ISSN 2079-3316 PROGRAM SYSTEMS: THEORY AND APPLICATIONS vol. 11, No 3(46), pp. 61–84

CSCSTI 50.09.31, 50.33.04
UDC 004.222+004.272.25

Konstantin Isupov, Vladimir Knyazkov

# Multiple-precision matrix-vector multiplication on graphics processing units

Abstract. We are considering a parallel implementation of matrix-vector multiplication (GEMV, Level 2 of the BLAS) for graphics processing units (GPUs) using multiple-precision arithmetic based on the residue number system. In our GEMV implementation, element-wise operations with multiple-precision vectors and matrices consist of several parts, each of which is calculated by a separate CUDA kernel. This feature eliminates branch divergence when performing sequential parts of multiple-precision operations and allows the full utilization of the GPU's resources. An efficient data structure for storing arrays with multiple-precision entries provides a coalesced access pattern to the GPU global memory. We have performed a rounding error analysis and derived error bounds for the proposed GEMV implementation. Experimental results show the high efficiency of the proposed solution compared to existing high-precision packages deployed on GPU.

*Key words and phrases:* multiple-precision computations, BLAS, GEMV, parallel algorithms, CUDA, GPU, residue number system.

2020 *Mathematics Subject Classification*: 68W10; 65F30, 68Q85

## Introduction

Floating-point operations have rounding errors that occur directly during calculations. Such errors are natural due to the limited length of the significand in the single-precision (binary32) and double-precision (binary64) IEEE 754 formats. For many applications, these errors do not prevent obtaining the correct calculation result. Moreover, for some applications such as deep learning, the best option is to use lower precision formats, e.g., the half-precision format [1]. At the same time, there

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

are currently many applications sensitive to rounding errors, and the accuracy of the standard formats is not enough for these applications [2–7]. Such applications use multiple-precision arithmetic libraries that provide operations with numbers whose digits of precision exceed the standard formats and in the general case are limited only by the available system memory.

Well-known multiple-precision libraries include GMP, MPFR, MPIR, and MPDECIMAL. These and many other libraries assume systems with central processing units (CPUs). However, applications requiring multiple-precision computations are often extremely resource-intensive and can be accelerated using massively parallel processing units such as graphics processing units (GPUs).

A modern graphics processing unit is actually an array of streaming multiprocessors (SMs), each of which contains many streaming processors. Massive parallelism of the GPU is achieved by replication of a common SM architecture, and each streaming processor can simultaneously execute the same instruction on different data. In 2007, NVIDIA released the Compute Unified Device Architecture (CUDA). Programmers have achieved parallel algorithms through C-like language without the need for any complex graphics programming [8]. Recent versions of GPUs, such as NVIDIA Tesla V100, offer up to 100 CPUs in a single GPU, making them an efficient tool for resource-intensive computing.

In this article, we consider a parallel implementation of matrix-vector multiplication (GEMV) with multiple precision for CUDA-compatible GPUs. GEMV is part of Level 2 of the Basic Linear Algebra Subprogram (BLAS) library [9] and performs one of the following operations:

$$y \leftarrow \alpha A x + \beta y \quad \text{or} \quad y \leftarrow \alpha A^T x + \beta y. \tag{1}$$

where $A$ is a dense $M$ by $N$ matrix, $x$ and $y$ are vectors, and $\alpha$ and $\beta$ are scalars. Along with other BLAS functions, GEMV is an important building block for many linear algebra algorithms, such as linear system and eigenvalue solvers.

One common strategy for optimizing GPU-based linear algebra calculations is blocking, according to which the matrix (or vector) divides into small parts that are loaded once into the GPU's shared memory and then reused in arithmetic operations [10]. The goal is to reduce global memory accesses. However, when working with multiple-precision numbers, intensive use of shared memory (as well as registers) is often not efficient. The reason is that if the size of each multiple-precision number is too large,

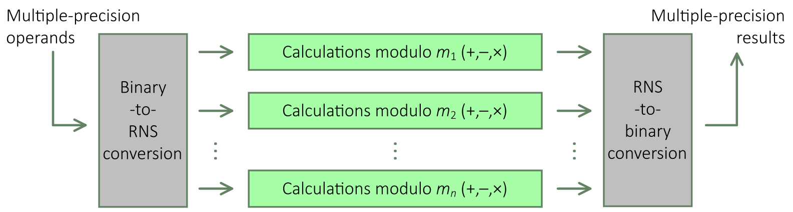
Multiple-precision operands
Binary -to- RNS conversion
Calculations modulo $m_1$ (+,–,×)
Calculations modulo $m_2$ (+,–,×)
Calculations modulo $m_n$ (+,–,×)
RNS -to- binary conversion
Multiple-precision results

Figure 1. Parallel arithmetic properties of the RNS.

then the scalability and performance of CUDA kernels drops significantly. Our implementation follows the approach proposed in [11], according to which each multiple-precision arithmetic operation divide into several parts. A separate CUDA kernel performs each piece with its configuration; all digits of multiple-precision numbers are calculated in parallel. This approach leads to an increase in the number of global memory accesses. It provides high performance and good scalability of computations with high precision on GPUs compared to the traditional paradigm where each multiple-precision arithmetic operation is performed as a single thread. To implement this approach, we use the residue number system (RNS) [12].

In RNS, a number is represented by its residues modulo a set of moduli called the base. The base bit width denotes the sum of the bit sizes of all moduli [13]. The residues are mutually independent. For addition, subtraction, and multiplication, the calculations with each residue are performed in the ring of integers modulo the corresponding modulus (see Figure 1), so that carry chains between the residues are eliminated. This approach allows one to compute all residues in a parallel manner.

The rest of the paper is structured as follows. Section 1 gives a brief overview of related work. Section 2 describes a format for representing multiple-precision numbers. Section 3 provides a data structure and layout for storing multiple-precision arrays in the GPU memory. Section 4 discusses algorithms and details of the proposed GEMV implementation, and Section 5 evaluates its accuracy. Experimental results are presented in Section 6. Conclusions and further research are presented in the last section of the paper.

## 1. High-Precision Computations and BLAS for GPU

Multiplication of a dense matrix by a vector (BLAS routines GEMV, SYMV, and TRMV) often occurs in scientific computing applications. The literature discusses the implementation of these operations for GPUs in IEEE 754 floating-point arithmetic [10,14–16]. On the other hand, there are studies on the implementation of high-precision BLAS operations for GPUs. There are also studies that consider accurate BLAS implementations for GPUs [11,17–19].

One approach to get higher 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 extension is the well-known double-double format. Here each extended-precision number comes as an unevaluated sum of two double-precision (binary64) numbers. This presentation corresponds to quadruple precision (at least 106 bits of significand). In turn, the quad-double format is capable of representing 212 bits of significand (octuple precision) by using four binary64 numbers to represent each extended precision number [20]. Algorithms for computing floating-point expansions are called error-free transformations [21,22]. Double-double arithmetic is used in several packages, including the QPBLAS-GPU package [17].

In [18], Iakymchuk et al. introduced ExBLAS, a package of optimized linear algebra operations that provides reproducible results. 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. ExBLAS contains a number of accurate and reproducible linear algebra operations for CPUs, Intel Xeon Phi coprocessors, as well as NVIDIA and AMD GPUs.

