Multiple-Precision BLAS Library for Graphics Processing Units

Konstantin Isupov [1] and Vladimir Knyazkov [2]

[1]Vyatka State University
[2]Affiliation not available

October 30, 2023

Abstract

The binary32 and binary64 floating-point formats provide good performance on current hardware, but also introduce a rounding error in almost every arithmetic operation. Consequently, the accumulation of rounding errors in large computations can cause accuracy issues. One way to prevent these issues is to use multiple-precision floating-point arithmetic. This preprint, submitted to Russian Supercomputing Days 2020, presents a new library of basic linear algebra operations with multiple precision for graphics processing units. The library is written in CUDA C/C++ and uses the residue number system to represent multiple-precision significands of floating-point numbers. The supported data types, memory layout, and main features of the library are considered. Experimental results are presented showing the performance of the library.

Preprint submitted to Russian Supercomputing Days 2020

Multiple-Precision BLAS Library for Graphics Processing Units

Konstantin Isupov[1] and Vladimir Knyazkov[2]

[1] Vyatka State University, Kirov, 610000, Russia,
ks_isupov@vyatsu.ru,
[2] Penza State University, Penza, 440026, Russia,
kniazkov@pnzgu.ru

Abstract. The binary32 and binary64 floating-point formats provide good performance on current hardware, but also introduce a rounding error in almost every arithmetic operation. Consequently, the accumulation of rounding errors in large computations can cause accuracy issues. One way to prevent these issues is to use multiple-precision floating-point arithmetic. This paper presents a new library of basic linear algebra operations with multiple precision for graphics processing units. The library is written in CUDA C/C++ and uses the residue number system to represent multiple-precision significands of floating-point numbers. The supported data types, memory layout, and main features of the library are considered. Experimental results are presented showing the performance of the library.

Keywords: Multiple-precision computation · Floating-point arithmetic · BLAS · CUDA · Parallel algorithm

1 Introduction

It is no surprise that floating-point operations have rounding errors that occur during calculations. Such errors are natural due to the limited length of the significand in the binary32 and binary64 formats from the IEEE 754 standard [1]. For many applications, these errors do not prevent obtaining the correct results. Moreover, for some applications such as deep learning, the best option is to use lower precision formats, e.g., the 16-bit (half-precision) format [2]. However, for a rapidly growing number of scientific computing applications the natively supported IEEE 754 formats are not enough and a higher level of precision is required [3–6]. For such applications, multiple-precision libraries are used, which allow one to perform arithmetic operations on numbers represented with hundreds and thousands of digits.

This paper describes a library that provides new parallel algorithms and implementations for a number of basic linear algebra operations, like the BLAS routines [7], with multiple precision. The library, called MPRES-BLAS, is designed to be used on high-performance computing systems equipped with modern graphics processing units (GPUs). The library uses the residue number system

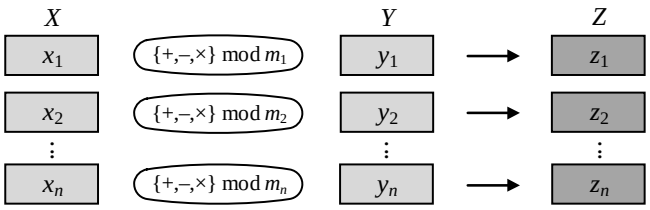

Fig. 1. Performing arithmetic operations in RNS

(RNS) [8,9] to represent multiple-precision numbers. In the RNS, a set of moduli are given which are independent of each other. A number is represented by the residue of each modulus and the arithmetic operations are based on the residues individually as shown in Fig. 1. This introduces parallelism in arithmetic with multiple precision and makes RNS a promising number system for many-core architectures such as GPUs.

2 Related Work

One approach to get greater precision and accuracy is to use floating-point expansions, when an extended-precision number is represented as an unevaluated sum of several ordinary floating-point numbers. An example of such an expansion is the double-double format, capable of representing at least 106 bits of significand. Each double-double number is represented as an unevaluated sum of two binary64 numbers. In turn, the quad-double format is capable of representing 212 bits of significand by using four binary64 numbers. Algorithms for computing floating-point expansions are called error-free transformations [10]. Double-double arithmetic is used in the XBLAS [11] and QPBLAS [12] packages for CPUs, as well as in the QPBLAS-GPU package for GPUs [13].

The ExBLAS package [14] contains a number of optimized implementations of accurate and reproducible linear algebra operations for parallel architectures such as Intel Xeon Phi and GPUs. Reproducibility is defined as the ability to obtain a bit-wise identical result from multiple runs of the code on the same input data. To ensure reproducibility, ExBLAS uses error-free transformations and long fixed-point accumulators that can represent every bit of information of the input floating-point format (binary64). The use of long accumulators provides the replacement of non-associative floating-point operations with fixed-point operations that are associative.

