarrow \alpha x y^T + \alpha y x^T + \beta A,\ A \in \mathrm{R}^{n \times n},\ A$: Symmetric

2. Prefixes:
   `s, d`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`

4. Interfaces:

```
void blas_ssyr2_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const float* alpha,
            const float** x, const int* incx,
            const float** y, const int* incy,
            float** a, const int* lda,
            int* info);

void blas_dsyr2_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const int* n,
            const double* alpha,
            const double** x, const int* incx,
            const double** y, const int* incy,
            double** a, const int* lda,
            int* info);
```

`blas_xspr2_batch`

1. Functionalities:
   $A \leftarrow \alpha xy^T + \alpha yx^T + \beta A,\ A \in \mathrm{R}^{n \times n},\ A : \mathrm{Packed\&Symmetric}$

2. Prefixes:
   `s, d`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, n, alpha, incx, incy, group_size`
   Each Job: `x, y, ap`

4. Interfaces:

```
void blas_sspr2_batch(
```

```
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const int* n,
                const float* alpha,
                const float** x, const int* incx,
                const float** y, const int* incy,
                float** ap,
                int* info);

    void blas_dspr2_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const int* n,
                const double* alpha,
                const double** x, const int* incx,
                const double** y, const int* incy, double** ap,
                int* info);
```

`blas_xgemv_batchf`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,\ y \leftarrow \alpha A^H x + \beta y,\ A \in \mathrm{K}^{m \times n}$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout`
   Group: `transa, m, n, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`
4. Interfaces:

```
    void blas_sgemv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t transa,
                const int m, const int n,
                const float alpha,
                const float** a, const int lda,
                const float** x, const int incx,
                const float beta,
                float** y, const int incy,
                int* info);

    void blas_dgemv_batchf(
```

```
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t transa,
                const int m, const int n,
                const double alpha,
                const double** a, const int lda,
                const double** x, const int incx,
                const double beta,
                double** y, const int incy,
                int* info);

    void blas_cgemv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t transa,
                const int m, const int n,
                const bblas_complex32_t alpha,
                const bblas_complex32_t** a, const int lda,
                const bblas_complex32_t** x, const int incx,
                const bblas_complex32_t beta,
                bblas_complex32_t** y, const int incy,
                int* info);

    void blas_zgemv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t transa,
                const int m, const int n,
                const bblas_complex64_t alpha,
                const bblas_complex64_t** a, const int lda,
                const bblas_complex64_t** x, const int incx,
                const bblas_complex64_t beta,
                bblas_complex64_t** y, const int incy,
                int* info);
```

`blas_xgbmv_batchf`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,\ y \leftarrow \alpha A^H x + \beta y,\ A \in \mathrm{Band}(\mathrm{kl}, \mathrm{ku})$

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout`
   Group: `trans, m, n, kl, ku, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`

4. Interfaces:

```
void blas_sgbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t trans,
            const int m, const int n, const int kl, const int ku,
            const float alpha,
            const float** a, const int lda,
            const float** x, const int incx,
            const float beta,
            float** y, const int incy,
            int* info);

void blas_dgbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t trans,
            const int m, const int n, const int kl, const int ku,
            const double alpha,
            const double** a, const int lda,
            const double** x, const int incx,
            const double beta,
            double** y, const int incy,
            int* info);

void blas_cgbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t trans,
            const int m, const int n, const int kl, const int ku,
            const bblas_complex32_t alpha,
            const bblas_complex32_t** a, const int lda,
            const bblas_complex32_t** x, const int incx,
            const bblas_complex32_t beta,
            bblas_complex32_t** y, const int incy,
            int* info);

void blas_zgbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t trans,
            const int m, const int n, const int kl, const int ku,
            const bblas_complex64_t alpha,
            const bblas_complex64_t** a, const int lda,
            const bblas_complex64_t** x, const int incx,
            const bblas_complex64_t beta,
            bblas_complex64_t** y, const int incy,
```

```
                int* info);
```

`blas_xhemv_batchf`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Hermite}$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`
4. Interfaces:

```
void blas_chemv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n,
            const bblas_complex32_t alpha,
            const bblas_complex32_t** a, const int lda,
            const bblas_complex32_t** x, const int incx,
            const bblas_complex32_t beta,
            bblas_complex32_t** y, const int incy,
            int* info);

void blas_zhemv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n,
            const bblas_complex64_t alpha,
            const bblas_complex64_t** a, const int lda,
            const bblas_complex64_t** x, const int incx,
            const bblas_complex64_t beta,
            bblas_complex64_t** y, const int incy,
            int* info);
```

`blas_xhbmv_batchf`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Hermit\&Band}(\text{band width} = k)$
2. Prefixes:
   `c, z`

3. Argument scope:
   All: `layout`
   Group: `uplo, n, k, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`

4. Interfaces:

```
void blas_chbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n,
            const int k, const bblas_complex32_t alpha,
            const bblas_complex32_t** a, const int lda,
            const bblas_complex32_t** x, const int incx,
            const bblas_complex32_t beta,
            bblas_complex32_t** y, const int incy,
            int* info);

void blas_zhbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const int k,
            const bblas_complex64_t alpha,
            const bblas_complex64_t** a, const int lda,
            const bblas_complex64_t** x, const int incx,
            const bblas_complex64_t beta,
            bblas_complex64_t** y, const int incy,
            int* info);
```

`blas_xhpmv_batchf`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Packed\&Hermite}$

2. Prefixes:
   `c, z`

3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, beta, incy, group_size`
   Each Job: `ap, x, y`

4. Interfaces:

```
void blas_chpmv_batchf(
            const int group_size,
```

```
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const int n,
                const bblas_complex32_t alpha,
                const bblas_complex32_t** ap,
                const bblas_complex32_t** x, const int incx,
                const bblas_complex32_t beta,
                bblas_complex32_t** y, const int incy,
                int* info);

    void blas_zhpmv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const int n,
                const bblas_complex64_t alpha,
                const bblas_complex64_t** ap,
                const bblas_complex64_t** x, const int incx,
                const bblas_complex64_t beta,
                bblas_complex64_t** y, const int incy,
                int* info);
```

`blas_xsymv_batchf`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Symmetric}$
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`
4. Interfaces:

```
    void blas_ssymv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const int n,
                const float alpha,
                const float** a, const int lda,
                const float** x, const int incx,
                const float beta,
                float** y, const int incy,
                int* info);
```

```
void blas_dsymv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n,
            const double alpha,
            const double** a, const int lda,
            const double** x, const int incx,
            const double beta,
            double** y, const int incy,
            int* info);
```

`blas_xsbmv_batchf`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A : \text{Symmetric\&Band}(\text{band width} = k)$
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, k, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`
4. Interfaces:

```
void blas_ssbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const int k,
            const float alpha,
            const float** a, const int lda,
            const float** x, const int incx,
            const float beta,
            float** y, const int incy,
            int* info);

void blas_dsbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const int k,
            const double alpha,
            const double** a, const int lda,
            const double** x, const int incx,
```

```
                const double beta,
                double** y, const int incy,
                int* info);
```

`blas_xspmv_batchf`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ A$: Packed&Symmetric
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, beta, incy, group_size`
   Each Job: `ap, x, y`
4. Interfaces:

```
void blas_sspmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n,
            const float alpha, const float** ap,
            const float** x, const int incx,
            const float beta,
            float** y, const int incy,
            int* info);

void blas_dspmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n,
            const double alpha, const double** ap,
            const double** x, const int incx,
            const double beta,
            double** y, const int incy,
            int* info);
```

`blas_xtrmv_batchf`

1. Functionalities:
   $x \leftarrow Ax,\ x \leftarrow A^T x,\ x \leftarrow A^H x,\ A \in \mathrm{K}^{n \times n},\ A$: Triangular
2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout`
   Group: `uplo, trans, diag, n, lda, incx, group_size`
   Each Job: `a, x`