A recent study [19] presents optimized CUDA 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. In contrast, for binary64 data, the calculation is performed using the Dot2 algorithm [23], which is based on error-free transformations.

There are also several arithmetic libraries supporting extended or multiple precision on GPUs. In particular, the double-double and quad-double formats are supported in GQD [24], which allows one to perform basic arithmetic operations and several mathematical functions with extended precision, such as square root, logarithm, exponent, and trigonometric functions. GQD mainly uses the same algorithms as the QD library for CPUs[1]. To represent extended precision numbers, GQD uses the vector types double2 and double4 available in CUDA.

CAMPARY [25] uses $n$-term floating-point expansions 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. In CAMPARY, each addition and multiplication of $n$-term expansions generally requires $3n^2 + 10n - 4$ and $2n^3 + 2n^2 + 6n - 4$ standard floating-point operations, respectively, and optimized algorithms are used for double-double arithmetic, i.e., in the case of two-term expansions.

GARPREC [24] and CUMP [26] 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 similar to those implemented in the ARPREC package[1] for CPUs, whereas CUMP is based on GMP (GNU MP Bignum Library)[2]. Most functions from CUMP have a GMP-like regular interface. In both GARPREC and CUMP, each multiple-precision operation is implemented as a single thread; an interval memory layout is used to exploit the GPU’s coalesced access feature.

In [11], the present authors have proposed new algorithms for multiple-precision arithmetic based on RNS, and also algorithms for parallel computations with multiple-precision vectors using GPU. Subsequently, the later ones were adapted for dense matrices. All these algorithms are used in MPRES-BLAS, a new library of basic linear algebra operations with multiple precision for GPUs. Our experiments have shown that, in many cases, MPRES-BLAS has better performance than implementations based on existing high-precision packages for CPU and GPU. The GEMV implementation presented in this paper is part of MPRES-BLAS.

Table 1 summarizes the software packages considered and provides links to the source code.

[1]`https://www.davidhbailey.com/dhbsoftware`

[2]`https://gmplib.org`

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

| Package | Reference | Source code |
|---|---|---|
| QPBLAS-GPU | [17] | `https://ccse.jaea.go.jp/software/QPBLAS-GPU` |
| ExBLAS | [18] | `https://github.com/riakymch/exblas` |
| BLAS-DOT2 | [19] | `http://www.math.twcu.ac.jp/ogita/post-k/results.html` |
| GQD | [24] | `https://code.google.com/archive/p/gpuprec` |
| CAMPARY | [25] | `http://homepages.laas.fr/mmjoldes/campary` |
| GARPREC | [24] | `https://code.google.com/archive/p/gpuprec` |
| CUMP | [26] | `https://github.com/skystar0227/CUMP` |
| MPRES-BLAS | [11] | `https://github.com/kisupov/mpres-blas` |

## 2. Representation of arbitrary length floating-point numbers using RNS

A residue number system is defined by a set of $n$ coprime integers (moduli) $\{m_1, m_2, \dots, m_n\}$. The dynamic range of the RNS is specified by the moduli product [3]

$$\mathcal{M} = m_1 \cdot m_2 \cdot \dots \cdot m_n. \tag{2}$$

An integer $X \in [0, \mathcal{M} - 1]$ is uniquely represented in RNS by the residues $(x_1, x_2, \dots, x_n)$, where $x_i = |X|_{m_i}$ denotes the operation $X \bmod m_i$. The conversion from the RNS representation to binary is usually performed using the Chinese Remainder Theorem (CRT) [12]:

$$X = \left| \sum_{i=1}^{n} \mathcal{M}_i \left| x_i w_i \right|_{m_i} \right|_{\mathcal{M}}, \tag{3}$$

where $\mathcal{M}_i$ and $w_i$ are the constants such that $\mathcal{M}_i = \mathcal{M}/m_i$ and $w_i$ is the modulo $m_i$ multiplicative inverse[4] of $\mathcal{M}_i$.

Unlike the addition, subtraction and multiplication operations that are performed in RNS in parallel on all residues, the operations of comparison (unless both numbers that need to be compared are equal), overflow detection, sign detection, scaling, and division are time-consuming in RNS.

[3]The symbol $M$ is usually used to denote the product of RNS moduli. We refer to this product as $\mathcal{M}$ in this paper to avoid confusion with the number of matrix rows.

[4]If $x$ is a non-zero integer, then $y$ is said to be the modulo $m$ multiplicative inverse of $x$ if $|x \times y|_m = 1$.

These operations are often referred to as "non-modular" operations. When the dynamic range of an RNS consists of hundreds or thousands of bits, the classical CRT-based method of performing non-modular operations becomes inefficient. For this reason, we use an alternative method for implementing non-modular operations, which is based on computing the interval evaluation of the fractional representation of an RNS number [27]. For $X$ given by the residues $(x_1, x_2, \ldots, x_n)$, the fractional representation (relative value) is calculated as follows:

$$\frac{X}{\mathcal{M}} = \left| \sum_{i=1}^{n} \frac{|x_i w_i|_{m_i}}{m_i} \right|_1 . \tag{4}$$

The interval evaluation for $X/\mathcal{M}$ is denoted by $I(X/\mathcal{M}) = [\underline{X/\mathcal{M}}, \overline{X/\mathcal{M}}]$ and represents an interval such that $\underline{X/\mathcal{M}} \le X/\mathcal{M} \le \overline{X/\mathcal{M}}$. The bounds of this interval, $\underline{X/\mathcal{M}}$ and $\overline{X/\mathcal{M}}$, are working precision floating-point numbers. No matter what the size of the RNS moduli set and dynamic range, only standard arithmetic operations are required to compute $\underline{X/\mathcal{M}}$ and $\overline{X/\mathcal{M}}$. Using interval evaluations, efficient algorithms have been developed for implementing a number of non-modular operations in the RNS, such as number comparison and division [27].

In our multiple-precision format, a number is represented as follows:

$$x = (-1)^s \times \left| \sum_{i=1}^{n} \mathcal{M}_i \, |x_i w_i|_{m_i} \right|_{\mathcal{M}} \times 2^e, \tag{5}$$

where $s \in \{0, 1\}$ is the sign, $e$ is the integer exponent, and $x_1, x_2, \ldots, x_n$ are the digits (residues) of the significand $X$. The significand represented in the RNS and can take values from 0 to $\mathcal{M} - 1$. Each digit $x_i = X \bmod m_i$ is a signed integer.

As additional information, the interval evaluation of the significand ($I(X/\mathcal{M})$) is included in the number format. It allows one to implement efficient comparison, calculation of the sign, exponent alignment, and check of the need for rounding.

