



К. С. Исупов, В. С. Князьков

## Матрично-векторное умножение многократной точности на графическом процессоре

Аннотация. Мы рассматриваем параллельную реализацию матрично-векторного умножения (GEMV, уровень 2 BLAS) для графических процессоров (GPU) с использованием арифметики многократной точности на основе системы остаточных классов. В нашей реализации GEMV покомпонентные операции с многоэлементами векторами и матрицами разбиваются на части, каждая из которых выполняется отдельным CUDA ядром. Это исключает ветвление логики исполнения и позволяет более полно использования ресурсов GPU. Эффективная структура данных для хранения многоэлементами массивов обеспечивает объединение доступов параллельных потоков к глобальной памяти GPU в транзакции. Для предложенной реализации GEMV выполнен анализ ошибок округления и получены оценки точности. Представлены экспериментальные результаты, показывающие высокую эффективность разработанной реализации по сравнению с существующими программными пакетами многократной точности для GPU.

*Ключевые слова и фразы:* вычисления высокой точности, BLAS, GEMV, параллельные алгоритмы, CUDA, GPU, система остаточных классов.

### Введение

Операции с плавающей точкой имеют ошибки округления, возникающие непосредственно во время вычислений. Такие ошибки являются естественными из-за ограниченной длины мантииссы в стандартных форматах IEEE 754 одинарной и двойной точности (binary32 и binary64, соответственно). Для многих задач эти ошибки не препятствуют получению корректного результата вычислений. Более того, для ряда приложений, таких как машинное обучение, лучшим вариантом является использование форматов пониженной (половинной) точности [1]. Вместе с тем, в настоящее время возникает все больше критичных

---

Исследование выполнено за счет гранта Российского научного фонда (проект № 18-71-00063).

© К. С. Исупов<sup>(1)</sup> В. С. Князьков<sup>(2)</sup> 2020

© Вятский государственный университет<sup>(1)</sup> 2020

© Пензенский государственный университет<sup>(2)</sup> 2020

© Программные системы: теория и приложения (дизайн), 2020



к ошибкам округления задач, для которых стандартных форматов IEEE 754 оказывается недостаточно [2–7]. Такие задачи решаются с использованием арифметических библиотек многократной точности, позволяющих выполнять операции с числами, разрядность которых превышает стандартные форматы, а в общем случае ограничена лишь объемом доступной памяти вычислительной системы.

К числу широко известных библиотек многократной точности относятся GMP, MPFR, MPR и MPDECIMAL. Эти и многие другие высокоточные библиотеки ориентированы на системы с центральными процессорами (CPU). Вместе с тем приложения, требующие вычислений повышенной точности, зачастую являются крайне ресурсоемкими и могут быть ускорены за счет использования аппаратных платформ с массивно-параллельной архитектурой, таких как графические процессоры видеокарт (GPU).

Современный графический процессор представляет собой массив потоковых мультипроцессоров, каждый из которых содержит в своем составе множество потоковых процессоров. Массивный параллелизм GPU обеспечивается репликацией общей архитектуры мультипроцессора, причем каждый потоковый процессор в один момент времени может выполнять одну и ту же инструкцию над множеством данных. В 2007 году компания NVIDIA выпустила архитектуру параллельных вычислений CUDA (Compute Unified Device Architecture). Используя CUDA, программисты получили возможность разработки параллельных алгоритмов для GPU на специальном диалекте языка C, без необходимости сложного графического программирования [8]. Производительность последних версий GPU, таких как NVIDIA Tesla V100, сравнима с производительностью 100 CPU, что делает GPU эффективным инструментом для ресурсоемких вычислений.

В статье мы рассматриваем параллельную реализацию матрично-векторного умножения (GEMV) многократной точности для CUDA-совместимых GPU. GEMV входит в состав уровня 2 базовых подпрограмм линейной алгебры (BLAS) [9] и выполняет одну из следующих операций:

$$(1) \quad y \leftarrow \alpha Ax + \beta y \quad \text{или} \quad y \leftarrow \alpha A^T x + \beta y.$$

где  $A$  — плотная  $M \times N$  матрица,  $x$  и  $y$  — векторы,  $\alpha$  и  $\beta$  — скаляры. Наряду с другими BLAS операциями, GEMV является важным строительным блоком для многих алгоритмов линейной алгебры, таких как решатели линейных систем и задач на собственные значения.