4. Interfaces:

```
void blas_strmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n,
            const float** a, const int lda,
            float** x, const int incx,
            int* info);

void blas_dtrmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n,
            const double** a, const int lda,
            double** x, const int incx,
            int* info);

void blas_ctrmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n,
            const bblas_complex32_t** a, const int lda,
            bblas_complex32_t** x, const int incx,
            int* info);

void blas_ztrmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n,
            const bblas_complex64_t** a, const int lda,
            bblas_complex64_t** x, const int incx,
            int* info);
```

`blas_xtbmv_batchf`

1. Functionalities:
   $x \leftarrow Ax$, $x \leftarrow A^T x$, $x \leftarrow A^H x$, $A \in \mathrm{K}^{n \times n}$, A : Band&Triangular(band width $= k$)

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout`
   Group: `uplo, trans, diag, n, k, lda, incx, group_size`
   Each Job: `a, x`

4. Interfaces:

```
void blas_stbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n, const int k,
            const float** a, const int lda,
            float** x, const int incx,
            int* info);

void blas_dtbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n, const int k,
            const double** a, const int lda,
            double** x, const int incx,
            int* info);

void blas_ctbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n, const int k,
            const bblas_complex32_t** a, const int lda,
            bblas_complex32_t** x, const int incx,
            int* info);

void blas_ztbmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
```

```
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n, const int k,
                const bblas_complex64_t** a, const int lda,
                bblas_complex64_t** x, const int incx,
                int* info);
```

`blas_xtpmv_batchf`

1. Functionalities:
   $x \leftarrow Ax,\ x \leftarrow A^T x,\ x \leftarrow A^H x,\ A \in \mathrm{K}^{n \times n},\ A$: Packed&Triangular
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout`
   Group: `uplo, trans, diag, n, incx, group_size`
   Each Job: `ap, x`
4. Interfaces:

```
void blas_stpmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n, const float** ap,
            float** x, const int incx,
            int* info);

void blas_dtpmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n, const double** ap,
            double** x, const int incx,
            int* info);

void blas_ctpmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n, const bblas_complex32_t** ap,
            bblas_complex32_t** x, const int incx,
            int* info);
```

```
void blas_ztpmv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n, const bblas_complex64_t** ap,
            bblas_complex64_t** x, const int incx,
            int* info);
```

`blas_xtrsv_batchf`

1. Functionalities:
   $x \leftarrow A^{-1}x,\ x \leftarrow A^{-T}x,\ x \leftarrow A^{-H}x,\ A \in \mathrm{K}^{n\times n},\ A$: Triangular
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout`
   Group: `uplo, trans, diag, n, lda, incx, group_size`
   Each Job: `a, x`
4. Interfaces:

```
void blas_strsv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n,
            const float** a, const int lda,
            float** x, const int incx,
            int* info);

void blas_dtrsv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
            const int n,
            const double** a, const int lda,
            double** x, const int incx,
            int* info);

void blas_ctrsv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const bblas_enum_t diag,
```

```
                const int n,
                const bblas_complex32_t** a, const int lda,
                bblas_complex32_t** x, const int incx,
                int* info);

    void blas_ztrsv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n,
                const bblas_complex64_t** a, const int lda,
                bblas_complex64_t** x, const int incx,
                int* info);
```

`blas_xtbsv_batchf`

1. Functionalities:
   $x \leftarrow A^{-1}x$, $x \leftarrow A^{-T}x$, $x \leftarrow A^{-H}x$, $A \in \mathrm{K}^{n\times n}$, A : Band&Triangular(band width = k
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout`
   Group: `uplo, trans, diag, n, k, lda, incx, group_size`
   Each Job: `a, x`
4. Interfaces:

```
    void blas_stbsv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n, const int k,
                const float** a, const int lda,
                float** x, const int incx,
                int* info);

    void blas_dtbsv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n, const int k,
                const double** a, const int lda,
```

```
                double** x, const int incx,
                int* info);

    void blas_ctbsv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n, const int k,
                const bblas_complex32_t** a, const int lda,
                bblas_complex32_t** x, const int incx,
                int* info);

    void blas_ztbsv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n, const int k,
                const bblas_complex64_t** a, const int lda,
                bblas_complex64_t** x, const int incx,
                int* info);
```

`blas_xtpsv_batchf`

1. Functionalities:
   $x \leftarrow A^{-1}x,\ x \leftarrow A^{-T}x,\ x \leftarrow A^{-H}x,\ A \in \mathrm{K}^{n\times n},\ A$: Packed&Triangular
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout`
   Group: `uplo, trans, diag, n, incx, group_size`
   Each Job: `ap, x`
4. Interfaces:

```
    void blas_stpsv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n, const float** ap,
                float** x, const int incx,
                int* info);

    void blas_dtpsv_batchf(
```

```
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n, const double** ap,
                double** x, const int incx,
                int* info);

    void blas_ctpsv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n, const bblas_complex32_t** ap,
                bblas_complex32_t** x, const int incx,
                int* info);

    void blas_ztpsv_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const bblas_enum_t diag,
                const int n, const bblas_complex64_t** ap,
                bblas_complex64_t** x, const int incx,
                int* info);
```

`blas_xger_batchf`

1. Functionalities:
   $A \leftarrow \alpha x y^T + \beta A,\ A \in \mathrm{R}^{m \times n}$
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout`
   Group: `m, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`
4. Interfaces:

```
    void blas_sger_batchf(
                const int group_size,
                const bblas_enum_t layout,
                const int m, const int n,
                const float alpha, const float** x, const int incx,
                const float** y, const int incy,
                float** a, const int lda,
```

```
                int* info);

    void blas_dger_batchf(
                const int group_size,
                const bblas_enum_t layout,
                const int m, const int n,
                const double alpha, const double** x, const int incx,
                const double** y, const int incy,
                double** a, const int lda,
                int* info);
```

`blas_xgeru_batchf`

1. Functionalities:
   $A \leftarrow \alpha x y^T + \beta A,\ A \in \mathrm{C}^{m \times n}$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout`
   Group: `m, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`
4. Interfaces:

```
    void blas_cgeru_batchf(
                const int group_size,
                const bblas_enum_t layout,
                const int m, const int n,
                const bblas_complex32_t alpha,
                const bblas_complex32_t** x, const int incx,
                const bblas_complex32_t** y, const int incy,
                bblas_complex32_t** a, const int lda,
                int* info);

    void blas_zgeru_batchf(
                const int group_size,
                const bblas_enum_t layout,
                const int m, const int n,
                const bblas_complex64_t alpha,
                const bblas_complex64_t** x, const int incx,
                const bblas_complex64_t** y, const int incy,
                bblas_complex64_t** a, const int lda,
                int* info);
```

`blas_xgerc_batchf`

1. Functionalities:
   $A \leftarrow \alpha x y^H + \beta A,\ A \in \mathrm{C}^{m \times n}$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout`
   Group: `m, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`
4. Interfaces:

```
void blas_cgerc_batchf(
            const int group_size,
            const bblas_enum_t layout,
            const int m, const int n, const bblas_complex32_t alpha,
            const bblas_complex32_t** x, const int incx,
            const bblas_complex32_t** y, const int incy,
            bblas_complex32_t** a, const int lda,
            int* info);

void blas_zgerc_batchf(
            const int group_size,
            const bblas_enum_t layout,
            const int m, const int n, const bblas_complex64_t alpha,
            const bblas_complex64_t** x, const int incx,
            const bblas_complex64_t** y, const int incy,
            bblas_complex64_t** a, const int lda,
            int* info);
```

`blas_xher_batchf`

1. Functionalities:
   $A \leftarrow \alpha x x^H + \beta A,\ A \in \mathrm{C}^{n \times n},\ A : \mathrm{Hermite}$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, lda, group_size`
   Each Job: `x, a`
4. Interfaces:

```
void blas_cher_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const float alpha,
            const bblas_complex32_t** x, const int incx,
            bblas_complex32_t** a, const int lda,
            int* info);

void blas_zher_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const double alpha,
            const bblas_complex64_t** x, const int incx,
            bblas_complex64_t** a, const int lda,
            int* info);
```