In previous papers of the authors, various methods of performing arithmetic operations with numbers of the form (5) were considered. In [11], improved algorithms are proposed for multiple-precision addition and multiplication, and the following relative error bound is derived. If $\mathrm{fl}(x \circ y)$ is the rounded result of an operation $\circ \in \{+, -, \times\}$ with numbers

Table 2. Specification of the `mp_array_t` structure for storing an array of multiple-precision numbers. Symbols $n$ and $L$ denote the size of the RNS moduli set and the length of the array, respectively.

| Structure member | Description |
|---|---|
| digits | An integer array of size $n \times L$ that stores the digits of multiple-precision significands. All the digits belonging to the same multiple-precision number are arranged consecutively in the memory. |
| sign | An integer array of size $L$ that stores the signs of numbers. |
| exp | An integer array of size $L$ that stores the exponents of numbers. |
| eval | An array of floating-point numbers with extended exponents that stores the interval evaluations of significands. The size of the array is $2L$, and the first $L$ elements represent the lower bounds of the interval evaluations, while the second $L$ elements represent the upper ones. |
| buf | An array (buffer) of length $L$ whose elements are numbers of the int4 type from the standard CUDA C/C++ headers. This buffer is used in addition and subtraction to transfer auxiliary variables between CUDA kernels. |
| len | The number of items that the multiple-precision array holds, i.e. $L$. For a vector of length $N$, the array must contain at least $(1 + (N - 1) \times \lvert incx \rvert)$ 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 $\max(1, M)$. |

of the form (5), then

$$\text{fl}(x \circ y) = (x \circ y)(1 + \delta), \qquad |\delta| < \mathbf{u}, \qquad \mathbf{u} < 4/\sqrt{\mathcal{M}}. \tag{6}$$

In Section 5, we use (6) to evaluate the accuracy of the implemented matrix-vector multiplication function.

## 3. Data layout

Our GEMV supports the column major format for storing matrices, which is standard for the BLAS and LAPACK libraries [14]. In order to ensure the coalesced global memory access, an array of multiple-precision numbers (vector or matrix) is stored as an instance of the `mp_array_t` structure, which is described in Table 2.

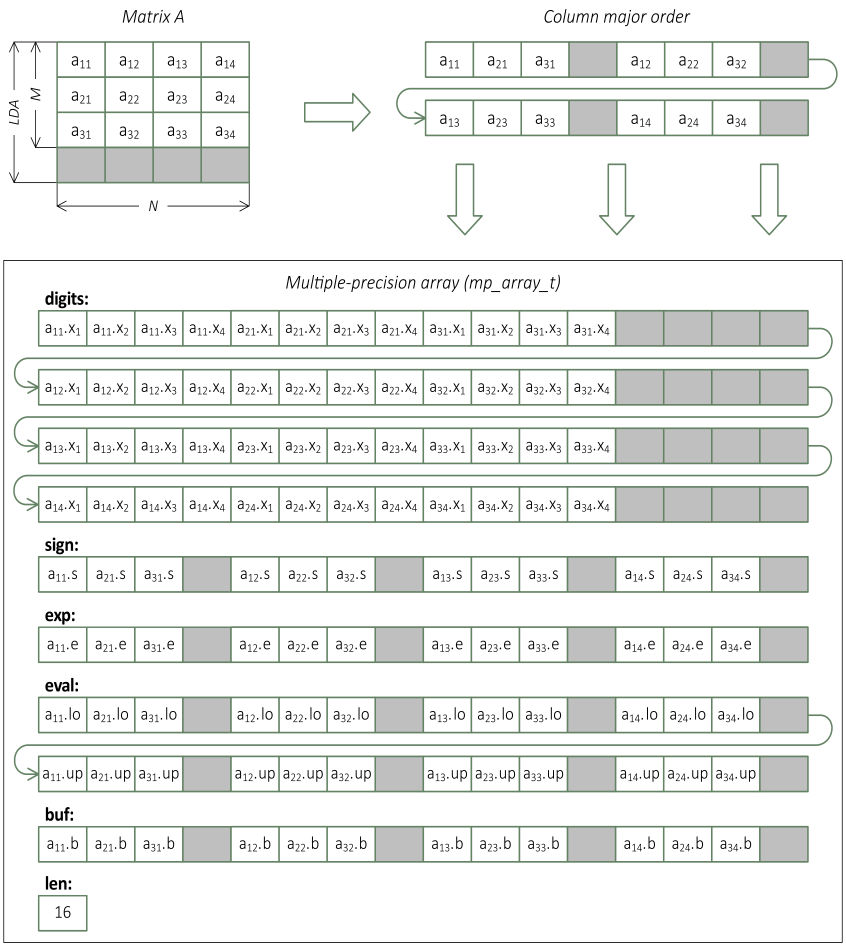
Matrix A
a11 a12 a13 a14
a21 a22 a23 a24
a31 a32 a33 a34
LDA
M
N
Column major order
a11 a21 a31 a12 a22 a32
a13 a23 a33 a14 a24 a34
Multiple-precision array (mp_array_t)
digits:
a11.x1 a11.x2 a11.x3 a11.x4 a21.x1 a21.x2 a21.x3 a21.x4 a31.x1 a31.x2 a31.x3 a31.x4
a12.x1 a12.x2 a12.x3 a12.x4 a22.x1 a22.x2 a22.x3 a22.x4 a32.x1 a32.x2 a32.x3 a32.x4
a13.x1 a13.x2 a13.x3 a13.x4 a23.x1 a23.x2 a23.x3 a23.x4 a33.x1 a33.x2 a33.x3 a33.x4
a14.x1 a14.x2 a14.x3 a14.x4 a24.x1 a24.x2 a24.x3 a24.x4 a34.x1 a34.x2 a34.x3 a34.x4
sign:
a11.s a21.s a31.s a12.s a22.s a32.s a13.s a23.s a33.s a14.s a24.s a34.s
exp:
a11.e a21.e a31.e a12.e a22.e a32.e a13.e a23.e a33.e a14.e a24.e a34.e
eval:
a11.lo a21.lo a31.lo a12.lo a22.lo a32.lo a13.lo a23.lo a33.lo a14.lo a24.lo a34.lo
a11.up a21.up a31.up a12.up a22.up a32.up a13.up a23.up a33.up a14.up a24.up a34.up
buf:
a11.b a21.b a31.b a12.b a22.b a32.b a13.b a23.b a33.b a14.b a24.b a34.b
len:
16

FIGURE 2. Storage layout of a $3 \times 4$ multiple-precision matrix ($LDA = 4$) in the GPU memory.

Figure 2 illustrates the column major layout of a $3 \times 4$ multiple-precision matrix. 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 $up := \overline{X/\mathcal{M}}$.

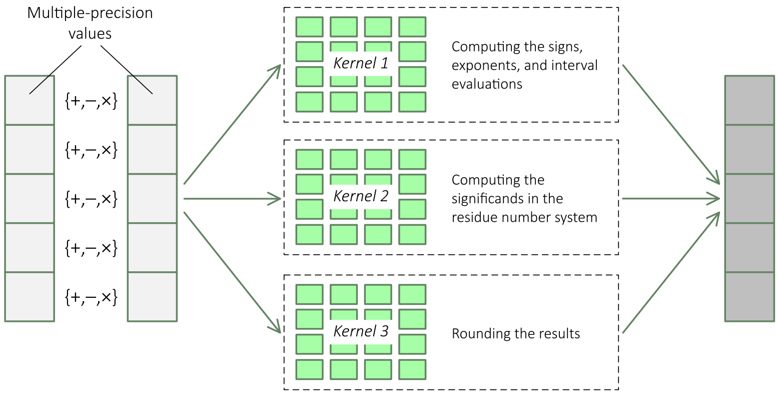
Multiple-precision values
{+,−,×}
{+,−,×}
{+,−,×}
{+,−,×}
{+,−,×}
Kernel 1
Computing the signs, exponents, and interval evaluations
Kernel 2
Computing the significands in the residue number system
Kernel 3
Rounding the results

Figure 3. Performing entrywise operations with multiple-precision arrays using GPU.