The paper [15] presents highly optimized GPU implementations of the DOT, GEMV, GEMM, and SpMV operations, which are included in the BLAS-DOT2 package. In these implementations, internal floating-point operations are performed with at least 2-fold the precision of the input and output data precision, namely, for binary32 data, the computation is performed using the binary64 format, whereas for binary64 data, the computation is performed using the Dot2 algorithm [16], which is based on error-free transformations.

Nakata et al. proposed MPACK (aka MPLAPACK) [17], a package of multiple-precision linear algebra routines. It consists of two modules, MBLAS and MLAPACK, which are multiple-precision versions of BLAS and LAPACK for CPUs, respectively. MPACK supports several libraries like GMP, MPFR, and QD for underlying multiple-precision arithmetic. In addition, MPACK provides double-double implementations of the GEMM and SYRK routines for GPUs.

There are also a number of extended- and multiple-precision arithmetic libraries for GPUs. Support for double-double and quad-double is implemented in GQD [18], which allows one to perform basic arithmetic operations and a number of mathematical functions with extended precision. GQD mainly uses the same algorithms as the QD library for CPUs. To represent extended precision numbers, GQD uses the vector types double2 and double4 available in CUDA.

CAMPARY [19] uses n-term floating-point expansions (generalization of the double-double and quad-double formats to an arbitrary number of terms) and provides flexible CPU and GPU implementations of multiple-precision arithmetic operations. Both the binary64 and the binary32 formats can be used as basic blocks for the floating-point expansion, and the precision (expansion size) is specified as a template parameter. Generally, each addition and multiplication of n-component expansions in CAMPARY requires $3n^2 + 10n - 4$ and $2n^3 + 2n^2 + 6n - 4$ standard floating-point operations.

GARPREC [18] and CUMP [20] support arbitrary precision on GPUs using the so-called "multi-digit" format. This format stores a multiple-precision number with a sequence of digits coupled with a single exponent. The digits are themselves machine integers. The GARPREC algorithms are from David Bailey's ARPREC package for CPUs, whereas CUMP is based on the GNU MP Bignum library (GMP). In both GARPREC and CUMP, each multiple-precision operation is implemented as a single thread and an interval memory layout is used in order to exploit the coalesced access feature of the GPU.

In [21], we have proposed new multiple-precision arithmetic algorithms using the residue number system and adapted them for efficient computations with multiple-precision vectors on GPUs. Similar algorithms for dense multiple-precision matrices were then proposed and implemented. All these algorithms are used in the MPRES-BLAS library, which is discussed in this paper.

Table 1 summarizes the software packages considered in this section.

Table 1. Software for accurate and/or higher precision computations

Library	Platform	Source code	Ref.
XBLAS	CPU	https://www.netlib.org/xblas	[11]
QPBLAS	CPU	https://ccse.jaea.go.jp/software/QPBLAS	[12]
QPBLAS-GPU	GPU	https://ccse.jaea.go.jp/software/QPBLAS-GPU	[13]
ExBLAS	CPU, Phi, GPU	https://github.com/riakymch/exblas	[14]
BLAS-DOT2	GPU	http://www.math.twcu.ac.jp/ogita/post-k/results.html	[15]
MPACK	CPU	https://github.com/nakatamaho/mplapack	[17]
GQD, GARPREC	GPU	https://code.google.com/archive/p/gpuprec	[18]
CAMPARY	CPU, GPU	http://homepages.laas.fr/mmjoldes/campary	[19]
CUMP	GPU	https://github.com/skystar0227/CUMP	[20]
MPRES-BLAS	GPU	https://github.com/kisupov/mpres-blas	[21]

3 Data Types and Conversions

In MPRES-BLAS, a floating-point number is an object consisting of a sign, a multiple-precision significand, an integer exponent, and some additional information about the significand. We denote such an object as follows:

$$x = \langle s, X, e, I(X/\mathcal{M}) \rangle, \tag{1}$$

where s is the sign, X is the significand, e is the exponent, and $I(X/\mathcal{M})$ is the interval evaluation of the significand (additional information).