`blas_xhpr_batchf`

1. Functionalities:
   $A \leftarrow \alpha x x^H + \beta A,\ A \in \mathrm{C}^{n \times n},\ A : \mathrm{Packed\&Hermite}$
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, group_size`
   Each Job: `x, ap`
4. Interfaces:

```
void blas_chpr_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const float alpha,
            const bblas_complex32_t** x, const int incx,
            bblas_complex32_t** ap,
            int* info);

void blas_zhpr_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const double alpha,
            const bblas_complex64_t** x, const int incx,
            bblas_complex64_t** ap,
            int* info);
```

`blas_xher2_batchf`

1. Functionalities:
   $A \leftarrow \alpha x y^H + y(\alpha x)^H + \beta A,\ A \in \mathrm{C}^{n \times n},\ A$: Hermite
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`
4. Interfaces:

```
void blas_cher2_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const bblas_complex32_t alpha,
            const bblas_complex32_t** x, const int incx,
            const bblas_complex32_t** y, const int incy,
            bblas_complex32_t** a, const int lda,
            int* info);

void blas_zher2_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const bblas_complex64_t alpha,
            const bblas_complex64_t** x, const int incx,
            const bblas_complex64_t** y, const int incy,
            bblas_complex64_t** a, const int lda,
            int* info);
```

`blas_xhpr2_batchf`

1. Functionalities:
   $A \leftarrow \alpha x y^H + \bar{\alpha} y x^H + \beta A,\ A \in \mathrm{C}^{n \times n},\ A$: Packed&Hermite
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, incy, group_size`
   Each Job: `x, y, ap`
4. Interfaces:

```
void blas_chpr2_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const bblas_complex32_t alpha,
            const bblas_complex32_t** x, const int incx,
            const bblas_complex32_t** y, const int incy,
            bblas_complex32_t** ap,
            int* info);

void blas_zhpr2_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const bblas_complex64_t alpha,
            const bblas_complex64_t** x, const int incx,
            const bblas_complex64_t** y, const int incy,
            bblas_complex64_t** ap,
            int* info);
```

`blas_xsyr_batchf`

1. Functionalities:
   $A \leftarrow \alpha x x^T + \beta A,\ A \in \mathrm{R}^{n \times n},\ A : \mathrm{Symmetric}$
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, lda, group_size`
   Each Job: `x, a`
4. Interfaces:

```
void blas_ssyr_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const float alpha,
            const float** x, const int incx,
            float** a, const int lda,
            int* info);

void blas_dsyr_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const double alpha,
```

```
                const double** x, const int incx,
                double** a, const int lda,
                int* info);
```

`blas_xspr_batchf`

1. Functionalities:
   $A \leftarrow \alpha x x^T + \beta A,\ A \in \mathrm{R}^{n \times n},\ A$: Packed&Symmetric
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, group_size`
   Each Job: `x, ap`
4. Interfaces:

```
void blas_sspr_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const float alpha,
            const float** x, const int incx,
            float** ap,
            int* info);

void blas_dspr_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const double alpha,
            const double** x, const int incx,
            double** ap,
            int* info);
```

`blas_xsyr2_batchf`

1. Functionalities:
   $A \leftarrow \alpha x y^T + \alpha y x^T + \beta A,\ A \in \mathrm{R}^{n \times n},\ A$: Symmetric
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`

4. Interfaces:

```
void blas_ssyr2_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const float alpha,
            const float** x, const int incx,
            const float** y, const int incy,
            float** a, const int lda,
            int* info);

void blas_dsyr2_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const double alpha,
            const double** x, const int incx,
            const double** y, const int incy,
            double** a, const int lda,
            int* info);
```

`blas_xspr2_batchf`

1. Functionalities:
   $A \leftarrow \alpha xy^T + \alpha yx^T + \beta A,\ A \in \mathrm{R}^{n \times n},\ A : \mathrm{Packed\&Symmetric}$
2. Prefixes:
   `s, d`
3. Argument scope:
   All: `layout`
   Group: `uplo, n, alpha, incx, incy, group_size`
   Each Job: `x, y, ap`
4. Interfaces:

```
void blas_sspr2_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const int n, const float alpha,
            const float** x, const int incx,
            const float** y, const int incy,
            float** ap,
            int* info);

void blas_dspr2_batchf(
```

```
const int group_size,
const bblas_enum_t layout, const bblas_enum_t uplo,
const int n, const double alpha,
const double** x, const int incx,
const double** y, const int incy,
double** ap,
int* info);
```

4.2.3 Level 3 BLAS Routines

The interfaces of the Batched BLAS Level 3 BLAS routines are as follows.

`blas_xgemm_batch`

1. Functionalities:
   $C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C,\ \mathrm{op}(X) = \{X, X^T, X^H\},\ C \in \mathrm{K}^{m \times n}$

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `transa, transb, m, n, k, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`

4. Interfaces:

```
void blas_sgemm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* transa,
            const bblas_enum_t* transb, const int* m, const int* n, const int* k,
            const float* alpha,
            const float** a, const int* lda,
            const float** b, const int* ldb,
            const float* beta,
            float** c, const int* ldc,
            int* info);

void blas_dgemm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* transa,
            const bblas_enum_t* transb, const int* m, const int* n, const int* k,
            const double* alpha,
            const double** a, const int* lda,
            const double** b, const int* ldb,
            const double* beta,
            double** c, const int* ldc,
            int* info);

void blas_cgemm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* transa,
            const bblas_enum_t* transb, const int* m, const int* n, const int* k,
```

```
                const bblas_complex32_t* alpha,
                const bblas_complex32_t** a, const int* lda,
                const bblas_complex32_t** b, const int* ldb,
                const bblas_complex32_t* beta,
                bblas_complex32_t** c, const int* ldc,
                int* info);

    void blas_zgemm_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* transa,
                const bblas_enum_t* transb, const int* m, const int* n, const int* k
                const bblas_complex64_t* alpha,
                const bblas_complex64_t** a, const int* lda,
                const bblas_complex64_t** b, const int* ldb,
                const bblas_complex64_t* beta,
                bblas_complex64_t** c, const int* ldc,
                int* info);
```

`blas_xsymm_batch`

1. Functionalities:
   $C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha BA + \beta C,\ C \in \mathrm{K}^{m \times n},\ A : \mathrm{Symmetric}$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `side, uplo, m, n, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`
4. Interfaces:

```
    void blas_ssymm_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* side,
                const bblas_enum_t* uplo, const int* m, const int* n,
                const float* alpha,
                const float** a, const int* lda,
                const float** b, const int* ldb,
                const float* beta,
                float** c, const int* ldc,
                int* info);

    void blas_dsymm_batch(
```

```
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* side,
                const bblas_enum_t* uplo, const int* m, const int* n,
                const double* alpha,
                const double** a, const int* lda,
                const double** b, const int* ldb,
                const double* beta,
                double** c, const int* ldc,
                int* info);

    void blas_csymm_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* side,
                const bblas_enum_t* uplo,
                const int* m, const int* n,
                const bblas_complex32_t* alpha,
                const bblas_complex32_t** a, const int* lda,
                const bblas_complex32_t** b, const int* ldb,
                const bblas_complex32_t* beta,
                bblas_complex32_t** c, const int* ldc,
                int* info);

    void blas_zsymm_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* side,
                const bblas_enum_t* uplo, const int* m, const int* n,
                const bblas_complex64_t* alpha,
                const bblas_complex64_t** a, const int* lda,
                const bblas_complex64_t** b, const int* ldb,
                const bblas_complex64_t* beta,
                bblas_complex64_t** c, const int* ldc,
                int* info);
```

`blas_xhemm_batch`

1. Functionalities:
   $C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha BA + \beta C,\ C \in \mathrm{C}^{m \times n},\ A$: Hermite
2. Prefixes:
   `c, z`
3. Argument scope:
   All: `layout, group_count`
   Group: `side, uplo, m, n, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`

