IMPLEMENTATION AND PERFORMANCE EVALUATION OF MULTIPLE PRECISION SPARSE MATRIX-VECTOR MULTIPLICATION ON CUDA USING THE RESIDUE NUMBER SYSTEM

K. Isupov, V. Knyazkov*, I. Babeshko, A. Krutikov

Vyatka State University,
610000, Kirov, Russia
*Penza State University,
440026, Penza, Russia

DOI: 10.24411/2073-0667-2021-10005

The solution of large systems of linear equations with sparse matrices is usually done using iterative methods that involve repeated computation of the sparse matrix-vector product (SpMV). The SpMV plays an important role in many scientific and engineering applications, and a lot of effort has been made to implement efficient SpMV kernels for highly parallel computing architectures such as graphics processing units (GPUs). However, round-off errors that occur when using finite precision floating-point arithmetic can cause poor convergence of iterative solvers. In this article, we present SpMV implementations for GPUs using multiple-precision floating-point arithmetic that are part of MPRES-BLAS, an open-source CUDA library of multiple-precision linear algebra operations (available at `https://github.com/kisupov/mpres-blas`).

We first give a brief description of the multiple-precision floating-point number representation supported in MPRES-BLAS. Unlike traditional approaches to expressing high-precision numbers, our representation is based on the residue number system (RNS). The RNS is an unweighted number system that is an alternative to the positional number representation. In the RNS, a number is represented by its residues modulo several pairwise relatively prime integers called the moduli. The RNS is interesting, in particular, because it provides efficient addition, subtraction and multiplication of large integers. the residues of a number are mutually independent, and the calculations with each residue are performed in the ring of integers modulo the corresponding modulus, so that carry chains between the residues are eliminated.

The SpMV performance depends on the format used to store the sparse matrix. In this article, we use the multiple-precision variants of CSR (Compressed Sparse Row) and ELLPACK sparse storage formats. The CSR format is widely used in iterative solvers, since it requires storing only nonzero matrix elements. In turn, the ELLPACK format is efficient when implemented on a GPU, provided the number of nonzero elements in each row of the matrix does not vary significantly.

Our multiple-precision SpMV algorithm consists of two steps. At the first step, the matrix and vector are multiplied element-by-element, and the computed array of component-wise products is stored in a global memory buffer. This step is highly parallelizable thanks to the use of RNS, namely, both all elements of the intermediate array and all digits of the significand (mantissa) for each multiple-precision element are computed in parallel. Each multiple-precision multiplication is performed by n GPU threads, where n is the size of the RNS moduli set. At the second step, the summation kernel is

This research was funded by the Russian Science Foundation grant number 20-71-00046.

© K. Isupov, V. Knyazkov, I. Babeshko, A. Krutikov, 2021

launched, which perform a per-row reduction of the intermediate array to produce the output vector. In this step, each multiple-precision addition is performed by one thread.

In the experimental part of the paper, we evaluated the performance of our multiple-precision SpMV routines on the marine1, largebasis, ESOC, Goodwin_127 and nd6k matrices from SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection).

We compare the proposed SpMV with implementations based on existing multiple-precision general-purpose arithmetic libraries for CUDA-enabled GPUs, namely, CUMP and CAMPARY. We carry out the experiments on an NVIDIA RTX 2080 graphics card (46 streaming multiprocessors, 8 GB of GDDR6 memory, compute capability version 7.5). The results indicate that, with the same precision, the proposed SpMV routines in many cases provide significantly better performance than the CUMP and CAMPARY library implementations. One exception is the 106-bit precision case, where the CAMPARY library is much faster.

Comparison of the CSR and ELLPACK formats shows that ELLPACK outperforms CSR at a moderate level of precision (106-212 bits). With higher arithmetic precision (424-868 bits), the ELLPACK advantages become less apparent. Moreover, the memory requirements of ELLPACK can significantly exceed the requirements of CSR, especially in the context of higher precision. We note that matrices from real-world applications usually consist of single or double precision entries, and there is no need to convert them to higher precision. Considering this aspect, we plan to implement multiplication of a sparse double-precision matrix by a high-precision vector and use it to improve the convergence of iterative solvers.

Key words: Sparsematrices, SpMV, multiple-precisionarithmetic, residue number system, GPU programming.

References

1. Filippone S., Cardellini V., Barbieri D., Fanfarillo A. Sparse Matrix-Vector Multiplication on GPGPUs // ACM Transactions on Mathematical Software. 2017. Vol. 43, N 4. Article No. 30. DOI: 10.1145/3017994.

2. Bell N., Garland M. Efficient Sparse Matrix-Vector Multiplication on CUDA. Technical Report NVR-2008-004. NVIDIA Corporation. 2008. [Electron. Res.]: https://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf (date of access: 10.01.2021).

3. Greathouse J. L., Daga M. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format // Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC14, November 16–21, 2014, New Orleans, LA, USA. IEEE, 2014. P. 769–780. DOI: 10.1109/SC.2014.68.

4. Liu Y., Schmidt B. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs // Proceedings of the 2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP), July 27–29, 2015, Toronto, ON, Canada. IEEE, 2015. P. 82–89. DOI: 10.1109/ASAP.2015.7245713.

5. Sedaghati N., Mu T., Pouchet L.-N., Parthasarathy S., Sadayappan P. Automatic selection of sparse matrix representation on GPUs // Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, June 8–11, 2015 at Newport Beach, CA, USA. ACM, 2015. P. 99–108. DOI: 10.1145/2751205.2751244.

6. Cools S., Yetkin E. F., Agullo E., Giraud L., Vanroose W. Analyzing the Effect of Local Rounding Error Propagation on the Maximal Attainable Accuracy of the Pipelined Conjugate Gradient Method // SIAM Journal on Matrix Analysis and Applications. 2018. Vol. 39, N 1. P. 426–450. DOI: 10.1137/17M1117872.

7. Saito T., Ishiwata E., Hasegawa H. Analysis of the GCR method with mixed precision arithmetic using QuPAT // Journal of Computational Science. 2012. Vol. 3, N 3. P. 87–91. DOI: 10.1016/j.jocs.2011.05.001.

8. Shewchuk J. R. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates // Discrete and Computational Geometry. 1997. Vol. 18, N 3. P. 305–363. DOI: 10.1007/PL00009321.