The significand is represented in the RNS by the residues $(x_1, x_2, \ldots, x_n)$ relative to the moduli set $\{m_1, m_2, \ldots, m_n\}$ and is considered as an integer in the range of 0 to $\mathcal{M} - 1$, where $\mathcal{M} = \prod_{i=1}^{n} m_i$. The residues $x_i = X \bmod m_i$ are machine integers. The binary number X corresponding to the given residues $(x_1, x_2, \ldots, x_n)$ can be derived using the Chinese remainder theorem [8] as $X = \left| \sum_{i=1}^{n} \mathcal{M}_i \left| x_i w_i \right|_{m_i} \right|_{\mathcal{M}}$, where $\mathcal{M}_i$ and w_i are the RNS constants, namely $\mathcal{M}_i = \mathcal{M}/m_i$ and w_i is the modulo m_i multiplicative inverse of $\mathcal{M}_i$. Hence, one can compute the value of a floating-point number (1) using the following formula:

$$x = (-1)^s \times \left| \sum_{i=1}^{n} \mathcal{M}_i \left| x_i w_i \right|_{m_i} \right|_{\mathcal{M}} \times 2^e.$$

Unlike the addition, subtraction, and multiplication operations that are based on the residues individually, comparison, sign determination, overflow detection, scaling, and general division are time-consuming operations in the RNS, since they require estimating the magnitude of numbers. These operations are called non-modular operations. The classic technique to perform these operations is based on the Chinese remainder theorem and consists in computing binary representations of numbers with their subsequent analysis. In large dynamic ranges this technique becomes slow. MPRES-BLAS uses an alternative method for implementing non-modular operations, which is based on computing the interval evaluation of the fractional representation of an RNS number [22]. This method is designed to be fast on modern massively parallel general-purpose computing platforms such as GPU. The interval evaluation $I(X/\mathcal{M})$ is defined by two bounds, $\underline{X/\mathcal{M}}$ and $\overline{X/\mathcal{M}}$, that localize the value of X scaled by $\mathcal{M}$. The bounds are represented in a working precision floating-point format with an extended exponent in order to avoid underflow when $\mathcal{M}$ is large. To compute $\underline{X/\mathcal{M}}$ and $\overline{X/\mathcal{M}}$, only modulo m_i operations and standard floating-point operations are required. Using interval evaluations, efficient algorithms have been developed for implementing a number of non-modular operations in the RNS [22].

In [21], basic algorithms are proposed for performing arithmetic operations with numbers of the form (1) and the following rounding error bound is given: if $\mathrm{fl}(x \circ y)$ is the rounded result of an operation $\circ \in (+, -, \times)$, then

$$\mathrm{fl}(x \circ y) = (x \circ y)(1 + \delta), \qquad |\delta| < \mathbf{u},$$

where $\mathbf{u} < 4/\sqrt{\mathcal{M}}$ and $\mathcal{M}$ is the RNS moduli product. Hence, the user can set arbitrary arithmetic precision by changing the used set of RNS moduli.

The C data type corresponding to a multiple-precision number of the form (1) is `mp_float_t`, defined as the following structure:

```
typedef struct {
    int digits[n];
    int sign;
    int exp;
    er_float_t eval[2];
} mp_float_t;   /* Single multiple-precision value */
```

where `er_float_t` is the C data type representing a working precision floating-point value with an extended exponent.

The `mp_float_t` type is used mainly in the host code, and MPRES-BLAS provides a set of functions for converting data between `mp_float_t` and `double`, and also between `mp_float_t` and the `mpfr_t` type from the GNU MPFR library[3].

To store an array of multiple-precision numbers in the GPU memory, MPRES-BLAS uses the data type `mp_array_t`, defined as the following structure:

```
typedef struct {
    int * digits;
    int * sign;
    int * exp;
    er_float_t * eval;
    int4 * buf;
    int * len;
} mp_array_t;   /* Array of multiple-precision values */
```

The fields of this structure are as follows, where n is the precision (size of the RNS moduli set) and L is the length of the multiple-precision array:

- `digits` are the digits (residues) of significands (an integer array of size $n \times L$); all digits belonging to the same multiple-precision number are arranged consecutively in the memory;
- `sign` are the signs (an integer array of size L);
- `exp` are the exponents (an integer array of size L);
- `eval` are the interval evaluations of significands (an array of size $2L$, where the first L elements represent the lower bounds of the interval evaluations, and the second L elements represent the upper bounds);
- `buf` is the buffer (an array of size L used in arithmetic operations to transfer auxiliary variables between CUDA kernels);
- `len` is the number of items that the array holds, i.e. L; for a vector of length N, the array must contain at least $(1 + (N-1) \times |incx|)$ items, where $incx$ specifies the stride for the items; for a matrix of size $M \times N$, the array must contain at least $LDA \times N$ items, where LDA specifies the leading dimension of the matrix; the value of LDA must be at least $\max(1, M)$.