4. Interfaces:

```
void blas_chemm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
            const bblas_enum_t* uplo, const int* m, const int* n,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** a, const int* lda,
            const bblas_complex32_t** b, const int* ldb,
            const bblas_complex32_t* beta,
            bblas_complex32_t** c, const int* ldc,
            int* info);

void blas_zhemm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
            const bblas_enum_t* uplo, const int* m, const int* n,
            const bblas_complex64_t* alpha,
            const bblas_complex64_t** a, const int* lda,
            const bblas_complex64_t** b, const int* ldb,
            const bblas_complex64_t* beta,
            bblas_complex64_t** c, const int* ldc,
            int* info);
```

`blas_xsyrk_batch`

1. Functionalities:
   $C \leftarrow \alpha AA^T + \beta C,\ C \leftarrow \alpha A^T A + \beta C,\ C \in \mathrm{K}^{n \times n},\ A : \mathrm{Symmetric}$

2. Prefixes:
   `s, d, c, z`

3. Argument scope:

All:	`layout, group_count`
Group:	`uplo, trans, n, k, alpha, lda, beta, ldc, group_size`
Each Job:	`a, c`

4. Interfaces:

```
void blas_ssyrk_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const int* n, const int* k,
            const float* alpha,
            const float** a, const int* lda,
```

```
                const float* beta,
                float** c, const int* ldc,
                int* info);

    void blas_dsyrk_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const int* n, const int* k,
                const double* alpha,
                const double** a, const int* lda,
                const double* beta,
                double** c, const int* ldc,
                int* info);

    void blas_csyrk_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const int* n, const int* k,
                const bblas_complex32_t* alpha,
                const bblas_complex32_t** a, const int* lda,
                const bblas_complex32_t* beta,
                bblas_complex32_t** c, const int* ldc,
                int* info);

    void blas_zsyrk_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const int* n, const int* k,
                const bblas_complex64_t* alpha,
                const bblas_complex64_t** a, const int* lda,
                const bblas_complex64_t* beta,
                bblas_complex64_t** c, const int* ldc,
                int* info);
```

`blas_xherk_batch`

1. Functionalities:
   $C \leftarrow \alpha A A^H + \beta C,\ C \leftarrow \alpha A^H A + \beta C,\ C \in \mathrm{C}^{n \times n},\ A : \mathrm{Hermite}$

2. Prefixes:
   `c, z`

3. Argument scope:

All: `layout, group_count`
Group: `uplo, trans, n, k, alpha, lda, beta, ldc, group_size`
Each Job: `a, c`

4. Interfaces:

```
void blas_cherk_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const int* n, const int* k,
            const float* alpha,
            const bblas_complex32_t** a, const int* lda,
            const float* beta,
            bblas_complex32_t** c, const int* ldc,
            int* info);

void blas_zherk_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const int* n, const int* k,
            const double* alpha,
            const bblas_complex64_t** a, const int* lda,
            const double* beta,
            bblas_complex64_t** c, const int* ldc,
            int* info);
```

`blas_xsyr2k_batch`

1. Functionalities:
   $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C,\ C \leftarrow \alpha B^T A + \alpha A^T B + \beta C,\ C \in \mathrm{K}^{n \times n},\ C : \mathrm{Symmetric}$

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, trans, n, k, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`

4. Interfaces:

```
void blas_ssyr2k_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const int* n, const int* k,
            const float* alpha,
```

```
                const float** a, const int* lda,
                const float** b, const int* ldb,
                const float* beta,
                float** c, const int* ldc,
                int* info);

    void blas_dsyr2k_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const int* n, const int* k,
                const double* alpha,
                const double** a, const int* lda,
                const double** b, const int* ldb,
                const double* beta,
                double** c, const int* ldc,
                int* info);

    void blas_csyr2k_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const int* n, const int* k,
                const bblas_complex32_t* alpha,
                const bblas_complex32_t** a, const int* lda,
                const bblas_complex32_t** b, const int* ldb,
                const bblas_complex32_t* beta,
                bblas_complex32_t** c, const int* ldc,
                int* info);

    void blas_zsyr2k_batch(
                const int group_count, const int* group_size,
                const bblas_enum_t layout, const bblas_enum_t* uplo,
                const bblas_enum_t* trans, const int* n, const int* k,
                const bblas_complex64_t* alpha,
                const bblas_complex64_t** a, const int* lda,
                const bblas_complex64_t** b, const int* ldb,
                const bblas_complex64_t* beta,
                bblas_complex64_t** c, const int* ldc,
                int* info);
```

`blas_xher2k_batch`

1. Functionalities:
   $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C,\ C \leftarrow \alpha B^T A + \alpha A^T B + \beta C,\ C \in \mathrm{C}^{n \times n},\ C$: Hermite

2. Prefixes:
   `c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `uplo, trans, n, k, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`

4. Interfaces:

```
void blas_cher2k_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const int* n, const int* k,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** a, const int* lda,
            const bblas_complex32_t** b, const int* ldb,
            const float* beta,
            bblas_complex32_t** c, const int* ldc,
            int* info);

void blas_zher2k_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* uplo,
            const bblas_enum_t* trans, const int* n, const int* k,
            const bblas_complex64_t* alpha,
            const bblas_complex64_t** a, const int* lda,
            const bblas_complex64_t** b, const int* ldb,
            const double* beta,
            bblas_complex64_t** c, const int* ldc,
            int* info);
```

`blas_xtrmm_batch`

1. Functionalities:
   $B \leftarrow \alpha \mathrm{op}(A)B,\ B \leftarrow \alpha B \mathrm{op}(A),\ \mathrm{op}(A) = \{A, A^T, A^H\},\ B \in \mathrm{K}^{m \times n},\ A : \mathrm{Triangular}$

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `side, uplo, transa, diag, m, n, alpha, lda, ldb, group_size`
   Each Job: `a, b,`

4. Interfaces:

```
void blas_strmm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
            const bblas_enum_t* uplo, const bblas_enum_t* transa,
            const bblas_enum_t* diag, const int* m, const int* n,
            const float* alpha,
            const float** a, const int* lda,
            float** b, const int* ldb,
            int* info);

void blas_dtrmm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
            const bblas_enum_t* uplo, const bblas_enum_t* transa,
            const bblas_enum_t* diag, const int* m, const int* n,
            const double* alpha,
            const double** a, const int* lda,
            double** b, const int* ldb,
            int* info);

void blas_ctrmm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
            const bblas_enum_t* uplo, const bblas_enum_t* transa,
            const bblas_enum_t* diag, const int* m, const int* n,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** a, const int* lda,
            bblas_complex32_t** b, const int* ldb,
            int* info);

void blas_ztrmm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
            const bblas_enum_t* uplo, const bblas_enum_t* transa,
            const bblas_enum_t* diag, const int* m, const int* n,
            const bblas_complex64_t* alpha,
            const bblas_complex64_t** a, const int* lda,
            bblas_complex64_t** b, const int* ldb,
            int* info);
```

`blas_xtrsm_batch`

1. Functionalities:
   $B \leftarrow \alpha \mathrm{op}(A)^{-1}B,\ B \leftarrow \alpha B \mathrm{op}(A)^{-1},\ \mathrm{op}(A) = \{A, A^T, A^H\},\ B \in \mathrm{K}^{m \times n},\ A :$

Triangular

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout, group_count`
   Group: `side, uplo, transa, diag, m, n, alpha, lda, ldb, group_size`
   Each Job: `a, b,`