## 4. Algorithms for implementing GEMV on GPUs

In [11], the present authors have proposed algorithms for computations with multiple-precision vectors specially designed for GPUs. In these algorithms, arithmetic operations are split into three parts, each of which is executed as a separate CUDA kernel. The parts are as follows:

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

Such a decomposition (see Figure 3) allows one to eliminate branch divergence when performing sequential parts of arithmetic operations with multiple-precision numbers. Furthermore, each CUDA kernel has its own execution configuration (the number of thread blocks and the size of each block) which makes it possible to use all available GPU resources. This approach underlies our GEMV implementation, which is described in Algorithm 1.

As shown in Figure 3, each step in Algorithm 1, except for step 4, consists of running three CUDA kernels. Step 4 is executed by a single kernel, since each multiple-precision addition at this step is computed serially, not in parallel across different RNS moduli.

Thus, performing our multiple-precision GEMV on a GPU involves launching a total of ten CUDA kernels. It is worth noting that the overheads associated with these launches do not significantly contribute to

Algorithm 1. Multiple-precision GEMV

```
1: The vector $x$ is multiplied by the scalar $\alpha$: $d \leftarrow \alpha x$. To store the result vector $d$, a buffer in the global GPU memory is used.
2: The vector $y$ is multiplied by the scalar $\beta$: $y \leftarrow \beta y$.
3: The vector $d$ is considered as the main diagonal of the matrix $D$:
```

$$d = (d_1, d_2, d_3 \dots, d_L) \quad \rightarrow \quad D = \begin{bmatrix} d_1 & 0 & 0 & \dots & 0 \\ 0 & d_2 & 0 & \dots & 0 \\ 0 & 0 & d_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & d_L \end{bmatrix}, \tag{7}$$

```
   where $L = N$ for the non-transposed matrix and $L = M$ for the transposed one. At this step, the matrix $A$ is scaled on the left side or the right side (depending on the operation to be performed) by the diagonal matrix $D$. Right-side scaling ($B \leftarrow AD$) consists in multiplying each $i$th column of the matrix $A$ by the corresponding diagonal element $d_i$. In turn, left-side scaling ($B \leftarrow DA$) consists in multiplying each $i$th row of the matrix $A$ by $d_i$. As a result, an intermediate matrix $B$ of size $M \times N$ is computed, which is stored in a buffer in the global GPU memory.
4: The elements of matrix $B$ are summed by rows or by columns (depending on the operation to be performed), and $\beta y$ is added to the calculated vector of sums.
```

the total computation time, even with small matrices. This is because the multiple-precision operations are themselves quite expensive.

In steps 1 and 2 of Algorithm 1, the multiplication of vectors by scalars can be performed using the SCAL function (Level 1 BLAS). A parallel CUDA implementation of SCAL with numbers of the form (5) is given in [11], so in this section we consider in details only steps 3 and 4. Since different computational schemes are used for the non-transposed and transposed matrix, these cases are considered separately.

### 4.1. The case of non-transposed matrix

The calculation of $y \leftarrow \alpha Ax + \beta y$ is illustrated in Figure 4. Each column of the matrix $B$ is obtained by multiplying the corresponding column of the matrix $A$ by one element of the vector $d = \alpha x$, i.e., the right-side scaling is performed by the diagonal matrix $D$. Then the sum of the elements of the matrix $B$ in each row is computed, and the $i$th element of the vector $\beta y$ is added to the $i$th computed sum.

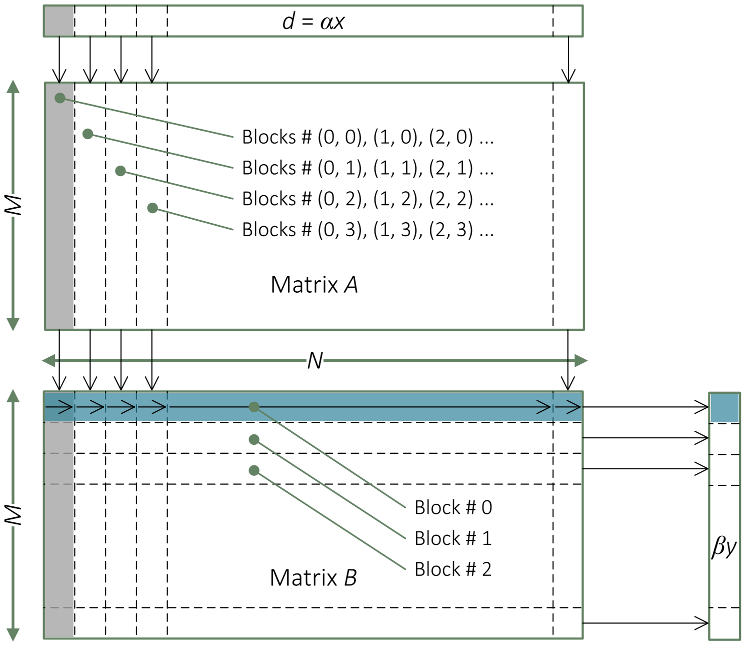
d = αx
Blocks # (0, 0), (1, 0), (2, 0) ...
Blocks # (0, 1), (1, 1), (2, 1) ...
Blocks # (0, 2), (1, 2), (2, 2) ...
Blocks # (0, 3), (1, 3), (2, 3) ...
Matrix A
M
N
M
Block # 0
Block # 1
Block # 2
Matrix B
βy

Figure 4. Performing GEMV for non-transposed matrix $A$

Construction of matrix $B$

As indicated above, the construction of the matrix $B$ is performed by three CUDA kernels: Kernel 1 (computing the signs, exponents, and interval evaluations), Kernel 2 (computing the digits of multiple-precision significands), and Kernel 3 (rounding the elements of the computed matrix). We now describe how these kernels work.

Kernel 1. The computation is performed with a two-dimensional grid of thread blocks. The grid size is $gridDim1 \times gridDim1$, and the size of each block is $blockDim1$, where $gridDim1$ and $blockDim1$ are tunable parameters. The coordinate $blockIdx.y$ defines the offset for the matrix columns. That is, all blocks with $blockIdx.y = i$ (assuming zero-based indexing) participate in the computation of the $i$th column of the matrix $B$, performing the multiplication of the $i$th column of $A$ and the $i$th element of $d = \alpha x$, i.e. $d_i$. Note that $d_i$ is loaded into registers or local memory. If $gridDim1 < N$, then the thread blocks for which $blockIdx.y = i$ calculate the columns of the matrix $B$ with indices $i$, $(i + gridDim1)$, $(i + 2 \times gridDim1)$, etc. In this kernel, one thread is used to calculate the sign, exponent and interval evaluation of one multiple-precision number. Using the `mp_array_t` structure provides grouping of thread accesses to the global GPU memory into transactions.