Одной из основных стратегий оптимизации вычислений линейной алгебры на GPU является блочное разбиение (blocking), в соответствии с которым матрица или вектор разбивается на небольшие части, которые однократно загружаются в разделяемую память GPU и затем многократно используются в арифметических операциях [10]. Целью является снижение числа обращений к глобальной памяти. Однако при работе с многоразрядными числами интенсивное использование разделяемой памяти (равно как и регистров) часто оказывается неэффективным, так как при большом размере каждого числа снижается масштабируемость и быстродействие CUDA ядер. Поэтому в нашей реализации GEMV используется другой подход, предложенный в [11]. Он основан на разбиении арифметических операций многократной точности на несколько этапов, каждый из которых выполняется отдельным CUDA ядром со своей конфигурацией, причем все цифры многоразрядных чисел вычисляются параллельно. Такой подход приводит к некоторому увеличению числа обращений к глобальной памяти, но вместе с тем обеспечивает высокую производительность и хорошую масштабируемость вычислений многократной точности на GPU по сравнению с традиционной парадигмой, в соответствии с которой каждая арифметическая операция с многоразрядными числами выполняется одним потоком. Для реализации этого подхода мы используем систему остаточных классов (СОК) [12].

В СОК число представляется своими остатками от деления на заданный набор модулей, называемый базисом. Разрядность базиса определяется суммой разрядностей всех модулей [13]. Остатки являются взаимно независимыми, а для операций сложения, вычитания и умножения вычисления с каждым остатком выполняются в кольце целых чисел по соответствующему модулю, так что цепочки переносов между остатками исключаются, как показано на рисунке 1. Это позволяет вычислять все остатки параллельно.

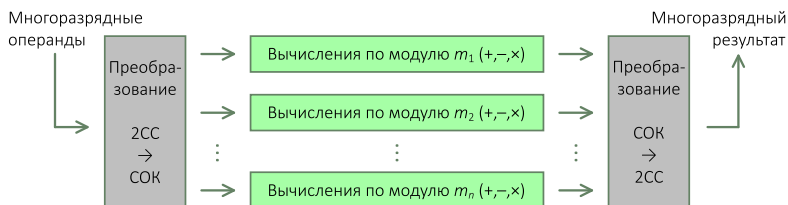


Рисунок 1. Параллельные арифметические свойства СОК

Оставшаяся часть статьи построена следующим образом. В разделе 1 приведен краткий обзор связанных работ. В разделе 2 описан используемый формат чисел многократной точности. В разделе 3 приводится структура данных для хранения многоразрядных массивов и используемая схема размещения матрицы в памяти GPU. В разделе 4 рассматриваются алгоритмы и основные особенности предложенной реализации GEMV, а в разделе 5 оценивается ее точность. Результаты экспериментов представлены в разделе 6, а последний раздел содержит выводы и направления дальнейших исследований.

## 1. Высокоточные вычисления и BLAS для GPU

Операции умножения плотной матрицы на вектор (GEMV, SYMV, TRMV) часто возникают в приложениях научных вычислений, и в литературе рассматриваются эффективные реализации этих операций на GPU с использованием арифметики IEEE 754 [10, 14–16]. В то же время существуют работы, посвященные созданию BLAS реализаций повышенной точности с поддержкой GPU [11, 17–19].

Распространенным в настоящее время способом повышения точности вычислений является использование разложений с плавающей точкой (floating-point expansions): число расширенной точности представляется в виде невычисленной суммы нескольких стандартных чисел с плавающей точкой. Примером таких разложений является известный формат double-double, в котором каждое число представляется в виде невычисленной суммы двух чисел двойной точности (binary64), что соответствует четверной точности (106 бит мантиисы). В свою очередь, формат quad-double обеспечивает восьмерную точность (212 бит мантиисы) за счет использования четырех чисел в формате binary64 для представления каждого числа расширенной точности [20]. Алгоритмы вычисления разложений с плавающей точкой называются безошибочными преобразованиями (error-free transformations) [21, 22]. Арифметика double-double используется в нескольких программных пакетах, например в QPBLAS-GPU [17].