4. Interfaces:

```
void blas_strsm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
            const bblas_enum_t* uplo, const bblas_enum_t* transa,
            const bblas_enum_t* diag, const int* m, const int* n,
            const float* alpha,
            const float** a, const int* lda,
            float** b, const int* ldb,
            int* info);

void blas_dtrsm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
            const bblas_enum_t* uplo, const bblas_enum_t* transa,
            const bblas_enum_t* diag, const int* m, const int* n,
            const double* alpha,
            const double** a, const int* lda,
            double** b, const int* ldb,
            int* info);

void blas_ctrsm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
            const bblas_enum_t* uplo, const bblas_enum_t* transa,
            const bblas_enum_t* diag, const int* m, const int* n,
            const bblas_complex32_t* alpha,
            const bblas_complex32_t** a, const int* lda,
            bblas_complex32_t** b, const int* ldb,
            int* info);

void blas_ztrsm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* side,
```

```
                const bblas_enum_t* uplo, const bblas_enum_t* transa,
                const bblas_enum_t* diag, const int* m, const int* n,
                const bblas_complex64_t* alpha,
                const bblas_complex64_t** a, const int* lda,
                bblas_complex64_t** b, const int* ldb,
                int* info);
```

`blas_xgemm_batchf`

1. Functionalities:
   $C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C,\ \mathrm{op}(X) = \{X, X^T, X^H\},\ C \in \mathrm{K}^{m \times n}$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout`
   Group: `transa, transb, m, n, k, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`
4. Interfaces:

```
void blas_sgemm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t transa,
            const bblas_enum_t transb, const int m, const int n, const int k,
            const float alpha,
            const float** a, const int lda,
            const float** b, const int ldb,
            const float beta,
            float** c, const int ldc,
            int* info);

void blas_dgemm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t transa,
            const bblas_enum_t transb, const int m, const int n, const int k,
            const double alpha,
            const double** a, const int lda,
            const double** b, const int ldb,
            const double beta,
            double** c, const int ldc,
            int* info);

void blas_cgemm_batchf(
```

```
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t transa,
                const bblas_enum_t transb, const int m, const int n, const int k,
                const bblas_complex32_t alpha,
                const bblas_complex32_t** a, const int lda,
                const bblas_complex32_t** b, const int ldb,
                const bblas_complex32_t beta,
                bblas_complex32_t** c, const int ldc,
                int* info);

    void blas_zgemm_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t transa,
                const bblas_enum_t transb, const int m, const int n, const int k,
                const bblas_complex64_t alpha,
                const bblas_complex64_t** a, const int lda,
                const bblas_complex64_t** b, const int ldb,
                const bblas_complex64_t beta,
                bblas_complex64_t** c, const int ldc,
                int* info);
```

`blas_xsymm_batchf`

1. Functionalities:
   $C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha BA + \beta C,\ C \in \mathrm{K}^{m \times n},\ A : \mathrm{Symmetric}$
2. Prefixes:
   `s, d, c, z`
3. Argument scope:
   All: `layout`
   Group: `side, uplo, m, n, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`
4. Interfaces:

```
    void blas_ssymm_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t side,
                const bblas_enum_t uplo, const int m, const int n,
                const float alpha,
                const float** a, const int lda,
                const float** b, const int ldb,
                const float beta,
                float** c, const int ldc,
```

```
                int* info);

    void blas_dsymm_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t side,
                const bblas_enum_t uplo, const int m, const int n,
                const double alpha,
                const double** a, const int lda,
                const double** b, const int ldb,
                const double beta,
                double** c, const int ldc,
                int* info);

    void blas_csymm_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t side,
                const bblas_enum_t uplo, const int m, const int n,
                const bblas_complex32_t alpha,
                const bblas_complex32_t** a, const int lda,
                const bblas_complex32_t** b, const int ldb,
                const bblas_complex32_t beta,
                bblas_complex32_t** c, const int ldc,
                int* info);

    void blas_zsymm_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t side,
                const bblas_enum_t uplo, const int m, const int n,
                const bblas_complex64_t alpha,
                const bblas_complex64_t** a, const int lda,
                const bblas_complex64_t** b, const int ldb,
                const bblas_complex64_t beta,
                bblas_complex64_t** c, const int ldc,
                int* info);
```

`blas_xhemm_batchf`

1. Functionalities:
   $C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha BA + \beta C,\ C \in \mathrm{C}^{m \times n},\ A$: Hermite
2. Prefixes:
   `c, z`
3. Argument scope:

All: `layout`
Group: `side, uplo, m, n, alpha, lda, ldb, beta, ldc, group_size`
Each Job: `a, b, c`

4. Interfaces:

```
void blas_chemm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side,
            const bblas_enum_t uplo, const int m, const int n,
            const bblas_complex32_t alpha,
            const bblas_complex32_t** a, const int lda,
            const bblas_complex32_t** b, const int ldb,
            const bblas_complex32_t beta,
            bblas_complex32_t** c, const int ldc,
            int* info);

void blas_zhemm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side,
            const bblas_enum_t uplo, const int m, const int n,
            const bblas_complex64_t alpha,
            const bblas_complex64_t** a, const int lda,
            const bblas_complex64_t** b, const int ldb,
            const bblas_complex64_t beta,
            bblas_complex64_t** c, const int ldc,
            int* info);
```

`blas_xsyrk_batchf`

1. Functionalities:
   $C \leftarrow \alpha AA^T + \beta C,\ C \leftarrow \alpha A^T A + \beta C,\ C \in \mathrm{K}^{n \times n},\ A : \mathrm{Symmetric}$

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout`
   Group: `uplo, trans, n, k, alpha, lda, beta, ldc, group_size`
   Each Job: `a, c`

4. Interfaces:

```
void blas_ssyrk_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
```

```
                const bblas_enum_t trans, const int n, const int k,
                const float alpha,
                const float** a, const int lda,
                const float beta,
                float** c, const int ldc,
                int* info);

    void blas_dsyrk_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const int n, const int k,
                const double alpha,
                const double** a, const int lda,
                const double beta,
                double** c, const int ldc,
                int* info);

    void blas_csyrk_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const int n, const int k,
                const bblas_complex32_t alpha,
                const bblas_complex32_t** a, const int lda,
                const bblas_complex32_t beta,
                bblas_complex32_t** c, const int ldc,
                int* info);

    void blas_zsyrk_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const int n, const int k,
                const bblas_complex64_t alpha,
                const bblas_complex64_t** a, const int lda,
                const bblas_complex64_t beta,
                bblas_complex64_t** c, const int ldc,
                int* info);
```

`blas_xherk_batchf`

1. Functionalities:
   $C \leftarrow \alpha A A^H + \beta C,\ C \leftarrow \alpha A^H A + \beta C,\ C \in \mathrm{C}^{n \times n},\ A : \mathrm{Hermite}$

2. Prefixes:
   `c, z`

3. Argument scope:
   All: `layout`
   Group: `uplo, trans, n, k, alpha, lda, beta, ldc, group_size`
   Each Job: `a, c`

4. Interfaces:

```
void blas_cherk_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const int n, const int k,
            const float alpha,
            const bblas_complex32_t** a, const int lda, //
            const float beta,
            bblas_complex32_t** c, const int ldc,
            int* info);

void blas_zherk_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const int n, const int k,
            const double alpha,
            const bblas_complex64_t** a, const int lda,
            const double beta,
            bblas_complex64_t** c, const int ldc,
            int* info);
```

`blas_xsyr2k_batchf`

1. Functionalities:
   $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C,\ C \leftarrow \alpha B^T A + \alpha A^T B + \beta C,\ C \in \mathrm{K}^{n \times n},\ C : \mathrm{Symmetric}$

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout`
   Group: `uplo, trans, n, k, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`

4. Interfaces:

```
void blas_ssyr2k_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const int n, const int k,
```

```
                const float alpha,
                const float** a, const int lda,
                const float** b, const int ldb,
                const float beta,
                float** c, const int ldc,
                int* info);

    void blas_dsyr2k_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const int n, const int k,
                const double alpha,
                const double** a, const int lda,
                const double** b, const int ldb,
                const double beta,
                double** c, const int ldc,
                int* info);

    void blas_csyr2k_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const int n, const int k,
                const bblas_complex32_t alpha,
                const bblas_complex32_t** a, const int lda,
                const bblas_complex32_t** b, const int ldb,
                const bblas_complex32_t beta,
                bblas_complex32_t** c, const int ldc,
                int* info);

    void blas_zsyr2k_batchf(
                const int group_size,
                const bblas_enum_t layout, const bblas_enum_t uplo,
                const bblas_enum_t trans, const int n, const int k,
                const bblas_complex64_t alpha,
                const bblas_complex64_t** a, const int lda,
                const bblas_complex64_t** b, const int ldb,
                const bblas_complex64_t beta,
                bblas_complex64_t** c, const int ldc,
                int* info);
```