9. Masui K., Ogino M. Research on the Convergence of Iterative Method Using Mixed Precision Calculation Solving Complex Symmetric Linear Equation // IEEE Transactions on Magnetics. 2020. Vol. 56, N 1. Article no. 7503604. DOI: 10.1109/TMAG.2019.2951280.

10. Mukunoki D., Takahashi D. Using Quadruple Precision Arithmetic to Accelerate Krylov Subspace Methods on GPUs // Lecture Notes in Computer Science. 2014. Vol. 8384. P. 632–642. DOI: 10.1007/978-3-642-55224-3_59.

11. Hishinuma T., Hasegawa H., Tanaka T. SIMD Parallel Sparse Matrix-Vector and Transposed-Matrix-Vector Multiplication in DD Precision // Lecture Notes in Computer Science. 2017. Vol. 10150. P. 21–34. DOI: 10.1007/978-3-319-61982-8_4.

12. Kouya T. A highly efficient implementation of multiple precision sparse matrix-vector multiplication and its application to product-type Krylov subspace methods // International Journal of Numerical Methods and Applications. 2012. Vol. 7, N 2. P. 107–119.

13. Hirokawa Y., Itoh T., Tadano H., Ikuno S. Speedup and numerical evaluation of multiple-precision Krylov subspace method using GPU cluster for large-sparse linear system // Proceedings of the international conference for high performance computing, networking, storage and analysis, SC13, November 17–22, Denver, Colorado, USA. Colorado Convention Center, 2013.

14. Mohan Ananda P. A. Residue Number Systems. Theory and Applications. Cham: Birkhäuser. 2016.

15. Omondi A., Premkumar B. Residue Number Systems: Theory and Implementation. Imperial College Press, 2007.

16. Isupov K. Using Floating-Point Intervals for Non-Modular Computations in Residue Number System // IEEE Access. 2020. Vol. 8. P. 58603–58619. DOI: 10.1109/ACCESS.2020.2982365.

17. 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. 2020. Vol. 140. P. 25–36. DOI: 10.1016/j.jpdc.2020.02.006.

18. Joldes M., Muller J. M., Popescu V. Implementation and Performance Evaluation of an Extended Precision Floating-Point Arithmetic Library for High-Accuracy Semidefinite Programming // Proceedings of the 24th IEEE Symposium on Computer Arithmetic, ARITH, July 24–26, 2017, London, UK. IEEE, 2017. P. 27–34. DOI: 10.1109/ARITH.2017.18.

19. Nakayama T., Takahashi D. Implementation of Multiple-Precision Floating-Point Arithmetic Library for GPU Computing // Proceedings of the 23rd IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS 2011, December 14–16, 2011, Dallas, USA. ACTA Press, 2011. P. 343–349.

РЕАЛИЗАЦИЯ И ОЦЕНКА ПРОИЗВОДИТЕЛЬНОСТИ РАЗРЕЖЕННОГО МАТРИЧНО-ВЕКТОРНОГО УМНОЖЕНИЯ МНОГОКРАТНОЙ ТОЧНОСТИ НА CUDA С ИСПОЛЬЗОВАНИЕМ СИСТЕМЫ ОСТАТОЧНЫХ КЛАССОВ

К. С. Исупов, В. С. Князьков*, И. П. Бабешко, А. К. Крутиков

Вятский государственный университет,
610000, Киров, Россия
*Пензенский государственный университет,
440026, Пенза, Россия

УДК 004.222+004.272.25
DOI: 10.24411/2073-0667-2021-10005

Умножение разреженной матрицы на плотный вектор (SpMV) является основным и наиболее затратным элементом в итерационных методах решения разреженных линейных систем и задач на собственные значения. Эффективная реализация SpMV имеет решающее значение для многих научных и инженерных вычислений, причем важно обеспечить не только высокое быстродействие, но и достаточную точность SpMV, поскольку итерационные методы известны своей чувствительностью к ошибкам округления. В статье мы рассматриваем параллельные реализации SpMV для CUDA-совместимых графических процессоров видеокарт (GPU) с использованием арифметики многократной точности на основе системы остаточных классов (СОК). Основным преимуществом СОК перед позиционными системами счисления является отсутствие переносов между цифрами числа, что позволяет заменить многоразрядные операции группами покомпонентных операций с цифрами небольшой разрядности, которые выполняются без накладных расходов, связанных с обработкой информации о переносах между цифрами. Мы рассматриваем реализации SpMV, основанные на двух широко распространенных форматах хранения разреженной матрицы — CSR и ELLPACK. Экспериментальная оценка с матрицами из реальных приложений показывает, что во многих случаях представленные реализации выполняются быстрее, чем реализации на основе существующих библиотек многократной точности для GPU.

Ключевые слова: разреженные матрицы, SpMV, арифметика многократной точности, система остаточных классов, программирование для графических процессоров.

Введение. Операция разреженного матрично-векторного умножения (SpMV) занимает важное место в алгоритмах линейной алгебры, и многие исследования посвящены разработке форматов хранения разреженных матриц и реализаций SpMV, оптимизированных для параллельных вычислительных платформ, таких как центральные многоядерные процессоры (CPU) и графические процессоры видеокарт (GPU) [1–5]. При этом главным критерием выступает скорость вычислений. С другой стороны, точность SpMV также играет важную роль, поскольку итерационные методы, такие как метод минимальных

Исследование выполнено за счет гранта Российского научного фонда (проект № 20-71-00046).

© К. С. Исупов, В. С. Князьков, И. П. Бабешко, А. К. Крутиков, 2021

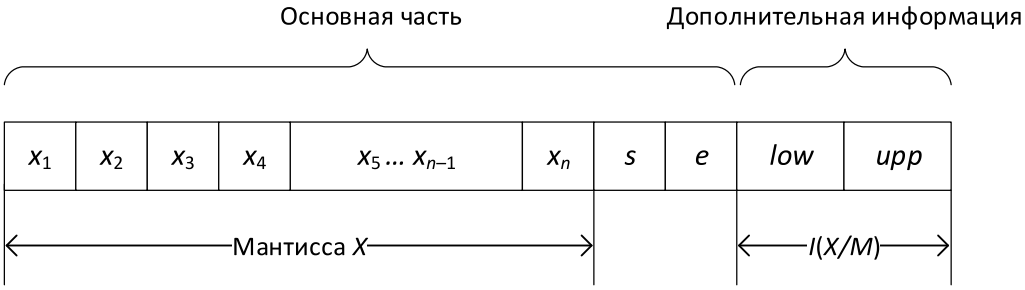


Рис. 1. Многоразрядный формат с плавающей точкой

невязок, метод сопряженных градиентов и метод бисопряженных градиентов, известны своей чувствительностью к ошибкам округления [6, 7].