В работе [18] представлен ExBLAS — пакет оптимизированных операций линейной алгебры, обеспечивающий воспроизводимость результатов вычислений. Под воспроизводимостью понимается возможность получения идентичных результатов при многократных запусках программы с одними и теми же входными данными. В ExBLAS для обеспечения воспроизводимости используются безошибочные

преобразования и специальные аккумуляторы — представления с фиксированной точкой большой длины, способные хранить каждый бит информации стандартного формата с плавающей точкой (binary64). Использование аккумуляторов позволяет заменить неассоциативные операции с плавающей точкой операциями с фиксированной точкой, являющимися ассоциативными. ExBLAS предоставляет реализации ряда воспроизводимых операций линейной алгебры для CPU, сопроцессоров Intel Xeon Phi, а также GPU компаний NVIDIA и AMD.

В недавнем исследовании [19] представлены оптимизированные CUDA реализации операций DOT, GEMV, GEMM и SpMV, составляющие пакет BLAS-DOT2. В этих реализациях входные данные хранятся в стандартных форматах binary32 и binary64, а для внутренних вычислений используется двукратная точность: для входных данных в формате binary32 вычисления выполняются в формате binary64; в свою очередь, для входных данных в формате binary64 вычисления выполняются с использованием алгоритма Dot2 [23], основанного на безошибочных преобразованиях.

Также существует ряд арифметических библиотек расширенной или многократной точности для GPU. Поддержка форматов double-double и quad-double реализована в библиотеке GQD [24], которая, помимо основных арифметических операций, позволяет вычислять ряд часто используемых математических функций расширенной точности, таких как квадратный корень, логарифм, экспонента и тригонометрические функции. Алгоритмы GQD по большей части основаны на алгоритмах, реализованных в известной CPU-библиотеке QD<sup>1</sup>. Для хранения чисел расширенной точности в GQD используются предоставляемые CUDA векторные типы double2 и double4.

В библиотеке CAMPARY [25] используются  $n$ -компонентные разложения с плавающей точкой. Данная библиотека предоставляет гибкие CPU и GPU реализации основных арифметических операций многократной точности. В качестве базовых типов для отдельных компонент разложения поддерживаются форматы binary32 и binary64, а точность вычислений (количество компонент в разложении) задается шаблонными параметрами функций. Каждая операция сложения и умножения  $n$ -компонентных разложений в CAMPARY в общем случае требует выполнения, соответственно,  $3n^2 + 10n - 4$  и  $2n^3 + 2n^2 + 6n - 4$  стандартных операций с плавающей точкой, а для double-double арифметики (при использовании двухкомпонентных разложений) применяются оптимизированные алгоритмы.

---

<sup>1</sup><https://www.davidhbailey.com/dhbsoftware>

Библиотеки GARPRES [24] и CUMP [26] поддерживают произвольную точность на GPU с использованием формата “multi-digit”. Число в таком формате представляется в виде последовательности цифр (целых чисел машинной точности) с одной общей экспонентой. Алгоритмы GARPRES аналогичны алгоритмам, реализованным в известном пакете ARPREC<sup>1</sup> для CPU. CUMP, в свою очередь, основана на библиотеке GMP (GNU MP Bignum Library)<sup>2</sup>. Большинство функций CUMP имеют интерфейс, подобный интерфейсу GMP. В GARPRES и CUMP каждая высокоточная операция реализована в виде одного потока, а для объединения доступов к памяти GPU в транзакции используется схема интервальной индексации.

В [11] авторы данной статьи предложили новые алгоритмы арифметики многократной точности на основе СОК, а также оптимизированные алгоритмы для параллельной обработки векторов многократной точности на GPU, которые в дальнейшем были расширены и адаптированы для обработки матриц. На основе этих алгоритмов разработан новый высокоточный пакет базовых подпрограмм линейной алгебры для GPU — MPRES-BLAS. Проведенные эксперименты показали, что во многих случаях MPRES-BLAS обладает более высоким быстродействием по сравнению с реализациями, основанными на существующих позиционных библиотеках многократной точности для CPU и GPU. Рассматриваемая в данной статье реализация GEMV является частью MPRES-BLAS.