`blas_xher2k_batchf`

1. Functionalities:
   $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C,\ C \leftarrow \alpha B^T A + \alpha A^T B + \beta C,\ C \in \mathrm{C}^{n \times n},\ C$: Hermite

2. Prefixes:
   `c, z`

3. Argument scope:
   All: `layout`
   Group: `uplo, trans, n, k, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`

4. Interfaces:

```
void blas_cher2k_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const int n, const int k,
            const bblas_complex32_t alpha,
            const bblas_complex32_t** a, const int lda,
            const bblas_complex32_t** b, const int ldb,
            const float beta,
            bblas_complex32_t** c, const int ldc,
            int* info);

void blas_zher2k_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t uplo,
            const bblas_enum_t trans, const int n, const int k,
            const bblas_complex64_t alpha,
            const bblas_complex64_t** a, const int lda,
            const bblas_complex64_t** b, const int ldb,
            const double beta,
            bblas_complex64_t** c, const int ldc,
            int* info);
```

`blas_xtrmm_batchf`

1. Functionalities:
   $B \leftarrow \alpha \mathrm{op}(A)B,\ B \leftarrow \alpha B \mathrm{op}(A),\ \mathrm{op}(A) = \{A, A^T, A^H\},\ B \in \mathrm{K}^{m \times n},\ A : \mathrm{Triangular}$

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout`
   Group: `side, uplo, transa, diag, m, n, alpha, lda, ldb, group_size`
   Each Job: `a, b,`

4. Interfaces:

```
void blas_strmm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side,
            const bblas_enum_t uplo, const bblas_enum_t transa, //
            const bblas_enum_t diag, const int m, const int n,
            const float alpha,
            const float** a, const int lda,
            float** b, const int ldb,
            int* info);

void blas_dtrmm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side,
            const bblas_enum_t uplo, const bblas_enum_t transa,
            const bblas_enum_t diag, const int m, const int n,
            const double alpha,
            const double** a, const int lda,
            double** b, const int ldb,
            int* info);

void blas_ctrmm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side, //
            const bblas_enum_t uplo, const bblas_enum_t transa,
            const bblas_enum_t diag, const int m, const int n,
            const bblas_complex32_t alpha,
            const bblas_complex32_t** a, const int lda,
            bblas_complex32_t** b, const int ldb,
            int* info);

void blas_ztrmm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side,
            const bblas_enum_t uplo, const bblas_enum_t transa,
            const bblas_enum_t diag, const int m, const int n,
            const bblas_complex64_t alpha,
            const bblas_complex64_t** a, const int lda,
            bblas_complex64_t** b, const int ldb,
            int* info);
```

`blas_xtrsm_batchf`

1. Functionalities:
   $B \leftarrow \alpha \mathrm{op}(A)^{-1}B,\ B \leftarrow \alpha B \mathrm{op}(A)^{-1},\ \mathrm{op}(A) = \{A, A^T, A^H\},\ B \in \mathrm{K}^{m \times n},\ A :$

Triangular

2. Prefixes:
   `s, d, c, z`

3. Argument scope:
   All: `layout`
   Group: `side, uplo, transa, diag, m, n, alpha, lda, ldb, group_size`
   Each Job: `a, b,`

4. Interfaces:

```
void blas_strsm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side,
            const bblas_enum_t uplo, const bblas_enum_t transa,
            const bblas_enum_t diag, const int m, const int n,
            const float alpha,
            const float** a, const int lda,
            float** b, const int ldb,
            int* info);

void blas_dtrsm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side,
            const bblas_enum_t uplo, const bblas_enum_t transa,
            const bblas_enum_t diag, const int m, const int n,
            const double alpha,
            const double** a, const int lda,
            double** b, const int ldb,
            int* info);

void blas_ctrsm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side,
            const bblas_enum_t uplo, const bblas_enum_t transa,
            const bblas_enum_t diag, const int m, const int n,
            const bblas_complex32_t alpha,
            const bblas_complex32_t** a, const int lda,
            bblas_complex32_t** b, const int ldb,
            int* info);

void blas_ztrsm_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t side,
```

```
const bblas_enum_t uplo, const bblas_enum_t transa,
const bblas_enum_t diag, const int m, const int n,
const bblas_complex64_t alpha,
const bblas_complex64_t** a, const int lda,
bblas_complex64_t** b, const int ldb,
int* info);
```

Chapter 5

Summary

This document summarizes a guidance of the Batched BLAS from installation to general usage. The developers have been involved in not only the simple implementation but some useful internal mechanisms and enhancements to the FP16 format.

The number of available functions is beyond two hundred or more. However, each function's core part is generated from a few templates, and we believe in extending the Batched-X approach to any thread-safe kernels. Of course, it should go without saying that BLAS performs well in a many-core environment, but we hope that Batched BLAS offers more variety application fields on many-core systems, also it can help in the challenging endeavor.

Appendices

Appendix A

Examples of Compiler-specific Makefiles

A.1 Makefile for Fujitsu compiler

If you use the Fujitsu compiler `fccpx` as a cross-compiler on Fugaku or its compatible system like a PRIMEHPC FX1000, you will likely use `SSL2` or `SSL2BLAMP`. In such cases, you need to change the Makefile at compile time as follows. (1) is for building libraries, and (2) is for making applications.

(1) Example of Makefile in `bblas_src`

```
CC=fccpx
CCFLAG=-Kfast,ocl,openmp -Nclang -D_CBLAS_ -I./
```

(2) Example of Makefile for compiling applications

```
CC=fccpx
CCFLAG=-Kfast,ocl,openmp -Nclang -D_CBLAS_ -I./

DEF = -I../bblas_src
LIB = -L../bblas_src -lbblas
BLAS=-SSL2 -lm
LD=$(CC)
LDFLAG=$(CCFLAG) $(LIB) $(BLAS)
```

If you would use a thread-parallel BLAS library on your application code, use `-SSL2BLAMP` option. See more details on the Fujitsu library manuals [6].

A.2 Makefile for ARM clang compiler

If you use `armclang` as a compiler, you will likely use Arm Performance Library (ArmPL). In the case, you need to change the Makefile at compile time as follows. (1) is for building libraries, and (2) is for making applications. Here, we suppose to compile on an A64FX compatible system.

(1) Example of Makefile in `bblas_src`

```
CC=armclang
CCFLAG=-O3 -fopenmp -Wall -armpl=sve -mcpu=a64fx -D_CBLAS_ -I./
```

(2) Example of Makefile for compiling applications

```
CC=armclang
CCFLAG=-O3 -fopenmp -Wall -armpl=sve -mcpu=afx64 -D_CBLAS_ -I./

DEF = -I../bblas_src
LIB = -L../bblas_src -lbblas
BLAS= -armpl -lm
LD=$(CC)
LDFLAG=$(CCFLAG) $(LIB) $(BLAS)
```

If you would use a thread-parallel ArmPL [7] on your application code, use `-armpl=sve+mp` option.

A.3 Makefile for GNU C compiler gcc

If you use gcc as a compiler on an Intel x86 system, you will likely use cblas, such as the one published in netlib or OSS like OpenBLAS, and so on. In such cases, you need to change the Makefile at compile time as follows. (1) is for building libraries, and (2) is for making applications, here, you suppose that cblas and blas are deployed in lapack that are installed on your home directory.