[3] https://www.mpfr.org

Using the `mp_array_t` structure instead of an array of `mp_float_t` structures allows ensuring coalesced memory access when dealing with multiple-precision vectors and matrices; details can be found in [21]. MPRES-BLAS provides the following helper functions for working with `mp_array_t` instances:

- `mp_array_init`: allocate a multiple-precision array in the GPU memory;
- `mp_array_clear`: release the GPU memory occupied by an array;
- `mp_array_host2device`: convert a regular multiple-precision array allocated on the host (with elements of type `mp_float_t`) to an `mp_array_t` instance allocated on the GPU;
- `mp_array_device2host`: convert an `mp_array_t` instance allocated on the GPU to a regular multiple-precision array allocated on the host.

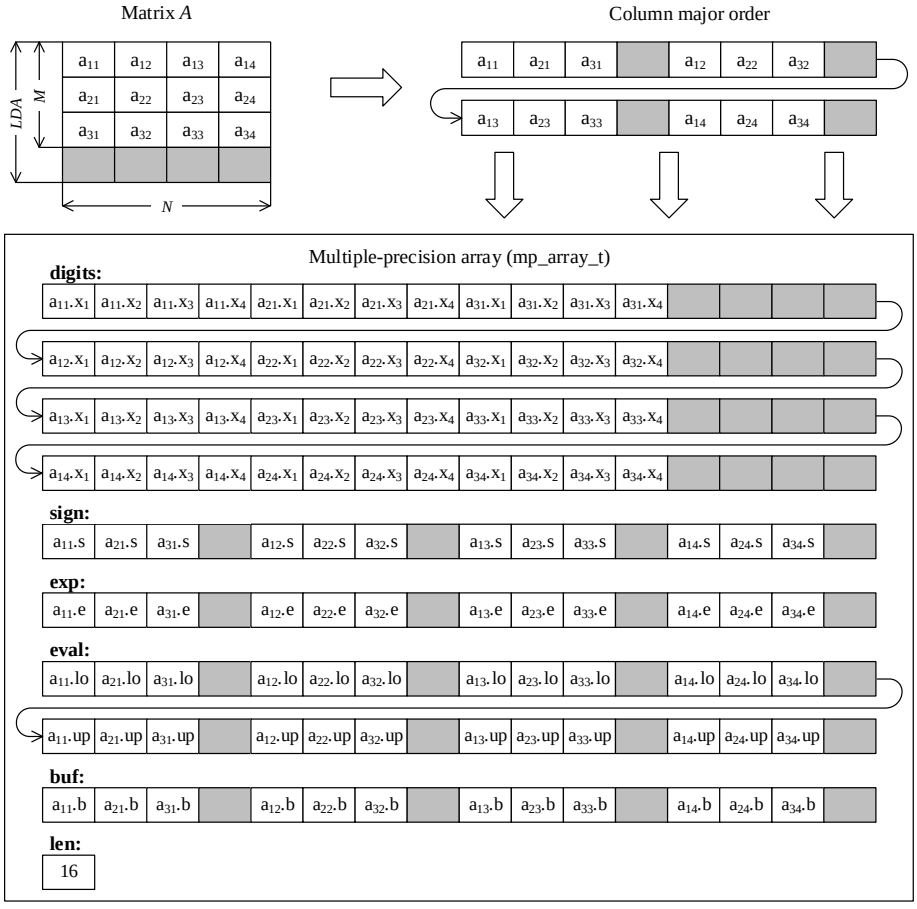

Fig. 2. Storage layout of a multiple-precision matrix in MPRES-BLAS

4 Data Layout

The BLAS routines in MPRES-BLAS follow the Fortran convention of storing matrices using the column major data format. Thus, for an M by N matrix A, the address of the kth digit of the element a_{ij} in 0-based indexing is given by

$$address = k + (i + j \times LDA) \times n,$$

where n is the size of the RNS moduli set, $0 \leq k < n$, $0 \leq i < M$, and $0 \leq j < N$.

Figure 2 illustrates the column major layout of a 3×4 multiple-precision matrix using the `mp_array_t` data type. In this example, $n = 4$, i.e. the significand of each element a_{ij} consists of four digits: $X = (x_1, x_2, x_3, x_4)$. We use the dot symbol (.) to access the parts of a multiple-precision number. The symbols *lo* and *up* denote the lower and upper bounds of the interval evaluation so that $lo := \underline{X/\mathcal{M}}$ and $up := \overline{X/\mathcal{M}}$. Note that we use 1-based indexing in this example.

5 Functionality

The current version of MPRES-BLAS (ver. 1.0) provides 16 GPU-based multiple-precision BLAS functions listed in Table 2. All the functions support the standard BLAS interface, except that some functions have additional arguments that are pointers to auxiliary buffers in the global GPU memory.