Существуют различные способы повышения точности вычислений. Распространенным способом является использование формата double-double (DD) [8], в котором каждое число представляется в виде невычисленной суммы двух стандартных чисел с плавающей точкой двойной точности, x_h и x_l, где x_h — число, ближайшее к истинному значению, а x_l — разница между истинным значением и x_h. На основе DD-формата были предложены высокоточные версии SpMV, а также ряд других вычислительных ядер линейной алгебры и итерационных решателей [7, 9–11]. Другим подходом является использование арифметических библиотек многократной точности, таких как GMP/MPFR и CUMP [12, 13].

В статье мы рассматриваем реализации SpMV в форматах CSR (Compressed Sparse Row) и ELLPACK/ITPACK для CUDA-совместимых GPU с использованием арифметики многократной точности на основе непозиционной системы счисления с параллельной структурой — системы остаточных классов (СОК). Основным преимуществом СОК перед позиционными системами счисления является отсутствие переносов между цифрами числа, что позволяет заменить многоразрядные операции группами покомпонентных операций с цифрами небольшой разрядности, которые выполняются эффективно [14, 15]. Эксперименты с матрицами различной структуры показывают, что производительность предложенных реализаций SpMV во многих случаях выше, чем производительность реализаций на основе существующих CUDA-библиотек многократной точности CUMP и CAMPARY. Наши реализация SpMV являются частью библиотеки высокоточных подпрограмм линейной алгебры MPRES-BLAS и доступны на GitHub (`https://github.com/kisupov/mpres-blas`).

1. Представление многоразрядных чисел. Мы используем формат представления многоразрядных чисел (чисел многократной точности) с плавающей точкой, представленный на рис. 1. Число x представляется в виде объекта, состоящего из знака s, мантиссы X, целочисленного порядка (экспоненты) e, а также дополнительной информации о величине мантиссы $I(X/M)$.

Знак интерпретируется обычным образом: он равен нулю, если x — неотрицательное число, и единице, если x — отрицательное. Мантисса представлена в СОК цифрами $(x_1,x_2,\dots,x_n)$ относительно заданного набора оснований (модулей) $\{m_1, m_2, \dots, m_n\}$ и интерпретируется как целое число в интервале от 0 до $M-1$, где M — произведение всех модулей. Цифры мантиссы, которые в терминологии СОК называются остатками, являются такими целыми числами в дополнительном коде, что $x_i = X \bmod m_i$. Примеча-

тельной особенностью СОК является возможность эффективного выполнения операций сложения, вычитания и умножения: данные операции выполняются без возникновения переносов между соседними остатками, т. е. покомпонентно. Следовательно, в отличие от многоразрядной позиционной арифметики, не возникает накладных расходов, связанных с обработкой цепочек переносов, и, более того, все остатки могут вычисляться параллельно.

Преобразование из представленного числового формата в обычную позиционную форму основано на китайской теореме об остатках [15] и выполняется по формуле:

$$x = (-1)^s \times \left| \sum_{i=1}^{n} M_i |x_i w_i|_{m_i} \right|_M \times 2^e,$$

где $M_i = M/m_i$, а w_i — мультипликативная инверсия M_i по модулю m_i.

С другой стороны, имеются сложности, связанные с оценкой величины числа в СОК по остаткам, что требуется для определения знака, обнаружения переполнения диапазона представления мантиссы, сравнения, деления, округления и ряда прочих операций. Для преодоления этих сложностей в формат числа включена дополнительная информация — интервальная оценка мантиссы $I(X/M)$. Эта оценка представляет собой интервал, в пределах которого находится значение мантиссы X, масштабированное относительно произведения модулей M, т. е. X/M. Такой интервал задается двумя границами: low и upp на рис. 1. Границы являются числами с плавающей точкой стандартной разрядности, но с расширенным диапазоном экспоненты, что предотвращает потерю значимости в случае большой величины M (при $M < 2^{1000}$ для хранения low и upp достаточно стандартного формата двойной точности binary64).

Интервальная оценка вычисляется при преобразовании числа в многоразрядный формат, а при выполнении какой-либо арифметической операции с многоразрядными числами интервальная оценка результата вычисляется с использованием формул интервальной арифметики за время $O(1)$. Кроме этого, в любой момент интервальная оценка может быть заново вычислена по остаткам $(x_1, x_2, \ldots, x_n)$ с использованием лишь стандартных операций с плавающей точкой, т. е. без трудоемкого преобразования из СОК в двоичную систему счисления. Подробное изложение метода интервальных оценок для выполнения проблемных операций в СОК представлено в статье [16].

В [17] описаны алгоритмы выполнения основных арифметических операций многократной точности в представленном числовом формате, а также дана следующая граница погрешности округления: если $fl(x \circ y)$ — округленный результат операции $\circ \in \{+, -, \times\}$, то $fl(x \circ y) = (x \circ y)(1 + \delta)$, где $|\delta| < 4/\sqrt{M}$ и M — произведение всех модулей СОК. Таким образом, выбирая необходимый набор модулей СОК, возможно задавать произвольную точность используемой арифметики.

В библиотеке MPRES-BLAS для представления многоразрядных чисел и массивов чисел определены структуры данных, изображенные на рис. 2.