(1) Example of Makefile in bblas_src

```
CC=gcc
CCFLAG=-O3 -fopenmp -Wall -mcpu=native -D_CBLAS_ -I./
CCFLAG+= -I$(HOME)/lapack/CBLAS/include
```

(2) Example of Makefile for compiling applications

```
CC=gcc
CCFLAG=-O3 -fopenmp -Wall -mcpu=native -D_CBLAS_ -I./

DEF = -I../bblas_src -I$(HOME)/lapack/CBLAS/include
LIB = -L../bblas_src -lbblas
BLAS=-L$(HOME)/lapack -lcblas -lrefblas -lgfortran -lm
LD=$(CC)
LDFLAG=$(CCFLAG) $(LIB) $(BLAS)
```

For other machines or platforms, please adjust the compiler options and the corresponding options to the BLAS libraries on your systems correctly.

Appendix B

APIs for FP16-enhanced BLAS

B.1 Fujitsu SSL-II

Fujitsu SSL-II on the A64FX environment [6] supports FP16 BLAS kernels partially. The Batched BLAS also provides supports for the FP16 enhancement. To validate an FP16 datatype on the Fujitsu HPC programming environment, user has to use the Fujitsu C/C++ compiler with the Clang mode, specifically `fcc -Nclang`. In addition, code generation phase has to be specialized for the Fujitsu SSL-II environment, thus, do as follows instead of the original procedures.

```
$ cd batched_blas_tool
$ python bblas.py data/bblas_data_ssl2.csv
$ CC=fccpx make lib
```

Or, if you compile with the default settings, please do a script file `make_fugaku.sh`.

```
$ cd batched_blas_tool
$ /bin/sh ./make_fugaku.sh
```

The Fujitsu compiler allows us to access FP16 data format with the data type `__fp16` based on ARM ACLE. However, there are several dialects to present the FP16 format, for example, ARM ACLE supports IEEE 754-2008 `__fp16`, ISO/IEC TS 18661-3:2015 recommends `_Float16` for portability, and LLVM uses `half`, or some compilers support `short float`, and so on. To relief such literal differences in data type naming among compiler environments, the current Batched BLAS adopts the typedef-ed structure `fp16` defined as follows.

```
// for Fujitsu SSL-II, and Clang compiler
typedef __fp16  fp16;
```

B.2 Arm Performance Library

ARM Performance Library (ArmPL [7]) supports `cblas_hgemm`. The Batched BLAS follows the support on the FP16 enhancement of the HGEMM kernel. Thus, `cblas_hgemm_batch`

and cblas_hgemm_batchf are available in the current Batched BLAS. In addition, code generation phase has to be specialized for the ARM Performance Library environment, thus, do as follows instead of the original procedures.

```
$ cd batched_blas_tool
$ python bblas.py data/bblas_data_armpl.csv
$ CC=armclang make lib
```

Or, proceed as follows,

```
$ cd batched_blas_tool
$ /bin/sh ./make_armpl.sh
```

As mentioned in the previous section, data-type for FP16 corresponds to __fp16.

```
// for ARM Performance Library
typedef __fp16  fp16;
```

B.3 Specification of FP16-enhancements

In the description, the prefix letter indicates h in the function name, that represents half-precision floating-point number equaling FP16.

B.3.1 F16-enhanced Level 1 BLAS Routines

blas_hswap_batch

1. Functionalities:
   $x \leftrightarrow y$

2. Argument scope:
   All: group_count
   Group: n, incx, incy, group_size
   Each Job: x, y

3. Interfaces:

```
void blas_hswap_batch(
            const int group_count, const int* group_size,
            const int* n,
            fp16** x, const int* incx,
            fp16** y, const int* incy,
            int* info);
```

`blas_hscal_batch`

1. Functionalities:
   $x \leftarrow \alpha x$

2. Argument scope:
   All: `group_count`
   Group: `n, a, incx, group_size`
   Each Job: `x`

3. Interfaces:

```
void blas_hscal_batch(
            const int group_count, const int* group_size,
            const int* n,
            const fp16* a, fp16** x, const int* incx,
            int* info);
```

`blas_hcopy_batch`

1. Functionalities:
   $y \leftarrow x$

2. Argument scope:
   All: `group_count`
   Group: `n, incx, incy, group_size`
   Each Job: `x, y`

3. Interfaces:

```
void blas_hcopy_batch(
            const int group_count, const int* group_size,
            const int* n,
            const fp16** x, const int* incx,
            fp16** y, const int* incy,
            int* info);
```

`blas_haxpy_batch`

1. Functionalities:
   $y \leftarrow \alpha x + y$

2. Argument scope:
   All: `group_count`
   Group: `n, a, incx, incy, group_size`
   Each Job: `x, y`

3. Interfaces:

```
void blas_haxpy_batch(
            const int group_count, const int* group_size,
            const int* n,
            const fp16* a,
            const fp16** x, const int* incx,
            fp16** y, const int* incy,
            int* info);
```

`blas_hdot_batch`

1. Functionalities:
   `dot` $\leftarrow x^T y$

2. Argument scope:

All:	`group_count`
Group:	`n, incx, incy, group_size`
Each Job:	`x, y, ret_cblas_xdot`
NOTE:	return values are stored in `ret_cblas_xdot`

3. Interfaces:

```
void blas_hdot_batch(
            const int group_count, const int* group_size,
            const int* n,
            const fp16** x, const int* incx,
            const fp16** y, const int* incy,
            fp16* ret_cblas_sdot,
            int* info);
```

`blas_hasum_batch`

1. Functionalities:
   `asum` $\leftarrow \|\mathrm{Re}(x)\|_1 + \|\mathrm{Im}(x)\|_1$

2. Argument scope:

All:	`group_count`
Group:	`n, incx, group_size`
Each Job:	`x, ret_cblas_xasum`
NOTE:	return values are stored in `ret_cblas_xasum`

3. Interfaces:

```
void blas_hasum_batch(
            const int group_count, const int* group_size,
```

```
                const int* n,
                const fp16** x, const int* incx,
                fp16* ret_cblas_sasum,
                int* info);
```

`blas_ihamax_batch`

1. Functionalities:
   amax $\leftarrow$ 1st $k \ni |\mathrm{Re}(x_k)| + |\mathrm{Im}(x_k)| = \max_i(|\mathrm{Re}(x_i)| + |\mathrm{Im}(x_i)|)$
2. Argument scope:
   All: `group_count`
   Group: `n, incx, group_size`
   Each Job: `x, ret_cblas_ixamax`
   NOTE: return values are stored in `ret_cblas_ixamax`
3. Interfaces:

```
void blas_ihamax_batch(
                const int group_count, const int* group_size,
                const int* n,
                const fp16** x, const int* incx,
                int* ret_cblas_isamax,
                int* info);
```

`blas_hswap_batchf`

1. Functionalities:
   $x \leftrightarrow y$
2. Argument scope:
   Group: `n, incx, incy, group_size`
   Each Job: `x, y`
3. Interfaces:

```
void blas_hswap_batchf(
                const int group_size,
                const int n,
                fp16** x, const int incx, fp16** y, const int incy,
                int* info);
```

`blas_hscal_batchf`

1. Functionalities:
   $x \leftarrow \alpha x$

2. Argument scope:
   Group: `n, a, incx, group_size`
   Each Job: `x`
3. Interfaces:

```
void blas_hscal_batchf(
            const int group_size,
            const int n,
            const fp16 a1, fp16** x, const int incx,
            int* info);
```

`blas_hcopy_batchf`

1. Functionalities:
   $y \leftarrow x$
2. Argument scope:
   Group: `n, incx, incy, group_size`
   Each Job: `x, y`
3. Interfaces:

```
void blas_hcopy_batchf(
            const int group_size,
            const int n,
            const fp16** x, const int incx,
            fp16** y, const int incy,
            int* info);
```

`blas_haxpy_batchf`

1. Functionalities:
   $y \leftarrow \alpha x + y$
2. Argument scope:
   Group: `n, a1, incx, incy, group_size`
   Each Job: `x, y`
3. Interfaces:

```
void blas_haxpy_batchf(
            const int group_size,
            const int n,
            const fp16 a1,
            const fp16** x, const int incx,
            fp16** y, const int incy,
            int* info);
```