Table 2. Multiple-precision linear algebra operations supported by MPRES-BLAS.

ASUM — Sum of absolute values	GEMV — Matrix-vector multiplication
DOT — Dot product of two vectors	GEMM — General matrix multiplication
SCAL — Vector-scalar product	GER — Rank-1 update of a general matrix
AXPY — Constant times a vector plus a vector	GE_ADD — Matrix add and scale
AXPY_DOT — Combined AXPY and DOT	GE_ACC — Matrix accumulation and scale
WAXPBY — Scaled vector addition	GE_DIAG_SCALE — Diagonal scaling
NORM — Vector norms	GE_LRSCALE — Two-sided diagonal scaling
ROT — Apply a plane rotation to vectors	GE_NORM — Matrix norms

Implementation details, performance data, and accuracy of the ASUM, DOT, SCAL, and AXPY functions are given in [21]. Table 3 show the forward error bounds for the GEMV, GEMM, GE_ADD, and GER routines. In this table it is assumed that each matrix is a square N by N matrix.

We note that not all of the functions listed in Table 2 can be ill-conditioned and require higher numeric precision. For example, as shown in [21], the SCAL and ASUM functions have relative forward error bounds of $\mathbf{u}$ and $\mathbf{u}(N-1)/(1-\mathbf{u}(N-1))$, respectively. However, support for such functions allow one to eliminate time-consuming conversions between the IEEE 754 formats and the multiple-precision format (1) in the intermediate steps of a computation.

To achieve high performance, GPU kernels must be written according to some basic principles/techniques, stemming from the specifics of the GPU architecture [24]. One such technique is blocking. The idea is to operate on blocks

Table 3. Error bounds for the functions from MPRES-BLAS. According to [23], the quantity γ_N is defined as $\gamma_N := N\mathbf{u}/(1 - N\mathbf{u})$.

	Absolute error bound	Relative error bound
GEMV	$\gamma_{N+2} \cdot \big\|\|\alpha A\| \cdot \|x\| + \|\beta y\|\big\|$	$\gamma_{N+2} \cdot \big\|\|\alpha A\| \cdot \|x\| + \|\beta y\|\big\| \;/\; \big\|\alpha Ax + \beta y\big\|$
GEMM	$\gamma_{N+2} \cdot \big\|\|\alpha A\| \cdot \|B\| + \|\beta C\|\big\|$	$\gamma_{N+2} \cdot \big\|\|\alpha A\| \cdot \|B\| + \|\beta C\|\big\| \;/\; \big\|\alpha AB + \beta C\big\|$
GE_ADD	$\gamma_2 \cdot \big\|\|\alpha A\| + \|\beta B\|\big\|$	$\gamma_2 \cdot \big\|\|\alpha A\| + \|\beta B\|\big\| \;/\; \big\|\alpha A + \beta B\big\|$
GER	$\gamma_3 \cdot \big\|\|\alpha xy^T\| + \|A\|\big\|$	$\gamma_3 \cdot \big\|\|\alpha xy^T\| + \|A\|\big\| \;/\; \big\|\alpha xy^T + A\big\|$

of the original vector or matrix. Blocks are loaded once into shared memory and then reused. The goal is to reduce off-chip memory accesses. However, when working with multiple-precision numbers, shared memory is the limiting factor for occupancy, since the size of each number may be too large. Another problem is that the multiple-precision arithmetic algorithms contain serial and parallel sections, and while one thread computes the exponent, sign, and interval evaluation, all other threads are idle. This results in divergent execution paths within a warp and can lead to significant performance degradation. In order to resolve this problem, MPRES-BLAS follows the approach proposed in [21], according to which multiple-precision operations are split into several parts, each of which is performed by a separate CUDA kernel (*_global_* function). These parts are

- computing the signs, exponents, and interval evaluations;
- computing the significands in the RNS;
- rounding the results.

Thus, each BLAS function consists of a sequence of kernel launches. In some cases, such a decomposition may increase the total number of global memory accesses, since all intermediate results should be stored in the global memory. However, this eliminates branch divergence when performing sequential parts of multiple-precision operations. Furthermore, each kernel has its own execution configuration which makes it possible to use all available GPU resources.

As an example, Fig. 3 shows a flowchart of the multiple-precision GE_ADD operation ($C \leftarrow \alpha A + \beta B$). Other BLAS operations have a similar structure. An exception is reduction operations, in which each operation with multiple precision is performed as a single thread, and communication between threads is performed using shared memory.

