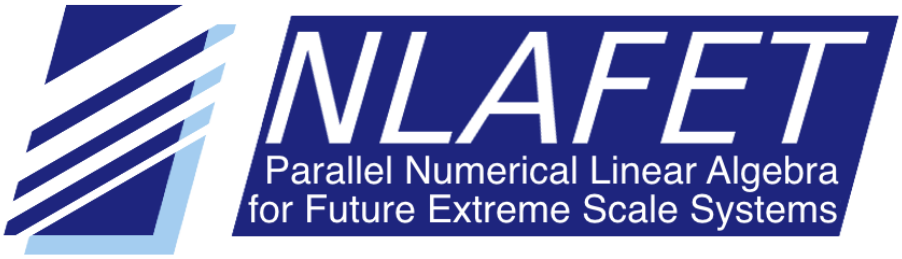

H2020–FETHPC–2014: GA 671633

D2.4

Batched BLAS 2018 Reference Implementation

October 2018

Document information

Scheduled delivery	2018-10-31
Actual delivery	2018-10-30
Version	1.1
Responsible partner	UNIMAN

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2018-10-27	Srikara Pranesh	Draft	1.1	Incorporation of reviewers comments.
2018 -10-25	Mawussi Zounon	Draft	0.1	Initial version of document produced.

Authors

Jack Dongarra (UNIMAN)
Sven Hammarling (UNIMAN)
Nicholas J. Higham (UNIMAN)
Srikara Pranesh (UNIMAN)
Mawussi Zounon (UNIMAN)

Internal reviewers

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

Copyright

Acknowledgements

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

Table of Contents

1 Introduction

D2.4: "Final implementation and evaluation of different versions of the BLAS based on the final specification from WP7"

This deliverable reports on the reference implementation of the Batched BLAS standard specification. A current trend in high-performance computing is to decompose a large linear algebra problem into batches containing thousands of smaller problems, which can be solved independently, before collating the results. To standardize the interface to these routines, the community developed an extension to the BLAS standard (the batched BLAS), enabling users to perform thousands of small BLAS operations in parallel whilst making efficient use of their hardware. The outcome of these efforts led to the Batched BLAS standard specification in the deliverable D7.6 reported at M33. The initial aim of this deliverable is to provide details on the final prototype implementation of such a standard and evaluate different versions of the Batched BLAS based on the final specification. However, we would like to emphasize that even though vendors such as Intel and NVIDIA were involved in the standardization process, at the time of writing this report, the optimized batched BLAS routines available in their libraries are not yet updated to meet the standard specifications. Therefore, we have limited the scope this deliverable to the evaluation of the correctness of the the reference implementation, and its conformity to the standard specification.

In Section 2 we provide implementation details and explain important design decisions. In Section 3 we describe the process to compile the software and get both the html and LaTeXdocumentation. Section 4 is dedicated to testing routines associated with the reference implementation before concluding remarks in Section 5.

2 Implementation details

Batched BLAS routines perform a sequence of independent BLAS operations on a large set of small matrices. So a typical example might be to perform

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

where *batch_count* is large, but A_i, B_i and C_i are small matrices.

The specifications for the BLAS operations have been very successful in providing a standard for vector [5], matrix-vector[4, 3] and matrix-matrix [1, 2] operations respectively, with the associated reference implementation [1]. Therefore, we want the Batched BLAS reference implementation to be based, at low level on calls to the reference BLAS routines.

While many promising APIs have been considered for specification, a consensus emerged around a group API. A group is a collection of BLAS operations for which the matrix dimensions, scalars, and other problem characteristics are identical. Within a group, the only variation between problems to be performed in the batched call are the matrix entries. The group API then accepts multiple groups per function call. For this reason, before implementing the group API, we found it necessary to develop an intermediary API called fixed API to operate on a batch of BLAS routines for which the matrix dimensions, scalars, and other problem characteristics are identical. These routines are implemented in folder *BBLAS/core* available on the NLAFET GitHub[2]. The reference implementation is then

[1] http://www.netlib.org/blas
[2] https://github.com/NLAFET/BBLAS

an iteration over the groups and calling the fixed size APIs and it is available in the folder *BBLAS/compute*.

Although only floating point double precision complex type files are available on the source files, note that we support all the four precisions. The other precisions are automatically generated during the compilation.