Структура `mp_float_t` представляет одно число. Для хранения в памяти GPU плотных многоразрядных векторов и матриц используется структура `mp_array_t`, в которой вектор из N элементов представляется в виде набора массивов, хранящих отдельные части многоразрядных чисел: все цифры многоразрядных мантисс хранятся в виде единого массива `digits` длины $n \times N$, где n — количество модулей СОК; знаки чисел, экспоненты и интервальные оценки мантисс хранятся в виде отдельных массивов. Дополнительно в структуру включены поля `len` (актуальная длина вектора) и `buf` (буфер с элементами векторного

```
typedef struct {
    int digits[n];
    int sign;
    int exp;
    er_float_t eval[2];
} mp_float_t;
```

```
typedef struct {
    int *digits;
    int *sign;
    int *exp;
    er_float_t *eval;
    int4 *buf;
    int *len;
} mp_array_t;
```

```
typedef struct {
    int *digits;
    int *sign;
    int *exp;
    er_float_t *eval;
} mp_collection_t;
```

Рис. 2. Структуры данных для представления многоразрядных чисел и массивов

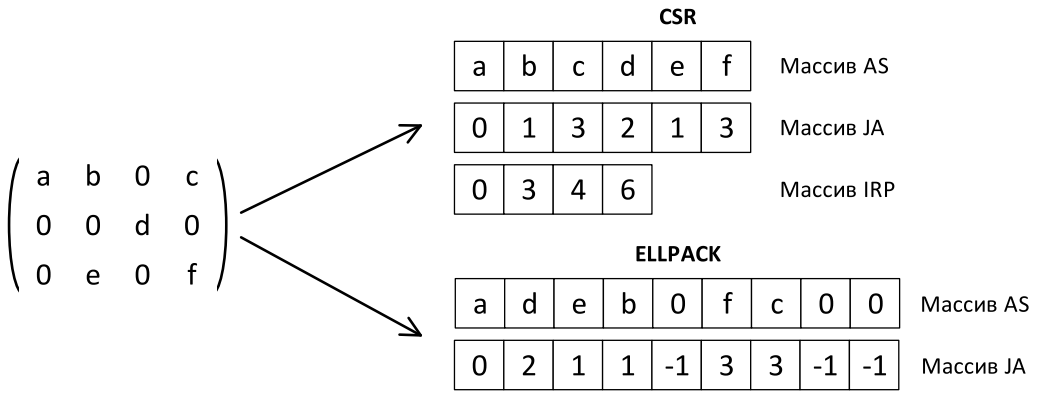


Рис. 3. Форматы CSR и ELLPACK. В качестве индексов нулевых элементов в ELLPACK могут использоваться контрольные значения, например -1

типа int4 из стандартных заголовков CUDA C/C++, который используется для передачи вспомогательных переменных между вычислительными ядрами). Как показано в [17], использование структуры `mp_array_t` совместно с последовательной схемой адресации, при которой все n цифр одного многоразрядного числа располагаются последовательно в памяти GPU, обеспечивает при разрядно-параллельных вычислениях эффективный доступ параллельных потоков к памяти GPU. Для хранения разреженной многоразрядной матрицы служит структура `mp_collection_t`, аналогичная структуре `mp_array_t`, за исключением отсутствия полей `len` и `buf`, которые не используются в ядрах разреженной линейной алгебры.

2. Многоразрядные CSR и ELLPACK форматы. Производительность SpMV зависит от формата, используемого для хранения ненулевых коэффициентов разреженной матрицы. Основными являются координатный, диагональный, Compressed Sparse Row (CSR), ELLPACK/ITPACK и гибридный (HYB) форматы. На основе этих форматов предложены различные производные форматы и реализации SpMV, направленные на эффективное выполнение на GPU. Их обширный обзор представлен в [1]. В данной статье мы используем форматы CSR и ELLPACK, примеры которых приведены на рис. 3, где, подобно [1], AS — массив коэффициентов матрицы, JA — массив индексов столбцов и IRP — массив указателей начала строк.

Формат CSR получил наибольшее распространение в итерационных решателях. В этом формате размер массивов AS и JA равен количеству ненулевых элементов матрицы. Формат ELLPACK эффективен при реализации на GPU, при условии, что число ненулевых

```
                    CSR
AS:
{
   digits = [a.x1 a.x2 a.x3 a.x4
             b.x1 b.x2 b.x3 b.x4
             c.x1 c.x2 c.x3 c.x4
             d.x1 d.x2 d.x3 d.x4
             e.x1 e.x2 e.x3 e.x4
             f.x1 f.x2 f.x3 f.x4]
   sign =    [a.s b.s c.s d.s e.s f.s]
   exp =     [a.e b.e c.e d.e e.s f.e]
   eval =    [a.l b.l c.l d.l e.l f.l
              a.u b.u c.u d.u e.u f.u]
}
JA:  [0 1 3 2 1 3]
IRP: [0 3 4 6]
```

```
                    ELLPACK
AS:
{
   digits = [a.x1 a.x2 a.x3 a.x4
             d.x1 d.x2 d.x3 d.x4
             e.x1 e.x2 e.x3 e.x4
             b.x1 b.x2 b.x3 b.x4
                0    0    0    0
             f.x1 f.x2 f.x3 f.x4
             c.x1 c.x2 c.x3 c.x4
                0    0    0    0
                0    0    0    0]
   sign =    [a.s d.s e.s b.s 0 f.s c.s 0 0]
   exp =     [a.e d.e e.e b.e 0 f.e c.e 0 0]
   eval =    [a.l d.l e.l b.l 0 f.l c.l 0 0
              a.u d.u e.u b.u 0 f.u c.u 0 0]
}
JA: [0 2 1 1 -1 3 3 -1 -1]
```

Рис. 4. Многоразрядные форматы CSR и ELLPACK

элементов в каждой строке матрицы варьируется незначительно [2]. Размер массивов AS и JA в ELLPACK равен $M \times MAXNZR$, где M — число строк и $MAXNZR$ — максимальное число ненулевых элементов в строке матрицы.

На рис. 4 приведен пример представления разреженной матрицы в многоразрядных CSR и ELLPACK форматах. В примере количество модулей СОК $n = 4$, т. е. мантисса каждого элемента $\circ \in \{a,b,c,d,e,f\}$ состоит из четырех остатков: $X = (x_1,x_2,x_3,x_4)$. Символ «.» используется для доступа к отдельным полям многоразрядного числа, а l и u обозначают соответственно нижнюю и верхнюю границы интервальной оценки мантиссы (low и upp на рис.).

3. GPU реализация SpMV многократной точности. Входными данными для наших реализаций SpMV ($y \leftarrow Ax$) являются разреженная матрица A, состоящая из M строк и N столбцов, содержащая NNZ ненулевых элементов, а также плотный вектор x размера N, хранящийся в виде экземпляра структуры `mp_array_t`. Ненулевые элементы матрицы (массив AS) хранятся в виде экземпляра структуры `mp_collection_t`. Дополнительно требуется выделить в глобальной памяти GPU массив BUF (`mp_collection_t`), размер которого равен NNZ для CSR формата и $M \times MAXNZR$ для ELLPACK формата. Результатом SpMV является плотный многоразрядный вектор y размера M (структура `mp_array_t`).