Kernel 2. The computation is performed on a two-dimensional $gridDim2 \times gridDim2$ grid of thread blocks, where $gridDim2$ is a tunable parameter. The assignment of blocks to the columns of the matrix is similar to that described above for Kernel 1, however, in this case, each thread is associated with its own RNS modulus. Thus, all $n$ digits of the multiple-precision significand are calculated simultaneously by $n$ threads within a single block. Each block can simultaneously compute several multiple-precision significands, depending on the size of the RNS moduli set $n$ and the minimum block size required to maximize the occupancy of a streaming multiprocessor. The size of each thread block is calculated automatically. The minimum block size depends on the Compute Capability of the GPU and is defined as follows [11]:

$$MinBS = MaxThreads/MaxBlocks, \tag{8}$$

where $MaxThreads$ is the maximum number of resident threads per multiprocessor, and $MaxBlocks$ is the maximum number of resident blocks per multiprocessor. For example, if $MinBS = 64$ and $n = 4$, then 16 multiple-precision significands will be calculated simultaneously in each thread block.

Kernel 3. This CUDA kernel is executed with a one-dimensional grid of one-dimensional thread blocks, and the matrix $B$ is considered as a vector of length $M \times N$. The rounding algorithm used and its CUDA implementation are described in [11]. When rounding a number of the form (5), it is necessary to recalculate the interval evaluation of the significand. The employed algorithm for calculating the interval evaluation is given in [27].

### Reduction of matrix $B$

The elements of each row of the matrix $B$ are summed up in accordance with Algorithm 6 from [11]. For this purpose, $M$ thread blocks are used, and in each $i$th block, the elements of the $i$th row of the matrix are loaded into shared memory, after which the reduction by shared memory is performed. The maximum size of each block depends on the precision (the size of the RNS moduli set) and is limited by the available shared memory.

Each multiple-precision addition is performed as a single thread. The array of structures `mp_float_t` is used to store multiple-precision numbers

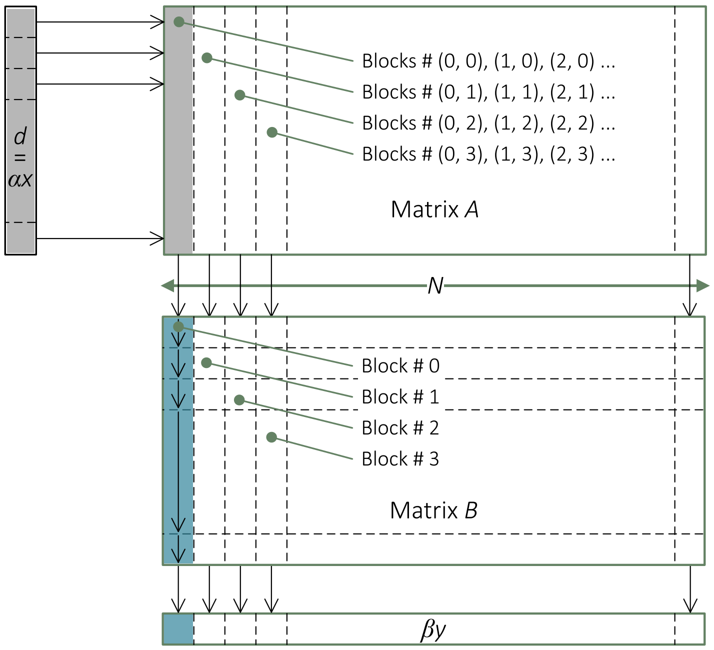
d = αx
Blocks # (0, 0), (1, 0), (2, 0) ...
Blocks # (0, 1), (1, 1), (2, 1) ...
Blocks # (0, 2), (1, 2), (2, 2) ...
Blocks # (0, 3), (1, 3), (2, 3) ...
Matrix A
M
N
Block # 0
Block # 1
Block # 2
Block # 3
Matrix B
M
βy

FIGURE 5. Performing GEMV for transposed matrix $A$

in shared memory. Unlike the mp_array_t structure, which represents a vector of multiple-precision numbers and can only be initialized on the host side, the mp_float_t structure represents a single multiple-precision number and can be initialized on the device side. The conversion between mp_array_t and mp_float_t is done directly during the addition operation and does not incur any overhead.

As a result of the main reduction cycle, the calculated sum of the elements of the $i$th row of matrix $B$ is stored in the shared memory of the $i$th block, and the corresponding element of the vector $y$ multiplied by the scalar $\beta$ is added to this sum. Since each thread block is associated with its own element of $y$, synchronization between thread blocks is not required.

### 4.2. The case of transposed matrix

The calculation of $y \leftarrow \alpha A^T x + \beta y$ is illustrated in Figure 5. In this case, each column of the matrix $B$ is obtained by multiplying the corresponding column of the matrix $A$ by the entire vector $d = \alpha x$, i.e., the left-side scaling is performed by the diagonal matrix $D$. Then the sum of the elements of the matrix $B$ in each column is computed, and the $i$th element of the vector $\beta y$ is added to the $i$th computed sum.

Just as in the non-transposed case, three CUDA kernels (Kernel 1, Kernel 2, and Kernel 3) are launched to compute the matrix $B$. Kernels 1 and 2 are executed on two-dimensional grids, and all thread blocks with $blockIdx.y = i$ participate in the computation of the $i$th column of $B$. Memory access scheme remains the same, and the difference is that each thread operates with its own element of $d$. After the matrix $B$ is computed, the calculation of the $i$th element of the result vector is reduced to summing the elements of the $i$th column of $B$, followed by adding the $i$th element of the vector $y$ multiplied by the scalar $\beta$.

## 5. Accuracy evaluation

In this section, we obtain accuracy estimates for the presented GEMV implementation. Let $M = N$ and let $y \leftarrow \alpha Ax + \beta y$ be the operation being performed. Let us denote the exact result by $y^* = (y_1^*, y_2^*, \dots, y_N^*)$ and the result of applying the GEMV operation by $\hat{y} = (\hat{y}_1, \hat{y}_2, \dots, \hat{y}_N)$. Our goal is to obtain the absolute forward error bound and the relative forward error bound, $\|y^* - \hat{y}\|$ and $\|y^* - \hat{y}\|/\|y^*\|$, respectively, where $\|\ \|$ is the $\ell_1$-norm. In our analysis, we use the standard model of floating-point arithmetic [28, p. 40].

For numbers of the form (5), the identity (6) corresponds (with some restrictions) to the standard model. For the sake of simplicity, we will use the $\theta_n$ notation, which is defined as follows [28, p. 63]:

$$\prod_{i=1}^{n}(1+\delta_i)^{\pm 1} = 1 + \theta_n, \qquad |\theta_n| \le \frac{n\mathbf{u}}{1 - n\mathbf{u}}. \tag{9}$$

Here $\delta_i$ is the relative error introduced by the $i$th floating-point operation from the computation, and $|\delta_i| \le \mathbf{u}$. The purpose of our analysis is to obtain the final error bounds of the GEMV routine, so there is no need to distinguish between the contributions of individual arithmetic operations. Therefore, we further assume $\delta_i \equiv \delta$, $|\delta| \le \mathbf{u}$. According to (6), we have $\mathbf{u} < 4/\sqrt{M}$.

First, the vector $d = \alpha x$ is computed. According to the standard model, the elements of $d$ are expressed as follows:

$$d_i = \mathrm{fl}(\alpha x_i) = \alpha x_i(1+\delta) = \alpha x_i(1+\theta_1), \quad 1 \le i \le N. \tag{10}$$

Hence, the elements of the matrix $B$ can be expressed as follows:

$$b_{ij} = \mathrm{fl}(a_{ij}d_j) = a_{ij}d_j(1+\delta) = \alpha a_{ij}x_j(1+\theta_2), \quad 1 \le i, j \le N. \tag{11}$$

Then the sum of the elements of the matrix $B$ in each row is computed. For the $i$th row, we denote this sum by $s_i$. It is well known that the accuracy of summation depends on the chosen algorithm.