Таблица 1 содержит сводную информацию по рассмотренному программному обеспечению со ссылками на исходный код.

ТАБЛИЦА 1. Высокоточное программное обеспечение с поддержкой GPU

Пакет	Ссылка	Исходный код
QPBLAS-GPU	[17]	<a href="https://ccse.jaea.go.jp/software/QPBLAS-GPU">https://ccse.jaea.go.jp/software/QPBLAS-GPU</a>
ExBLAS	[18]	<a href="https://github.com/riakymch/exblas">https://github.com/riakymch/exblas</a>
BLAS-DOT2	[19]	<a href="http://www.math.twcu.ac.jp/ogita/post-k/results.html">http://www.math.twcu.ac.jp/ogita/post-k/results.html</a>
GQD	[24]	<a href="https://code.google.com/archive/p/gpuprec">https://code.google.com/archive/p/gpuprec</a>
CAMPARY	[25]	<a href="http://homepages.laas.fr/mmjoldes/campary">http://homepages.laas.fr/mmjoldes/campary</a>
GARPRES	[24]	<a href="https://code.google.com/archive/p/gpuprec">https://code.google.com/archive/p/gpuprec</a>
CUMP	[26]	<a href="https://github.com/skystar0227/CUMP">https://github.com/skystar0227/CUMP</a>
MPRES-BLAS	[11]	<a href="https://github.com/kisupov/mpres-blas">https://github.com/kisupov/mpres-blas</a>

<sup>2</sup><https://gmpilib.org>

## 2. Представление многоразрядных чисел с плавающей точкой с использованием СОК

Система остаточных классов задается набором из  $n$  попарно взаимно простых целых чисел (модулей)  $\{m_1, m_2, \dots, m_n\}$ . Динамический диапазон СОК определяется произведением модулей<sup>3</sup>

$$(2) \quad \mathcal{M} = m_1 \cdot m_2 \cdot \dots \cdot m_n.$$

Целое число  $X \in [0, \mathcal{M} - 1]$  однозначным образом представляется в СОК остатками  $(x_1, x_2, \dots, x_n)$ , где  $x_i = |X|_{m_i}$  соответствует операции  $X \bmod m_i$ . Преобразование из СОК в двоичную систему выполняется с использованием китайской теоремы об остатках (КТО)[12]:

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

где  $\mathcal{M}_i$  и  $w_i$  — константы, такие что  $\mathcal{M}_i = \mathcal{M}/m_i$  и  $w_i$  — мультипликативная инверсия<sup>4</sup>  $\mathcal{M}_i$  по модулю  $m_i$ .

В отличие от операций сложения, вычитания и умножения, которые выполняются параллельно над всеми остатками  $x_i$ , операции сравнения (за исключением случая сравнения на равенство), определения переполнения, вычисления знака, масштабирования и деления, называемые “немодульными”, являются трудоемкими для СОК. Классический метод выполнения немодульных операций, основанный на КТО, становится неэффективным при больших динамических диапазонах, состоящих из сотен и тысяч бит. Поэтому мы используем альтернативный метод, основанный на интервальной оценке дробного представления числа в СОК [27]. Для  $X$ , заданного остатками  $(x_1, x_2, \dots, x_n)$ , дробное представление (относительная величина) вычисляется следующим образом:

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

Интервальная оценка для  $X/\mathcal{M}$  обозначается  $I(X/\mathcal{M}) = [\underline{X}/\underline{\mathcal{M}}, \overline{X}/\overline{\mathcal{M}}]$  и представляет собой такой интервал, что  $\underline{X}/\underline{\mathcal{M}} \leq X/\mathcal{M} \leq \overline{X}/\overline{\mathcal{M}}$ .

<sup>3</sup>Обычно для обозначения произведения модулей СОК используют символ  $M$ . В данной работе, чтобы избежать коллизии с количеством строк матрицы, мы используем символ  $\mathcal{M}$  для обозначения произведения модулей СОК.