`blas_hdot_batchf`

1. Functionalities:
   `dot` $\leftarrow x^T y$
2. Argument scope:
   Group: `n, incx, incy, group_size`
   Each Job: `x, y, ret_cblas_hdot`
   NOTE: return values are stored in `ret_cblas_hdot`
3. Interfaces:

```
void blas_hdot_batchf(
            const int group_size,
            const int n,
            const fp16** x, const int incx, const fp16** y, const int incy,
            fp16* ret_cblas_hdot,
            int* info);
```

`blas_hasum_batchf`

1. Functionalities:
   `asum` $\leftarrow \|\mathrm{Re}(x)\|_1 + \|\mathrm{Im}(x)\|_1$
2. Argument scope:
   Group: `n, incx, group_size`
   Each Job: `x, ret_cblas_hasum`
   NOTE: return values are stored in `ret_cblas_hasum`
3. Interfaces:

```
void blas_hasum_batchf(
            const int group_size,
            const int n,
            const fp16** x, const int incx,
            fp16* ret_cblas_hasum,
            int* info);
```

`blas_ihamax_batchf`

1. Functionalities:
   `amax` $\leftarrow 1\mathrm{st}\ k \ni |\mathrm{Re}(x_k)| + |\mathrm{Im}(x_k)| = \max_i(|\mathrm{Re}(x_i)| + |\mathrm{Im}(x_i)|)$
2. Argument scope:
   Group: `n, incx, group_size`
   Each Job: `x, ret_cblas_ihamax`
   NOTE: return values are stored in `ret_cblas_ihamax`

3. Interfaces:

```
void blas_ihamax_batchf(
            const int group_size,
            const int n,
            const fp16** x, const int incx,
            int* ret_cblas_ihamax,
            int* info);
```

B.3.2 FP16-enhanced Level 2 BLAS Routines

`blas_hgemv_batch`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,\ y \leftarrow \alpha A^H x + \beta y,\ A \in \mathrm{K}^{m \times n}$

2. Argument scope:
   All: `layout, group_count`
   Group: `transa, m, n, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`

3. Interfaces:

```
void blas_hgemv_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout, const bblas_enum_t* transa,
            const int* m, const int* n,
            const fp16* alpha,
            const fp16** a, const int* lda,
            const fp16** x, const int* incx,
            const fp16* beta,
            fp16** y, const int* incy,
            int* info);
```

`blas_hger_batch`

1. Functionalities:
   $A \leftarrow \alpha xy^T + \beta A,\ A \in \mathrm{R}^{m \times n}$

2. Argument scope:
   All: `layout, group_count`
   Group: `m, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`

3. Interfaces:

```
void blas_hger_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout,
            const int* m, const int* n,
            const fp16* alpha,
            const fp16** x, const int* incx,
            const fp16** y, const int* incy,
            fp16** a, const int* lda,
            int* info);
```

`blas_hgemv_batchf`

1. Functionalities:
   $y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,\ y \leftarrow \alpha A^H x + \beta y,\ A \in \mathrm{K}^{m \times n}$

2. Argument scope:
   All: `layout`
   Group: `transa, m, n, alpha, lda, incx, beta, incy, group_size`
   Each Job: `a, x, y`

3. Interfaces:

```
void blas_hgemv_batchf(
            const int group_size,
            const bblas_enum_t layout, const bblas_enum_t transa,
            const int m, const int n,
            const fp16 alpha,
            const fp16** a, const int lda,
            const fp16** x, const int incx,
            const fp16 beta,
            fp16** y, const int incy,
            int* info);
```

`blas_hger_batchf`

1. Functionalities:
   $A \leftarrow \alpha xy^T + \beta A,\ A \in \mathrm{R}^{m \times n}$

2. Argument scope:
   All: `layout`
   Group: `m, n, alpha, incx, incy, lda, group_size`
   Each Job: `x, y, a`

3. Interfaces:

```
void blas_hger_batchf(
            const int group_size,
            const bblas_enum_t layout,
            const int m, const int n,
            const fp16 alpha, const fp16** x, const int incx,
            const fp16** y, const int incy,
            fp16** a, const int lda,
            int* info);
```

B.3.3 F16-enhanced Level 3 BLAS Routines

`blas_hgemm_batch`

1. Functionalities:
   $C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C,\ \mathrm{op}(X) = \{X, X^T, X^H\},\ C \in \mathrm{K}^{m \times n}$

2. Argument scope:
   All: `layout, group_count`
   Group: `transa, transb, m, n, k, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`

3. Interfaces:

```
void blas_hgemm_batch(
            const int group_count, const int* group_size,
            const bblas_enum_t layout,
            const bblas_enum_t* transa, const bblas_enum_t* transb,
            const int* m, const int* n, const int* k,
            const fp16* alpha,
            const fp16** a, const int* lda,
            const fp16** b, const int* ldb,
            const fp16* beta,
            fp16** c, const int* ldc,
            int* info);
```

`blas_hgemm_batchf`

1. Functionalities:
   $C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C,\ \mathrm{op}(X) = \{X, X^T, X^H\},\ C \in \mathrm{K}^{m \times n}$

2. Argument scope:
   All: `layout`
   Group: `transa, transb, m, n, k, alpha, lda, ldb, beta, ldc, group_size`
   Each Job: `a, b, c`

3. Interfaces:

```
void blas_hgemm_batchf(
            const int group_size,
            const bblas_enum_t layout,
            const bblas_enum_t transa, const bblas_enum_t transb,
            const int m, const int n, const int k,
            const fp16 alpha,
            const fp16** a, const int lda,
            const fp16** b, const int ldb,
            const fp16 beta,
            fp16** c, const int ldc,
            int* info);
```

Appendix C

Release Notes

C.1 Version 1.0

Version 1.0 has been released on February 2021.

Acknowledgements

We would like to thank all the members of RIKEN Center for Computational Science (R-CCS), the Batched BLAS developper group from the Flagship 2020 project. In addition, we would like to acknowledge the DL4Fugaku project for their fruitful discussion and feedbacks.

The Batched BLAS project has been supported by the following funds.

1. Flagship 2020 (post-K) project (FY2014–2020)
2. Computational resources supported by the Grant-in-aid for the K computer, so called 「京調整高度化枠」 in Japanese originally, poroject ID ra000005 (FY2012–2019)
3. Computational resources for the pre-operation of the supercomputer Fugaku, so called 「富岳共用前評価環境」 in Japanese originally, poroject ID ra000006 (FY2020)

References

[1] Sven J. Hammarling: "Standardization of the Batched BLAS," SIAM-CSE19 Minisymposium, Tuesday February 26, 2019 in Spokane, WA, USA, http://icl.utk.edu/bblas/siam-cse19/files/01-SVEN_SIAM_CSE19.pdf

[2] Jack Dongarra, Iain Duff, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jonathan Hogg, Pedro Valero Lara, Piotr Luszczek, Mawussi Zounon, Samuel D. Relton, Stanimire Tomov, Timothy Costa, and Sarah Knepper: "Batched BLAS (Basic Linear Algebra Subprograms) 2018 Specification," July 2018, https://www.icl.utk.edu/files/publications/2018/icl-utk-1170-2018.pdf

[3] Yusuke Hirota, Daichi Mukunoki, and Toshiyuki Imamura: "Automatic Generation ofFull-Set Batched BLAS," Research Poster, International Supercomputing Conference(ISC'18), June 26, 2018 in Frankfurt, Germany, https://2018.isc-program.com/presentation/?id=post129&sess=sess113

[4] BLAS (Basic Linear Algebra Subprograms), http://www.netlib.org/blas/

[5] Reference API of Batched BLAS from NLAFET project, https://github.com/NLAFET/BBLAS

[6] FUJITSU Software Technical Computing Suite V4.0L20, Development Studio, BLAS LAPACK ScaLAPACK, User's Guide, J2UL-2575-01ENZ0(01), June 2020

[7] Arm Performance Libraries Reference Guide 2020-00, 25 June 2020, https://developer.arm.com/documentation/101004/2020