As noted above, $s_i$ is computed on the GPU using a single thread block (one thread block per one row). The used summation algorithm (Algorithm 6 from [11]) implements pairwise summation. However, the maximum number of parallel threads performing summation (i.e., the size of each block) depends on the size of the RNS moduli set, i.e. on the computation precision. When the block size is equal to one, $s_i$ is evaluated sequentially, resulting in the classic recursive summation algorithm:

```
$s_i \leftarrow b_{i1}$
**for** $j = 2$ **to** $N$ **do**
    $s_i \leftarrow \mathrm{fl}(s_i + b_{ij})$
**end for**
```

For this algorithm, the computed value of $s_i$ can be represented in the following form [29]:

$$s_i = (b_{i1} + b_{i2})(1 + \theta_{N-1}) + b_{i3}(1 + \theta_{N-2}) + b_{i4}(1 + \theta_{N-3}) + \cdots + b_{iN}(1 + \theta_1). \tag{12}$$

This is the same as

$$s_i = (\alpha a_{i1}x_1 + \alpha a_{i2}x_2)(1 + \theta_{N+1}) + \alpha a_{i3}x_3(1 + \theta_N) + \alpha a_{i4}x_4(1 + \theta_{N-1}) + \cdots + \alpha a_{iN}x_N(1 + \theta_3)$$

Adding $s_i$ and $\beta y_i(1 + \theta_1)$ gives

$$\hat{y}_i = \mathrm{fl}(s_i + \beta y_i(1 + \theta_1)) = s_i(1 + \theta_1) + \beta y_i(1 + \theta_2), \tag{13}$$

and after substitution we obtain the final expression for the $i$th element of the computed vector:

$$\hat{y}_i = \beta y_i + \sum_{j=1}^{N} \alpha a_{ij}x_j + \beta y_i\theta_2 + \alpha a_{i1}x_1\theta_{N+2} + \alpha a_{i2}x_2\theta_{N+2} + \alpha a_{i3}x_3\theta_{N+1} + \alpha a_{i4}x_4\theta_N + \cdots + \alpha a_{iN}x_N\theta_4.$$

To obtain the upper error bound, it should be assumed that the individual error components ($\theta_i$) are not compensated for each other. Thus, assuming $\theta_i > 0$, the above expression takes the form

$$\hat{y}_i = \beta y_i + \sum_{j=1}^{N} \alpha a_{ij}x_j + \theta_2|\beta y_i| + \theta_{N+2}|\alpha a_{i1}x_1| + \theta_{N+2}|\alpha a_{i2}x_2|$$

$$+\,\theta_{N+1}|\alpha a_{i3}x_3| + \theta_N|\alpha a_{i4}x_4| + \cdots + \theta_4|\alpha a_{iN}x_N|.$$

For further simplification, we will use the notation $\gamma_n$ [28, p. 63], which is defined as follows:

$$\gamma_n := \frac{n\mathbf{u}}{1-n\mathbf{u}} \qquad \text{for } n\mathbf{u} < 1. \tag{14}$$

One of the properties of this notation is $\gamma_n \le \gamma_{n+1}$ (see [28, p. 67]). Considering this property, the above expression for $\hat{y}_i$ can be rewritten as follows

$$\hat{y}_i = \beta y_i + \sum_{j=1}^{N} \alpha a_{ij}x_j + E_N, \tag{15}$$

where $E_n \le \gamma_{N+2}\big(|\beta y_i| + \sum_{j=1}^{N} |\alpha a_{ij}x_j|\big)$.

Since the $i$th element of the exact result vector $y^*$ is expressed in the form $y_i^* = \beta y_i + \sum_{j=1}^{N} \alpha a_{ij}x_j$, we have the following absolute error bound for our GEMV implementation:

$$\|y^* - \hat{y}\| \le \frac{(N+2)\mathbf{u}}{1-(N+2)\mathbf{u}} \cdot \sum_{i=1}^{N}\left(|\beta y_i| + \sum_{j=1}^{N}|\alpha a_{ij}x_j|\right), \tag{16}$$

and the following relative error bound:

$$\frac{\|y^* - \hat{y}\|}{\|y^*\|} \le \frac{(N+2)\mathbf{u}}{1-(N+2)\mathbf{u}} \cdot \frac{\sum_{i=1}^{N}\big(|\beta y_i| + \sum_{j=1}^{N}|\alpha a_{ij}x_j|\big)}{\sum_{i=1}^{N}|\beta y_i + \sum_{j=1}^{N}\alpha a_{ij}x_j|}, \tag{17}$$

where $\mathbf{u} < 4/\sqrt{\mathcal{M}}$ and $\mathcal{M}$ is the product of the RNS moduli.

The same error bounds in a more concise form:

$$\|y^* - \hat{y}\| \le \frac{(N+2)\mathbf{u}}{1-(N+2)\mathbf{u}} \cdot \||\alpha A|\cdot|x| + |\beta y|\|, \tag{18}$$

$$\frac{\|y^* - \hat{y}\|}{\|y^*\|} \le \frac{(N+2)\mathbf{u}}{1-(N+2)\mathbf{u}} \cdot \frac{\||\alpha A|\cdot|x| + |\beta y|\|}{\|\alpha Ax + \beta y\|}. \tag{19}$$

For $M = N$, the bounds obtained are also hold for the operation with the transposed matrix, $y \leftarrow \alpha A^T x + \beta y$.

## 6. Performance results

In this section, we report the results of performance experiments. The experiments were performed on an NVIDIA GeForce GTX 1080 GPU (8 GB GDDR5X, 2560 CUDA cores, Compute Capability 6.1). The host

Table 3. Contribution of individual CUDA kernels to the total execution time of our multiple-precision GEMV.

| Kernel | Comment | % |
|---|---|---|
| mp_vec2scal_mul_esi | Multiplying $x$ by $\alpha$ and $y$ by $\beta$: computing the sings, exponents, and interval evaluations | 0.2 |
| mp_vec2scal_mul_digits | Multiplying $x$ by $\alpha$ and $y$ by $\beta$: computing the digits of multiple-precision significands | 0.1 |
| mp_mat2vec_right_scal_esi | Building the matrix $B$: computing the sings, exponents, and interval evaluations | 5.6 |
| mp_mat2vec_right_scal_digits | Building the matrix $B$: computing the digits of multiple-precision significands | 20.8 |
| mp_vector_round | Rounding the intermediate results | 1.7 |
| mp_matrix_row_sum | Summing the elements in each row of the matrix $B$ and adding the vector $\beta y$ to the result | 71.5 |

system has the following configuration: Intel Core i5 7500 (3.40 GHz) / 16 GB DDR4 RAM / Ubuntu 18.04.4 LTS / CUDA 10.2 / NVIDIA Driver 440.33.01. The source code was compiled using the nvcc compiler with the options -O3 -use_fast_math -Xcompiler=-O3,-fopenmp,-ffast-math.