3 Compilation and documentation

The reference implementation is delivered along with both *make.inc* and Makefile. The *make.inc* is a configuration file to let the user specify his compiler along with the compilation flags and his choice of the BLAS library. An initial configuration is provide using the GNU Collection Compiler (gcc) and the Intel Math Kernel Library (Intel MKL) as a BLAS library. To get an html and LaTeXdocumentation, the *doxygen* library is required.

After the configuration of make.inc, the compilation is very simple:

- `make [all] - - make lib test`
- `make lib - - make lib/libbblas.a,so lib/libcore.a,so`
- `make test - - make test/test`
- `make docs - - make docs/html`
- `make generate - - generate precisions`
- `make clean - - remove objects, libraries, and executables`
- `make cleangen - - remove generated precision files`
- `make distclean - - remove above, Makefile.*.gen, and anything else that can be generated`

For MacOS the libraries have to be linked manually. Use the following in the .profile file.
export $DYLD_FALLBACK_LIBRARY_PATH=
/path/to/BBLAS/lib:$DYLD_FALLBACK_LIBRARY_PATH

4 Testing

At the end of the compilation, testing routines (in four precisions [c,d,s,z]) will be available in the folder *BBLAS/test*. The main testing driver is the binary ./test. It should be called with the kernel to test as the first parameter followed by the kernel arguments. For example *./test dgemm_batch* will run the double precision of version of *dgemm_batch* with default arguments. For help ./test -h will display the list of kernel available for testing, while *./test dgemm_batch -h* will display more help and details on how to test *dgemm_batch*, this holds for all the kernels. We use random matrices for the test. To simplify the way to provide matrices in a batch and the size of each group, we provide the following arguments:

- `--ng` : the number groups [default: `--ng=10`].

- `--gs` : the number matrices in the first group [default: `--gs=100`].
- `--incg` : the increment of group sizes. The size of the i^th group is $gs + i * incg$ [default: `--incg=10`].
- `--dim` : M x N x K dimensions of the matrices in the first group [default: `--dim=50 x 50 x 50`].
- `--incm` : The increment of matrix size across the group. If the matrix size in the first group is M x N x K, the matrix size in the i-th group will be $(M + \texttt{incm} \times i) \times (N + \texttt{incm} \times i) \times (K + \texttt{incm} \times i)$ [default: `--incm = 1`].
- `--info`[a|g|n|o] : The parameter to set an error handling option [default: `--info=a`].
  - `a`: which indicates that all errors will be specified on output.
  - `g`: which indicates that only a single error will be reported for each group, independently.
  - `n`: which indicates that no errors will be reported on output.
  - `o`: which indicates that the occurrence of errors will be specified on output as a single integer value.

In addition to these options, arguments like `trans`, `transa`, `transb`, `side`, `diag`, `uplo`, etc., can be set, and for the sake of simplicity, they have the same value in all the groups.

- `--transa=`[n|t|c] : transposition [default: `--transa=n`].
- `--transa=`[n|t|c] : transposition of A [default: `--transa=n`].
- `--transb=`[n|t|c] : transposition of B [default: `--transb=n`].
- `--side=`[l|r] : left or right side application [default: `--side=l`].
- `--uplo=`[g|u|l] : general rectangular or upper or lower triangular matrix [default: `--uplo=l`].
- `--diag=`[n|u] : non-unit diagonal or unit diagonal [default: `--diag=n`].

One important aspect of a reference implementation is the accuracy testing. As the Batched BLAS is a high level API on top of BLAS operations, its accuracy depends on the correctness of the results from the BLAS library provided by the user. Our accuracy testing consists then in comparing the results from the call the Batched BLAS routines to results from direct calls to the corresponding BLAS routines. A test is successful if the forward error associated with each single problem in the batch is less than predetermined error.

5 Conclusion

The goal of this deliverable is to provide vendors like NVIDIA, Intel, AMD, etc., and the developers of applications with a reference implementation of Batched BLAS operations. This implementation satisfies all the requirements of the Batched BLAS standard specification, and is delivered with a detailed documentation and testing routines. As some Batched BLAS routines are already developed by Intel and NVIDIA, we hope that in the coming releases, they will follow this reference, and that all the vendors will join and will provide an optimized version for their machine, to enable application portability across different architectures.

With the advent of 16 bits floating point arithmetic computing capability in hardware and the need of 16 bits computation in deep learning applications, we are currently extending both the standard specification and the reference implementation to 16 bits floating point precision.

Acknowledgments

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

References

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

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

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

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

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