Выполнение операции состоит из двух шагов:

1) Вычисление массива покомпонентных произведений матрицы A и вектора x с сохранением результатов в промежуточный массив BUF.

2) Формирование вектора y путем суммирования массива BUF по строкам.

Использование СОК позволяет реализовать полностью параллельную схему выполнения первого шага SpMV, в соответствии с которой одновременно вычисляются не только элементы массива BUF, но также все цифры многоразрядной мантиссы каждого элемента. Для выполнения каждого умножения многократной точности используется n потоков GPU, где n — размер набора модулей СОК. Для эффективной реализации данной схе-

мы операция многоразрядного умножения разбивается на три этапа, каждый из которых выполняется отдельным вычислительным ядром (глобальной функцией GPU), как предложено в [17]: (1) ядро для вычисления знаков, экспонент и интервальных оценок мантисс, (2) ядро для полностью параллельного вычисления цифр многоразрядных мантисс и (3) ядро для округления.

Таким образом, выполнение SpMV многократной точности состоит из запуска четырех CUDA ядер: три ядра для умножения и одно для построчной редукции промежуточного массива.

В алгоритмах 1–3 представлен псевдокод первых двух ядер умножения и ядра редукции в CSR формате. В алгоритмах 4–6 представлен псевдокод соответствующих ядер для ELLPACK формата, в которых используется проверка контрольного значения индекса столбца для исключения ненужных обращений к памяти. Функции *RoundDown* и *RoundUp* обозначают вычисления с плавающей точкой стандартной точности (double) с расширенным диапазоном экспоненты, выполняемые с округлением „вниз“ и „вверх“ соответственно.

Алгоритм 1. Вычисление знаков, экспонент и интервальных оценок в CSR формате

```
i = blockDim.x * blockIdx.x + threadIdx.x
while i < NNZ do
    id = JA(i)
    BUF.sign(i) = AS.sign(i) xor x.sign(id)
    BUF.exp(i) = AS.exp(i) + x.exp(id)
    BUF.eval(i) = RoundDown(AS.eval(i) * x.eval(id) / ONE.upp)
    BUF.eval(NNZ + i) = RoundUp(AS.eval(NNZ + i) * x.eval(N + id) / ONE.low)
    i += gridDim.x * blockDim.x
end while
```

Алгоритм 2. Вычисление цифр многоразрядных мантисс в CSR формате

```
tid = threadIdx.x mod n                                   ▷ n — размер набора модулей
m = MODULI(tid)
i = blockIdx.x * blockDim.x + threadIdx.x
j = (blockIdx.x * blockDim.x + threadIdx.x) / n
while i < NNZ * n do
    BUF.digits(i) = (AS.digits(i) * x.digits(JA(j) * n + tid)) mod m
    i += gridDim.x * blockDim.x
    j += gridDim.x * blockDim.x / n
end while
```

Алгоритм 3. Редукция промежуточного массива BUF в CSR формате

```
row = threadIdx.x + blockIdx.x * blockDim.x
if row < M then
    sum(threadIdx.x) = 0
    for i = IRP(row):IRP(row + 1) do
        sum(threadIdx.x) += BUF(i)                        ▷ Многоразрядное сложение
    end for
    y(row) = sum(threadIdx.x)
end if
```

Алгоритм округления и его CUDA-реализация ранее описаны в работе [17] и поэтому в данной статье не рассматриваются.

Алгоритм 4. Вычисление знаков, экспонент и интервальных оценок в ELLPACK формате

```
i = blockDim.x * blockIdx.x + threadIdx.x
len = M * MAXNZR
while i < len do
    id = JA(i)
    if id ≥ 0 then
        BUF.sign(i) = AS.sign(i) xor x.sign(id)
        BUF.exp(i) = AS.exp(i) + x.exp(id)
        BUF.eval(i) = RoundDown(AS.eval(i) * x.eval(id) / ONE.upp)
        BUF.eval(len + i) = RoundUp(AS.eval(len + i) * x.eval(N + id) / ONE.low)
    end if
    i += gridDim.x * blockDim.x
end while
```

Алгоритм 5. Вычисление цифр многоразрядных мантисс в ELLPACK формате

```
tid = threadIdx.x mod n                                   ▷ n — размер набора модулей
m = MODULI(tid)
i = blockIdx.x * blockDim.x + threadIdx.x
j = (blockIdx.x * blockDim.x + threadIdx.x) / n
while i < M * MAXNZR * n do
    id = JA(j)
    if id ≥ 0 then
        BUF.digits(i) = (AS.digits(i) * x.digits(id * n + tid)) mod m
    end if
    i += gridDim.x * blockDim.x
    j += gridDim.x * blockDim.x / n
end while
```

Алгоритм 6. Редукция промежуточного массива BUF в ELLPACK формате

```
row = threadIdx.x + blockIdx.x * blockDim.x
if row < M then
    sum(threadIdx.x) = 0
    for i = 0:MAXNZR do
        j = i * M + row
        id = JA(j)
        if id ≥ 0 then
            sum(threadIdx.x) += BUF(j)                    ▷ Многоразрядное сложение
        end if
    end for
    y(row) = sum(threadIdx.x)
end if
```

4. Оценка производительности и затрат памяти. Производительность рассмотренных реализаций SpMV оценивалась на матрицах различной структуры из коллекции Университета Флориды (`https://sparse.tamu.edu`). Описание матриц представлено в табл. 1. Символы M, N, NNZ и $MAXNZR$ обозначают, соответственно, число строк, число столбцов, общее число ненулевых элементов матрицы и максимальное число ненулевых элементов в строке.