## 6.1. Performance of individual CUDA kernels

In this experiment, we use the NVIDIA Visual Profiler to evaluate the performance of individual CUDA kernels running in our GEMV implementation. The experiment was carried out with a non-transposed matrix of size $M = N = LDA = 1000$ at 424-bit precision. To achieve this precision, a set of 32 RNS moduli was used, each of 32 bits, with a dynamic range of $\mathcal{M} = 850$ bits. The precision of computations with numbers of the form (5) is $\lfloor \log_2 \sqrt{\mathcal{M}} \rfloor - 1$ bits. It should be noted that multiple roundings can have a significant impact on the overall performance of our GEMV implementation. However, we can always reduce the number of roundings by using a larger set of moduli. In this regard, the input data for the experiment were such that too many roundings were not required.

Table 3 shows the contribution of individual CUDA kernels to the total operation time. We see that the summation of the elements of the matrix $B$ is the most expensive process. It takes 71.5% of the total computation time. Also, a significant contribution is made by the calculation of the digits of multiple-precision significands when computing $B$.

Table 4. Performance metrics for the most expensive kernels

| Kernel | Occupancy, % | Divergence, % | Bandwidth, GB/s |
|---|---|---|---|
| mp_mat2vec_right_scal_digits | 95.5 | 0 | 163.3 |
| mp_matrix_row_sum | 26.8 | 66.8 | 49.9 |

Table 4 shows the actual occupancy, effective global memory bandwidth and branch divergence for the two most expensive kernels. We see that splitting multiple-precision operations into parts leads to significantly more efficient implementations compared to the traditional approach, where each operation with multiple-precision is performed by a single thread, as implemented in the mp_matrix_row_sum kernel.

We note that the NVIDIA 1080 GPU uses GDDR5X RAM with a memory clock rate of 1251 MHz and a 256-bit wide memory interface, so the peak theoretical memory bandwidth of the NVIDIA 1080 GPU is

$$1251 \times 10^6 \times (256/8) \times 8/10^9 = 320.3 \text{ GB/s}. \tag{20}$$

For the mp_mat2vec_right_scal_digits kernel, the effective bandwidth is 51% of the peak bandwidth. This is because calculating each digit of the significand requires finding the remainder of the intermediate result divided by the corresponding modulus (the *mod* operation). However, it is well known that the *mod* operation is one of the most time consuming arithmetic operations. We can speed up the computation with multiple-precision significands in RNS by speeding up the *mod* operation. One way to achieve this is using the Barrett reduction. For the mp_matrix_row_sum kernel, the effective bandwidth is 16% of the peak bandwidth. This is because for each thread block, all intermediate results are stored in the shared memory, and only the final result is loaded into the global memory. However, for this core, shared memory is the limiting factor for fully utilizing the available GPU resources.

## 6.2. Comparison with other implementations

In this experiment, the performance of the presented GEMV implementation was compared with implementations based on the existing CUDA libraries that support multiple precision, namely GARPREC, CAMPARY, and CUMP (see Section 1). All of these libraries are based on the positional number system. To evaluate the practical effect of using the mp_array_t structure and splitting operations with multiple precision into several

Table 5. Execution time (in milliseconds) of GEMV with multiple precision on the GeForce GTX 1080 GPU.

| | Precision in bits | | | | |
|---|---|---|---|---|---|
| | 106 | 212 | 424 | 848 | 1696 |
| Proposed implementation | 3.1 | 5.6 | 8.5 | 12.9 | 24.6 |
| Proposed implementation (transposed) | 2.9 | 4.6 | 7.1 | 11.2 | 19.4 |
| Basic implementation | 12.1 | 22.8 | 42.4 | 75.5 | 150.6 |
| CUMP | 20.1 | 20.8 | 26.8 | 56.8 | 154.0 |
| CAMPARY | 0.7 | 6.0 | 26.3 | 124.4 | 2499.9 |
| GARPREC | 26.5 | 37.4 | 70.2 | 206.0 | 689.1 |

CUDA kernels, we also implemented the classical GEMV algorithm. In this algorithm, each arithmetic operation with multiple-precision numbers of the form (5) is performed by a single thread, and the array of `mp_float_t` instances is used to store an array of multiple-precision numbers. Starting now, this version of GEMV is referred to as “basic implementation”.

The experiments were carried out at a fixed matrix size of $M = N = LDA = 1000$. We considered various levels of arithmetic precision, from 106 to 1696 bits. The input datasets were composed of randomly generated floating-point numbers in the range $[-1, 1]$. To reduce the impact of noise, the GEMV functions under evaluation were repeated several times. The total execution time of all iterations was measured in milliseconds, and then the average execution time of one iteration was calculated.

The experimental results presented in Table 5 show that in many cases, the performance of the proposed implementation is significantly higher than that of existing multiple-precision libraries, and is always higher than that of the basic implementation. As the precision increases, the time of the proposed implementation increases linearly. The operation with the transposed matrix turned out to be slightly faster than the operation with the non-transposed matrix, which is due to the better memory read pattern when accessing the matrix $B$.

At 106-bit precision (quadruple precision), the CAMPARY library is faster than our implementation; however, as the precision increases, the execution time of CAMPARY also increases significantly. This effect is consistent with the estimates for the complexity of arithmetic operations in CAMPARY given in Section 1.

## 7. Conclusion

In this paper, we have presented a parallel implementation of the multiple-precision GEMV operation for systems with CUDA-compatible GPUs. Our implementation uses the original two-level scheme of parallel computations. In this scheme, both all elements of the vector/matrix and all digits of the multiple-precision significand for each element are computed in parallel. This became possible due to the use of the residue number system (RNS) instead of positional number systems.

However, straightforward use of RNS does not have a significant effect, since multiple-precision operations contain sequential and parallel sections, which results in divergent execution paths. To eliminate the drawback, multiple precision arithmetic operations in the proposed GEMV are split into several parts, and each part is performed as a separate CUDA kernel. This splitting increases in the total number of global memory accesses since all intermediate results should be stored in the global memory. However, this overhead is offset by the elimination of branch divergence in the kernels and more efficient utilization of the GPU's resources. Experiments have been carried out that confirm the efficiency of the used computation scheme.

The developed GEMV implementation is part of MPRES-BLAS, a new multiple-precision library for GPUs (available on GitHub). We note that the functionality of MPRES-BLAS is not limited to the basic linear algebra routines. The library provides basic arithmetic operations with multiple precision for CPU and CUDA, so it can even be thought of as a general-purpose multiple-precision arithmetic library. In addition, MPRES-BLAS implements several 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.

We plan to implement automatic tuning of the kernels execution parameters, taking into account the level of arithmetic precision and the size of the problem. We also plan to adapt the approaches discussed in this paper to implement GPU-based sparse matrix-vector multiplication (SpMV) with multiple precision.