In MPRES-BLAS, multiple-precision vectors are processed on the GPU with 1-dimensional grids of 1-dimensional thread blocks, whereas matrices are processed with 2-dimensional grids of 1-dimensional or 2-dimensional thread blocks. Each BLAS operation has template parameters that specify the execution configuration of the kernels. For computations with multiple-precision significands, the number of threads per block is precision dependent and is configured automatically so as to ensure maximum occupancy of a streaming multiprocessor.

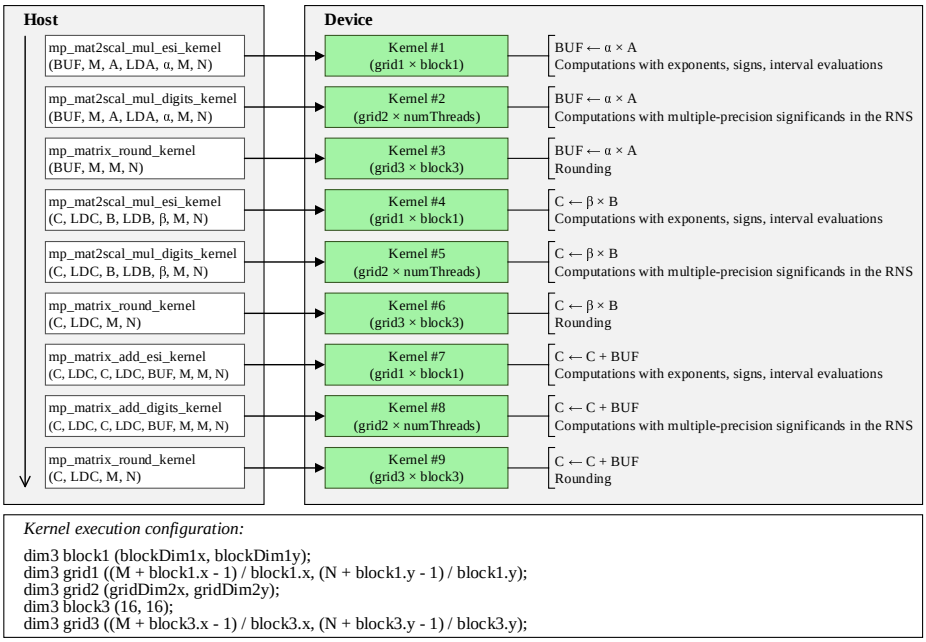

Fig. 3. Flowchart of the multiple-precision GE_ADD operation. Here *blockDim1x*, *blockDim1y*, *gridDim2x*, and *gridDim2y* are tunable parameters that are passed to the function as template arguments, whereas *numThreads* is automatically configured depending on the precision and hardware specification.

6 Performance Evaluation

In this section, we evaluate the performance of the GEMV, GE_ADD and GEMM functions from MPRES-BLAS for various problem sizes and levels of numeric precision. In the experiments, we used the NVIDIA GeForce GTX 1080 GPU. The host machine was equipped with an Intel Core i5-7500 and 16 GB of RAM and run Ubuntu 18.04.4 LTS. The GPU codes were compiled by nvcc 10.2.89 with options -O3 -use_fast_math -Xcompiler=-O3,-fopenmp,-ffast-math. For comparison, we also evaluated the GARPREC (only for GEMV), CUMP, and CAMPARY libraries; see Section 2 for details. The input vectors, matrices, and scalars were initialized with random numbers over [-1, 1]. Our measurements do not include the time spent transferring data between the CPU and the GPU, as well as time of converting data into internal multiple-precision representations.

Figure 4 shows the experimental results. In many cases, MPRES-BLAS has better performance than implementations based on existing multiple-precision libraries for GPUs. Increasing the precision of computations increases the speedup of MPRES-BLAS. At 106-bit precision, the performance of MPRES-BLAS is on average 6 times lower than that of CAMPARY. This is because the RNS-based

algorithms implemented in MPRES-BLAS are designed for arbitrary (mostly large) precision and have overhead associated with calculating interval evaluations and storing intermediate results in the global memory. On the other hand, MPRES-BLAS is less dependent on precision than CAMPARY and with 848-bit precision, MPRES-BLAS performs on average 9 times better than CAMPARY.