Эксперименты выполнялись на вычислительной системе с видеокартой NVIDIA RTX 2080 GPU (46 потоковых мультипроцессоров, 8 Гб памяти GDDR6, версия аппаратной совместимости 7.5), центральным процессором Intel Core i5 7500 и 16 Гб оперативной памяти DDR4, работающей под управлением операционной системы Ubuntu 20.04.1 LTS. Исполь-

Таблица 1

Матрицы для экспериментов

Матрица	M	N	NNZ	$MAXNZR$
marine1	400 320	400 320	6 226 538	18
largebasis	440 020	440 020	5 240 084	14
ESOC	327 062	37 830	6 019 939	19
Goodwin_127	178 437	178 437	5 778 545	62
nd6k	18 000	18 000	6 897 316	514

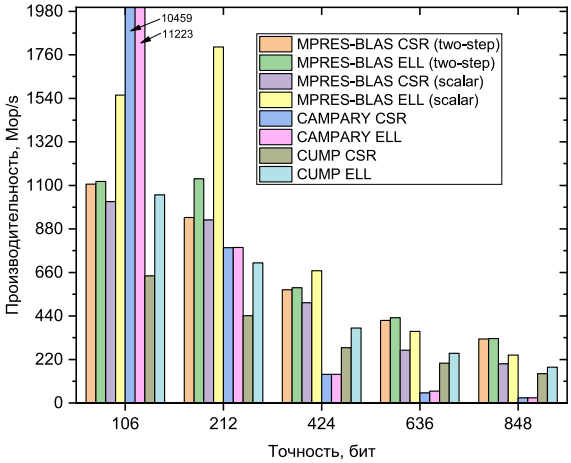
зовались CUDA Toolkit версии 11.1.105 и драйвер NVIDIA версии 455.32.00. Исходный код компилировался с опцией *-O3*.

Оценивалось восемь реализаций SpMV многократной точности, построенных на базе высокоточных CUDA-библиотек MPRES-BLAS, CAMPARY [18] и CUMP [19]:

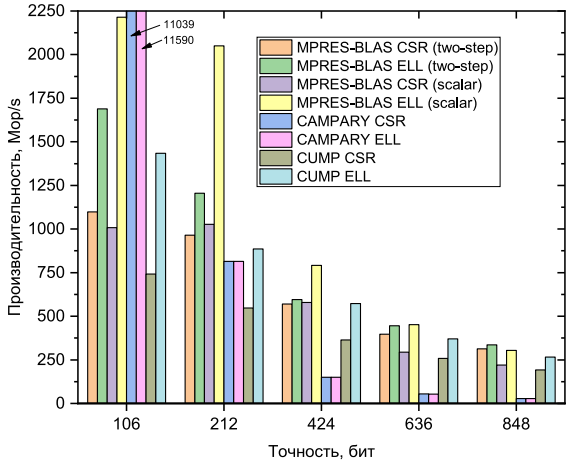
1) *MPRES-BLAS CSR (two-step)* — рассмотренная в предыдущем разделе двухшаговая реализация из библиотеки MPRES-BLAS с использованием формата CSR; 2) *MPRES-BLAS ELL (two-step)* — рассмотренная в предыдущем разделе двухшаговая реализация из библиотеки MPRES-BLAS с использованием формата ELLPACK; 3) *MPRES-BLAS CSR (scalar)* — реализация из библиотеки MPRES-BLAS, полученная из обычного CSR ядра SpMV заменой стандартных операций с плавающей точкой многоразрядными операциями; каждая строка матрицы обрабатывается одним GPU потоком; (см. [2], рис. 20); матрица и векторы хранятся в виде массивов многоразрядных чисел типа `mp_float_t`; 4) *MPRES-BLAS ELL (scalar)* — реализация из библиотеки MPRES-BLAS, полученная из обычного ELLPACK ядра SpMV заменой стандартных операций с плавающей точкой многоразрядными операциями; каждая строка мтарицы обрабатывается одним GPU потоком; (см. [2], рис. 17); матрица и векторы хранятся в виде массивов многоразрядных чисел типа `mp_float_t`; 5) *CAMPARY CSR* — реализация CSR ядра с использованием библиотеки CAMPARY; 6) *CAMPARY ELL* — реализация ELLPACK ядра с использованием CAMPARY; 7) *CUMP CSR* — реализация CSR ядра с использованием библиотеки CUMP; 8) *CUMP ELL* — реализация ELLPACK ядра с использованием библиотеки CUMP.

Производительность измерялась при арифметической точности от 106 до 848 бит. Для достижения заданной точности в MPRES-BLAS использовалось от 8 до 64 модулей СОК. Мерой производительности P является Mop/s — миллион *многоразрядных* операций с плавающей точкой в секунду: $P = 2NNZ/(1000T)$, где T — измеренное время выполнения операции в мс. Заметим, что каждая многоразрядная операция состоит из ряда стандартных операций. Измерялось только время вычислений, т. е. не учитывались время преобразования данных в многоразрядные форматы и время копирования в память GPU. Полученные результаты представлены на рис. 4.

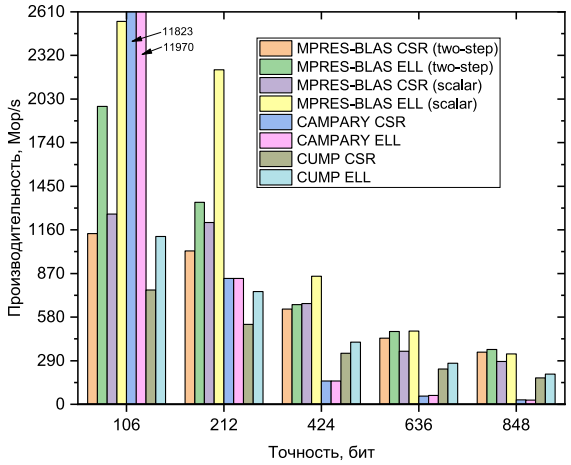
Эффективность многоразрядных реализаций SpMV зависит как от точности, так и от структуры матрицы. Лучшая производительность достигается на матрицах, состоящих из большого числа коротких строк (marine1, largebasis и ESOC), так как такая структура обеспечивает лучшую масштабируемость вычислительных ядер за счет мелкозернистого параллелизма: все строки могут обрабатываться параллельно, и объем вычислений для каждого потока относительно небольшой. В свою очередь, матрицы Goodwin_127 и nd6k состоят из меньшего числа строк, но каждая строка содержит большее число элементов,