## References

[1] M. Courbariaux, Y. Bengio, J. David. *Training deep neural networks with low precision multiplications*, 2014. arXiv 1412.7024 ↑61

[2] D. H. Bailey, J. M. Borwein. "High-precision arithmetic in mathematical physics", *Mathematics*, **3**:2 (2015), pp. 337–367. doi ↑62

[3] J. Daněk, J. Pospíšil. "Numerical aspects of integration in semi-closed option pricing formulas for stochastic volatility jump diffusion models", *International Journal of Computer Mathematics*, **97**:6 (2020), pp. 1268-1292. doi ↑62

[4] Y. Feng, J. Chen, W. Wu. "The PSLQ algorithm for empirical data", *Math. Comp.*, **88**:317 (2019), pp. 1479–1501. doi ↑62

[5] S. Leweke, E. von Lieres. "Fast arbitrary order moments and arbitrary precision solution of the general rate model of column liquid chromatography with linear isotherm", *Comput. Chem. Eng.*, **84** (2016), pp. 350–362. doi ↑62

[6] M. Kyung, E. Sacks, V. Milenkovic. "Robust polyhedral Minkowski sums with GPU implementation", *Comput. Aided Des.*, **67–68** (2015), pp. 48–57. doi ↑62

[7] B. Pan, Y. Wang, S. Tian. "A high-precision single shooting method for solving hypersensitive optimal control problems", *Mathematical Problems in Engineering*, **2018** (2018), 7908378, 11 pp. doi ↑62

[8] Y. Xuan, D. Li, W. Han. "Efficient optimization approach for fast GPU computation of Zernike moments", *Journal of Parallel and Distributed Computing*, **111** (2018), pp. 104–114. doi ↑62

[9] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh. "Basic linear algebra subprograms for Fortran usage", *ACM Trans. Math. Softw.*, **5**:3 (1979), pp. 308—323. doi ↑62

[10] R. Nath, S. Tomov, T. Tim Dong, J. Dongarra. "Optimizing symmetric dense matrix-vector multiplication on GPUs", *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, New York, NY, USA, 2011, pp. 1–10. doi ↑62,64

[11] K. Isupov, V. Knyazkov, A. Kuvaev. "Design and implementation of multiple-precision BLAS Level 1 functions for graphics processing units", *Journal of Parallel and Distributed Computing*, **140** (2020), pp. 25–36. doi ↑63,64,65,66,67,70,71,73,76

[12] A. Omondi, B. Premkumar. *Residue number systems: theory and implementation*, Imperial College Press, London, UK, 2007. ↑63,66

[13] K. Bigou, A. Tisserand. "Single base modular multiplication for efficient hardware RNS implementations of ECC", *Cryptographic hardware and embedded systems – CHES 2015*, eds. T. Güneysu, H. Handschuh, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 123–140. ↑63

[14] A. Abdelfattah, D. Keyes, H. Ltaief. "KBLAS: an optimized library for dense matrix-vector multiplication on GPU accelerators", *ACM Trans. Math. Softw.*, **42**:3 (2016), 18. doi ↑64,68

[15] G. He, J. Gao, J. Wang. "Efficient dense matrix-vector multiplication on

GPU", *Concurrency and Computation: Practice and Experience*, **30**:19 (2018), e4705. doi ↑64

[16] T. Inoue, H. Tokura, K. Nakano, Y. Ito. "Efficient triangular matrix vector multiplication on the GPU", *PPAM 2019: Parallel Processing and Applied Mathematics*, Lecture Notes in Computer Science, vol. **12043**, eds. R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, Springer International Publishing, Cham, 2020, pp. 493–504. doi ↑64

[17] *Quadruple precision BLAS routines for GPU: QPBLAS-GPU ver.1.0. User's manual*, 2013 (accessed 19 May 2019), 58 pp. URL ↑64,66

[18] R. Iakymchuk, S. Collange, D. Defour, S. Graillat. "ExBLAS: reproducible and accurate BLAS library", Numerical Reproducibility at Exascale (NRE2015) workshop held as part of the Supercomputing Conference (SC15) (November 20, 2015, Austin, TX, USA). URL ↑64,66

[19] D. Mukunoki, T. Ogita. "Performance and energy consumption of accurate and mixed-precision linear algebra kernels on GPUs", *Journal of Computational and Applied Mathematics*, **372** (2020), 112701. doi ↑64,66

[20] Y. Hida, X. S. Li, D. H. Bailey. "Algorithms for quad-double precision floating point arithmetic", *Proceedings 15th IEEE Symposium on Computer Arithmetic*, ARITH-15 (11–13 June 2001, Vail, CO, USA), pp. 155–162. doi ↑64

[21] D. E. Knuth. *The art of computer programming*. V. 2: *Seminumerical algorithms*, 3rd ed., Addison-Wesley Longman Publishing Co., Inc., USA, 1997, ISBN 978-0201896848. ↑64

[22] J. R. Shewchuk. "Adaptive precision floating-point arithmetic and fast robust geometric predicates", *Discrete & Computational Geometry*, **18**:3 (1997), pp. 305–363. doi ↑64

[23] T. Ogita, S. M. Rump, S. Oishi. "Accurate sum and dot product", *SIAM J. Sci. Comput.*, **26**:6 (2005), pp. 1955-–1988. doi ↑64

[24] M. Lu, B. He, Q. Luo. "Supporting extended precision on graphics processors", DaMoN'10: Proceedings of the Sixth International Workshop on Data Management on New Hardware (2010, Indianapolis, Indiana, USA), pp. 19–26. doi ↑65,66

[25] M. Joldes, J. Muller, V. Popescu. "Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming", 2017 IEEE 24th Symposium on Computer Arithmetic (ARITH) (24–26 July 2017, London, UK), pp. 27–34. doi ↑65,66

[26] T. Nakayama, D. Takahashi. "Implementation of multiple-precision floating-point arithmetic library for GPU computing", The 23rd IASTED International Conference on Parallel and Distributed Computing and Systems PDCS 2011 (December 14–16 2011, Dallas, USA), pp. 343–349. doi ↑65,66

[27] K. Isupov. "Using floating-point intervals for non-modular computations in residue number system", *IEEE Access*, **8** (2020), pp. 58603–58619. doi ↑67,73

[28] N. J. Higham. *Accuracy and stability of numerical algorithms*, 2nd, SIAM,

Philadelphia, PA, USA, 2002, ISBN 978-0-89871-521-7, xxvii+663 pp. ↑75,77

[29] J. Muller, N. Brunie, F. de Dinechin, C. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, S. Torres. *Handbook of floating-Point arithmetic*, 2, Birkhäuser, Basel, 2018, ISBN 978-3-319-76525-9. ↑76

*Received* 29.04.2020
*Revised* 24.07.2020
*Published* 20.08.2020

Recommended by *ph.d. S. A. Romanenko*

*Sample citation of this publication:*
Konstantin Isupov, Vladimir Knyazkov. "Multiple-precision matrix-vector multiplication on graphics processing units". *Program Systems: Theory and Applications*, 2020, **11**:3(46), pp. 61–84.
10.25209/2079-3316-2020-11-3-61-84
`http://psta.psiras.ru/read/psta2020_3_61-84.pdf`

The same article in Russian: 10.25209/2079-3316-2020-11-3-33-59

*About the authors:*

**Konstantin Isupov**
PhD in Computer Science. Assistant professor at the Department of Electronic Computing Machines, Vyatka State University. Area of scientific interests: high-precision computations, residue number system, computer arithmetic, parallel algorithms, and GPU computing.
0000-0003-0239-0404
**e-mail:** ks_isupov@vyatsu.ru

**Vladimir Knyazkov**
Dr. Sci. in Computer Sciences and Engineering. Principal research scientist at the Research Institute of Fundamental and Applied Studies, Penza State University. Area of scientific interests: parallel architectures, high-performance computing, and reconfigurable computing systems.
0000-0003-3820-6541
e-mail: kniazkov@pnzgu.ru

Эта же статья по-русски: 10.25209/2079-3316-2020-11-3-33-59