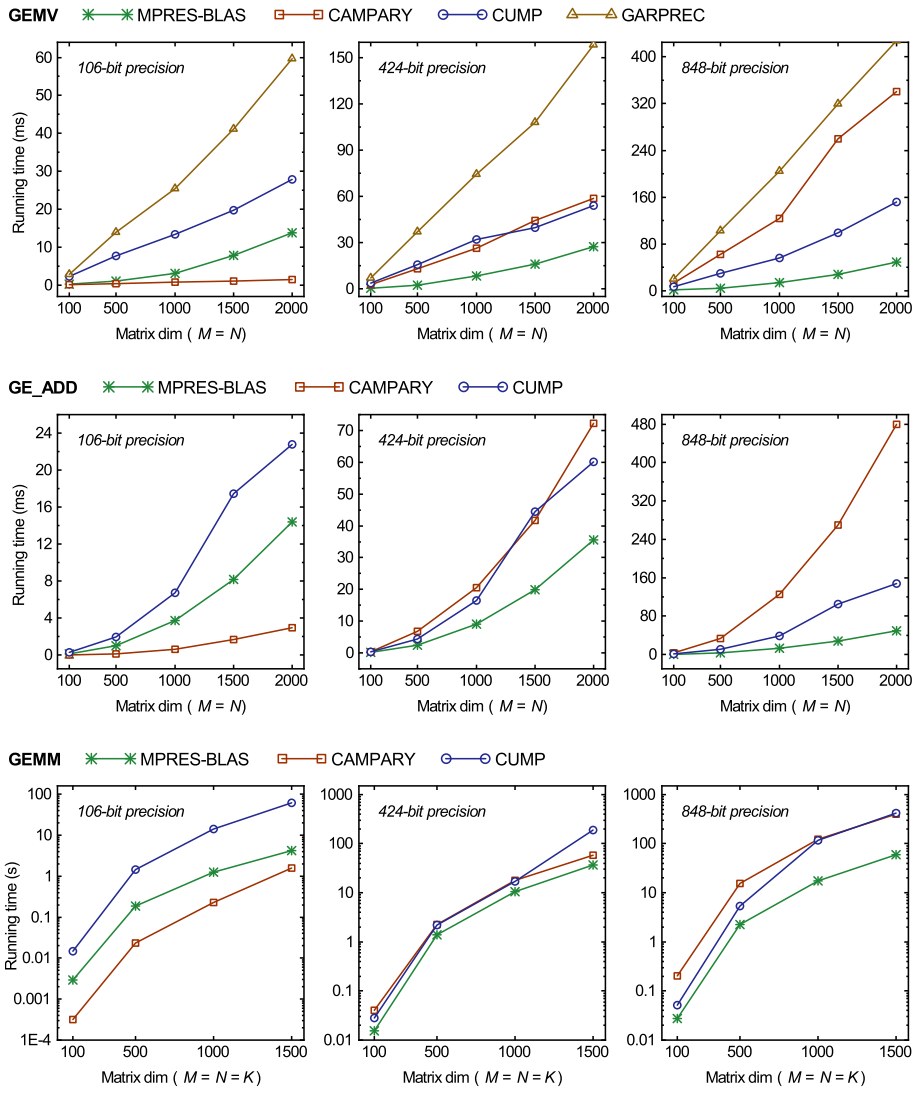

Fig. 4. Performance of multiple-precision GEMV, GE_ADD and GEMM on GeForce GTX 1080 as a function of problem size.

7 Conclusion

In this paper, we presented MPRES-BLAS, a new library of basic linear algebra operations with multiple precision for CUDA compatible GPUs based on the residue number system. The current version of MPRES-BLAS (ver. 1.0) supports 16 multiple-precision operations from levels 1, 2, and 3 of the BLAS. Some operations such as SCAL, NORM and ASUM typically do not require precision higher than the one provided by the natively supported floating-point formats. Nevertheless, support for such functions allow one to eliminate time-consuming intermediate conversions between the natively supported formats and the RNS-based multiple-precision format used.

In addition to the linear algebra kernels, MPRES-BLAS implements basic arithmetic operations with multiple precision for CPU and GPU (through the `mp_float_t` data type), so it can also be considered as a general purpose multiple-precision arithmetic library. Furthermore, the library provides a number of optimized RNS algorithms, such as magnitude comparison and power-of-two scaling, and also supports extended-range floating-point arithmetic with working precision, which prevents underflow and overflow in a computation involving extremely large or small quantities. The functionality of MPRES-BLAS will be supplemented and improved over time.

Acknowledgement. This work was supported by the Russian Science Foundation (grant number 18-71-00063).