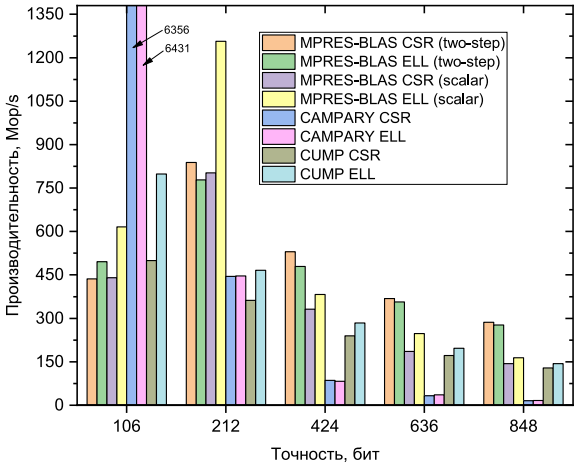
(a) marine1



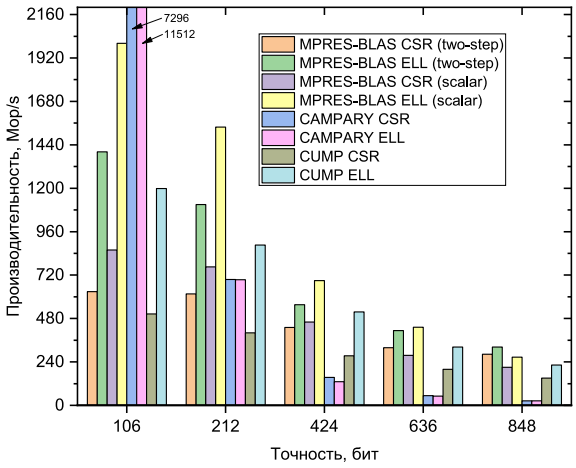
(b) largebasis



(c) ESOC



(d) Goodwin_127



(e) nd6k

Рис. 4. Производительность SpMV многократной точности на NVIDIA RTX 2080

Таблица 2

Размер (МБ) многоразрядной матрицы в форматах CSR и ELLPACK

Точность	marine1		largebasis		ESOC		Goodwin_127		nd6k	
	CSR	ELL	CSR	ELL	CSR	ELL	CSR	ELL	CSR	ELL
106	427	494	381	422	413	426	396	759	473	635
212	617	714	551	610	597	616	573	1097	684	917
424	997	1154	890	986	964	995	925	1772	1105	1482
636	1377	1594	1230	1362	1331	1374	1278	2447	1526	2047
848	1757	2034	1569	1738	1699	1754	1631	3122	1947	2611

что приводит к меньшей производительности, так как увеличивается объем вычислений, выполняемый каждым потоком GPU.

При 106-битной точности самой быстрой оказалась библиотека CAMPARY, реализующая оптимизированные алгоритмы double-double арифметики (для этой точности в экспериментах использовался формат 2-double CAMPARY). Однако при увеличении точности производительность CAMPARY существенно снижается, и при точности от 212 до 848 бит лучшие результаты показывают реализации на основе библиотеки MPRES-BLAS.

При точности от 106 до 424 бит рассмотренные в предыдущем разделе двухшаговые реализации имеют производительность, сравнимую со скалярными SpMV ядрами на основе mp_float_t, либо проигрывают им. Однако при более высокой точности производительность двухшаговых реализаций, как правило, выше, чем производительность скалярных ядер. Наибольшее ускорение (2×) достигнуто при точности 848 бит на матрице Goodwin_127 с использованием CSR формата. Также следует отметить, что на матрице Goodwin_127 производительнсть 106-битных MPRES-BLAS реализаций ниже, чем производительность 212-битных MPRES-BLAS реализаций. Это связано с бóльшим количеством округлений, возникающих в процессе вычислений. Оптимизация округления является важным направлением работ по развитию библиотеки MPRES-BLAS.

В табл. 2 приведен размер массива ненулевых элементов многоразрядной матрицы для реализаций SpMV, рассмотренных в разделе 3. Стоит заметить, что такой же размер имеет буфер, необходимый для хранения промежуточного массива произведений.

Заключение. В работе рассмотрены реализации разреженного матрично-векторного умножения (SpMV) для графических процессоров видеокарт в арифметике многократной точности с использованием системы остаточных классов (СОК), являющиеся частью библиотеки высокоточных подпрограмм линейной алгебры MPRES-BLAS. Непозиционная структура СОК исключает переносы между соседними цифрами многоразрядных мантисс и позволяет вычислять все цифры параллельно. Экспериментальные результаты свидетельствуют о том, что при одинаковой точности разработанные версии SpMV во многих случаях обеспечивают лучшее быстродействие по сравнению с реализациями, основанными на позиционных библиотеках CUMP и CAMPARY. Сравнение форматов CSR и ELLPACK показывает, что ELLPACK превосходит CSR при небольшой точности (106–212 бит). При более высокой точности преимущества ELLPACK становятся менее очевидными, и, более того, затраты памяти ELLPACK могут существенно превышать затраты CSR, в особенности когда речь идет о многоразрядных матрицах.

В дальнейшем запланирована реализация векторных CSR ядер многократной точности, в которых каждая строка матрицы обрабатывается группой потоков GPU. Также

стоит отметить, что матрицы, возникающие в реальных приложениях, как правило, представлены в формате двойной точности (double) и выполнять их преобразование в многоразрядный формат не обязательно. Поэтому в дальнейшем, в рамках развития библиотеки MPRES-BLAS планируются реализация и оценка эффективности операций умножения разреженной матрицы двойной точности на вектор многократной точности, что позволит существенно снизить расходы памяти GPU.

Список литературы

1. Filippone S., Cardellini V., Barbieri D., Fanfarillo A. Sparse Matrix-Vector Multiplication on GPGPUs // ACM Transactions on Mathematical Software. 2017. Vol. 43, N 4. Article No. 30. DOI: 10.1145/3017994.

2. Bell N., Garland M. Efficient Sparse Matrix-Vector Multiplication on CUDA. Technical Report NVR-2008-004. NVIDIA Corporation. 2008. [Electron. Res.]: https://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf (date of access: 10.01.2021).

3. Greathouse J. L., Daga M. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format // Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC14, November 16–21, 2014, New Orleans, LA, USA. IEEE, 2014. P. 769–780. DOI: 10.1109/SC.2014.68.

4. Liu Y., Schmidt B. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs // Proceedings of the 2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP), July 27–29, 2015, Toronto, ON, Canada. IEEE, 2015. P. 82–89. DOI: 10.1109/ASAP.2015.7245713.

5. Sedaghati N., Mu T., Pouchet L.-N., Parthasarathy S., Sadayappan P. Automatic selection of sparse matrix representation on GPUs // Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, June 8–11, 2015 at Newport Beach, CA, USA. ACM, 2015. P. 99–108. DOI: 10.1145/2751205.2751244.