References

1. IEEE Standard for Floating-Point Arithmetic. Standard, Institute of Electrical and Electronics Engineers, New York, NY, USA (2019). DOI 10.1109/IEEESTD.2019.8766229
2. Courbariaux, M., Bengio, Y., David, J.P.: Training deep neural networks with low precision multiplications (2014). URL https://arxiv.org/abs/1412.7024. Accessed 10 April 2020
3. Bailey, D.H., Borwein, J.M.: High-precision arithmetic in mathematical physics. Mathematics **3**(2), 337–367 (2015). DOI 10.3390/math3020337
4. Daněk, J., Pospíšil, J.: Numerical aspects of integration in semi-closed option pricing formulas for stochastic volatility jump diffusion models. International Journal of Computer Mathematics pp. 1–25 (2019). DOI 10.1080/00207160.2019.1614174
5. Feng, Y., Chen, J., Wu, W.: The PSLQ algorithm for empirical data. Math. Comp. **88**(317), 1479–1501 (2019). DOI 10.1090/mcom/3356
6. Kyung, M.H., Sacks, E., Milenkovic, V.: Robust polyhedral Minkowski sums with GPU implementation. Comput. Aided Des. **67-68**, 48–57 (2015). DOI 10.1016/j.cad.2015.04.012
7. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for Fortran usage. ACM Trans. Math. Softw. **5**(3), 308—-323 (1979). DOI 10.1145/355841.355847
8. Omondi, A., Premkumar, B.: Residue Number Systems: Theory and Implementation. Imperial College Press, London, UK (2007)

9. Lu, M.: Arithmetic and Logic in Computer Systems. John Wiley and Sons, Inc., Hoboken, NJ, USA (2004)
10. Shewchuk, J.R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete & Computational Geometry **18**(3), 305–363 (1997). DOI 10.1007/PL00009321
11. Li, X.S., Demmel, J.W., Bailey, D.H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S.Y., Kapur, A., Martin, M.C., Thompson, B.J., Tung, T., Yoo, D.J.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. **28**(2), 152–205 (2002). DOI 10.1145/567806.567808
12. Yamada, S., Ina, T., Sasa, N., Idomura, Y., Machida, M., Imamura, T.: Quadruple-precision blas using bailey's arithmetic with fma instruction: its performance and applications. In: Proc. 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1418–1425 (2017). DOI 10.1109/IPDPSW.2017.42
13. Quadruple precision BLAS routines for GPU: QPBLAS-GPU (2013). URL https://ccse.jaea.go.jp/software/QPBLAS-GPU/1.0/manual/qpblas-gpu_manual_en-1.0.pdf. Accessed 16 May 2020
14. Iakymchuk, R., Collange, S., Defour, D., Graillat, S.: ExBLAS: Reproducible and accurate BLAS library. In: Proc. Numerical Reproducibility at Exascale (NRE2015) Workshop at SC15 (2015)
15. Mukunoki, D., Ogita, T.: Performance and energy consumption of accurate and mixed-precision linear algebra kernels on GPUs. Journal of Computational and Applied Mathematics **372**, 112,701 (2020). DOI 10.1016/j.cam.2019.112701
16. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. **26**(6), 1955—1988 (2005). DOI 10.1137/030601818
17. Nakata, M.: Poster: MPACK 0.7.0: Multiple precision version of BLAS and LAPACK. In: Proc. 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 1353–1353 (2012). DOI 10.1109/SC.Companion.2012.183
18. Lu, M., He, B., Luo, Q.: Supporting extended precision on graphics processors. In: Sixth International Workshop on Data Management on New Hardware (DaMoN'10), pp. 19–26 (2010). DOI 10.1145/1869389.1869392
19. Joldes, M., Muller, J., Popescu, V.: Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming. In: Proc. 2017 IEEE 24th Symposium on Computer Arithmetic (ARITH), pp. 27–34 (2017). DOI 10.1109/ARITH.2017.18
20. Nakayama, T., Takahashi, D.: Implementation of multiple-precision floating-point arithmetic library for GPU computing. In: Proc. 23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011), pp. 343–349 (2011). DOI 10.2316/P.2011.757-041
21. Isupov, K., Knyazkov, V., Kuvaev, A.: Design and implementation of multiple-precision BLAS Level 1 functions for graphics processing units. Journal of Parallel and Distributed Computing **140**, 25–36 (2020). DOI 10.1016/j.jpdc.2020.02.006
22. Isupov, K.: Using floating-point intervals for non-modular computations in residue number system. IEEE Acces **8**, 58,603–58,619 (2020). DOI 10.1109/ACCESS.2020.2982365
23. Higham, N.J.: Accuracy and stability of numerical algorithms, 2nd edn. SIAM, Philadelphia, PA, USA (2002). DOI 10.1137/1.9780898718027
24. Nath, R., Tomov, S., Dong, T.T., Dongarra, J.: Optimizing symmetric dense matrix-vector multiplication on GPUs. In: Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–10 (2011). DOI 10.1145/2063384.2063392. Article no. 6