6. Cools S., Yetkin E.F., Agullo E., Giraud L., Vanroose W. Analyzing the Effect of Local Rounding Error Propagation on the Maximal Attainable Accuracy of the Pipelined Conjugate Gradient Method // SIAM Journal on Matrix Analysis and Applications. 2018. Vol. 39, N 1. P. 426–450. DOI: 10.1137/17M1117872.

7. Saito T., Ishiwata E., Hasegawa H. Analysis of the GCR method with mixed precision arithmetic using QuPAT // Journal of Computational Science. 2012. Vol. 3, N 3. P. 87–91. DOI: 10.1016/j.jocs.2011.05.001.

8. Shewchuk J. R. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates // Discrete and Computational Geometry. 1997. Vol. 18, N 3. P. 305–363. DOI: 10.1007/PL00009321.

9. Masui K. Ogino M. Research on the Convergence of Iterative Method Using Mixed Precision Calculation Solving Complex Symmetric Linear Equation // IEEE Transactions on Magnetics. 2020. Vol. 56, N 1. Article no. 7503604. DOI: 10.1109/TMAG.2019.2951280.

10. Mukunoki D., Takahashi D. Using Quadruple Precision Arithmetic to Accelerate Krylov Subspace Methods on GPUs // Lecture Notes in Computer Science. 2014. Vol. 8384. P. 632–642. DOI: 10.1007/978-3-642-55224-3_59.

11. Hishinuma T., Hasegawa H., Tanaka T. SIMD Parallel Sparse Matrix-Vector and Transposed-Matrix-Vector Multiplication in DD Precision // Lecture Notes in Computer Science. 2017. Vol. 10150. P. 21–34. DOI: 10.1007/978-3-319-61982-8_4.

12. Kouya T. A highly efficient implementation of multiple precision sparse matrix-vector multiplication and its application to product-type Krylov subspace methods // International Journal of Numerical Methods and Applications. 2012. Vol. 7, N 2. P. 107–119.

13. Hirokawa Y., Itoh T., Tadano H., Ikuno S. Speedup and numerical evaluation of multiple-precision Krylov subspace method using GPU cluster for large-sparse linear system // Proceedings of the international conference for high performance computing, networking, storage and analysis, SC13, November 17–22, Denver, Colorado, USA. Colorado Convention Center, 2013.

14. Mohan Ananda P. A. Residue Number Systems. Theory and Applications. Cham: Birkhäuser. 2016.

15. Omondi A., Premkumar B. Residue Number Systems: Theory and Implementation. Imperial College Press, 2007.

16. Isupov K. Using Floating-Point Intervals for Non-Modular Computations in Residue Number System // IEEE Access. 2020. Vol. 8. P. 58603–58619. DOI: 10.1109/ACCESS.2020.2982365.

17. 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. 2020. Vol. 140. P. 25–36. DOI: 10.1016/j.jpdc.2020.02.006.

18. Joldes M., Muller J. M., Popescu V. Implementation and Performance Evaluation of an Extended Precision Floating-Point Arithmetic Library for High-Accuracy Semidefinite Programming // Proceedings of the 24th IEEE Symposium on Computer Arithmetic, ARITH, July 24–26, 2017, London, UK. IEEE, 2017. P. 27–34. DOI: 10.1109/ARITH.2017.18.

19. Nakayama T., Takahashi D. Implementation of Multiple-Precision Floating-Point Arithmetic Library for GPU Computing // Proceedings of the 23rd IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS 2011, December 14–16, 2011, Dallas, USA. ACTA Press, 2011. P. 343–349.

Исупов Константин Сергеевич — канд. технич. наук, доцент кафедры электронных вычислительных машин Вятского государственного университета, e-mail: `ks_isupov@vyatsu.ru`. Область профессиональных интересов: вычисления многократной точности, система остаточных классов, компьютерная арифметика, параллельные алгоритмы, программирование для графических процессоров, CUDA. Является автором и соавтором более 30 научных работ в указанных областях.

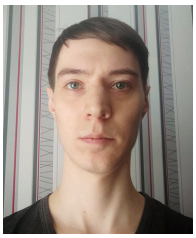
Konstantin Isupov, PhD in Computer Science.Assistant professor at the Department of Electronic Computing Machines, Vyatka State University, e-mail: `ks_isupov@vyatsu.ru`. His research interests include high-precision computation, residue number system, computer arithmetic, parallel algorithms, GPU programming, CUDA. He has authored or coauthored more than 30 papers in these areas.

Князьков Владимир Сергеевич — д-р технич. наук, профессор, главный научный сотрудник научно-исследовательского института фундаментальных и прикладных исследований Пензенского государственного университета, e-mail: `kniazkov@pnzgu.ru`. Является автором более 50 свидетельств на изобретения и патентов, опубликовал более 100 научных работ в области параллельных архитектур, высокопроизводительных вычислений и реконфигурируемых вычислительных систем.

Vladimir Knyazkov, Dr. Sci. in Computer Sciences and Engineering. Principalresearch scientist at the Research Institute of Fundamental and Applied Studies, Penza State University, e-mail: `kniazkov@pnzgu.ru`. He is the author of 50 invention certificates and patents, and published over 100 papers in the fields of parallel architectures, high-performance computing and reconfigurable computing systems.

Бабешко Иван Павлович — аспирант кафедры электронных вычислительных машин Вятского государственного университета, e-mail: `stud120279@vyatsu.ru`. Область научных интересов: высокоточные вычисления, арифме-

тика в остаточных классах, программирование для графических процессоров.



Ivan Babeshko is a postgraduate student at the Department of Electronic Computing Machines, Vyatka State University, e-mail: stud120279@vyatsu.ru. Educational degrees: Master's degree in Informatics and Computer Engineering, 2019, Vyatka State University. His current research interests are multiple-precision arithmetic, residue number system, GPU programming.

Крутиков Александр Константинович — аспирант кафедры электронных вычислительных машин Вятского государственного университета, e-mail: usr09603@vyatsu.ru. Область научных интересов: прогнозирование методами и средствами искусственного интеллекта, искусственные нейронные сети, высокопроизводительные вычисления.

Alexander Krutikov is a postgraduate student at the Department of Electronic Computing Machines, Vyatka State University, e-mail: usr09603@vyatsu.ru. Educational degrees: Master's degree in Informatics and Computer Engineering, 2018, Vyatka State University. His current research interests are artificial neural networks and high-performance computing.

Дата поступления — 03.02.2021