<sup>4</sup>Если  $x$  является ненулевым целым числом, то  $y$  называется мультипликативной инверсией  $x$  по модулю  $m$ , если  $|x \times y|_m = 1$ .

Границами этого интервала,  $\underline{X/M}$  и  $\overline{X/M}$ , являются числа с плавающей точкой машинной точности, и для их вычисления требуются только стандартные арифметические операции, вне зависимости от размера набора модулей СОК и величины динамического диапазона. С использованием интервальных оценок разработаны эффективные алгоритмы выполнения ряда немодульных операций в СОК, включая сравнение по величине и общее деление [27].

Мы используем следующий способ представления многоразрядных чисел с плавающей точкой:

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

где  $s \in \{0, 1\}$  — знак числа,  $e$  — экспонента (порядок) и  $x_1, x_2, \dots, x_n$  — цифры (остатки) мантиисы  $X$ , представленной в СОК, которая принимает значения в диапазоне от 0 до  $\mathcal{M} - 1$ . Каждая цифра  $x_i = X \bmod m_i$  является целым машинным числом со знаком.

В качестве дополнительной информации в числовой формат включена интервальная оценка мантиисы,  $I(X/M)$ , которая позволяет реализовать эффективное сравнение, вычисление знака результата, выравнивание экспонент чисел и оценку необходимости округления.

В предыдущих работах авторов рассматривались различные способы выполнения арифметических операций с числами вида (5). В работе [11] предложены улучшенные алгоритмы сложения и умножения многократной точности, а также получена следующая граница относительной ошибки вычислений. Если  $\text{fl}(x \circ y)$  — округленный результат операции  $\circ \in \{+, -, \times\}$  с числами вида (5), то

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

В разделе 5 мы используем (6) для оценки точности разработанной операции матрично-векторного умножения.

### 3. Размещение данных

В нашей реализации GEMV используется column major формат хранения матрицы, который является стандартным для библиотек BLAS и LAPACK [14]. Для того, чтобы обеспечить объединение доступов к глобальной памяти GPU в транзакции (coalesced access), массив многоразрядных чисел (вектор или матрица) хранится в виде структуры `mp_array_t`, описание которой представлено в таблице 2.



Таблица 2. Спецификация структуры `mp_array_t` для хранения массива многоразрядных чисел; здесь  $n$  — число модулей СОК,  $L$  — размер массива

Поле	Описание
<code>digits</code>	Массив размера $n \times L$ , который хранит цифры многоразрядных мантисс. Все цифры одного числа хранятся последовательно в памяти.
<code>sign</code>	Целочисленный массив размера $L$ , который хранит знаки чисел.
<code>exp</code>	Целочисленный массив размера $L$ , который хранит экспоненты чисел.
<code>eval</code>	Массив чисел с плавающей точкой с расширенной экспонентой размера $2L$ , который хранит интервальные оценки мантисс. Первые $L$ элементов хранят нижние границы интервальных оценок, а вторые $L$ — верхние.
<code>buf</code>	Массив (буфер) длины $L$ с элементами векторного типа <code>int4</code> из стандартных заголовков <code>CUDA C/C++</code> . Используется в операциях сложения и вычитания для передачи вспомогательных переменных между вычислительными ядрами.
<code>len</code>	Количество элементов в многоразрядном массиве ( $L$ ). Для вектора длины $N$ массив должен содержать не менее $(1 + (N - 1) \times  incx )$ элементов, где <code>incx</code> определяет расстояние между двумя соседними элементами вектора; для матрицы размера $M \times N$ , массив должен содержать не менее $LDA \times N$ элементов, где <code>LDA</code> определяет ведущую размерность матрицы; значение <code>LDA</code> должно быть не менее $\max(1, M)$ .

На рисунке 2 приведен пример размещения многоэлемента матрицы  $3 \times 4$  в памяти GPU. В этом примере  $n = 4$ , т.е. мантисса каждого многоэлемента  $a_{ij}$  состоит из четырех цифр:  $X = (x_1, x_2, x_3, x_4)$ . Символ “.” используется для доступа к отдельным частям элементов. Символы *lo* и *up* обозначают нижнюю и верхнюю границы интервальной оценки мантиссы, соответственно, т.е.  $lo := \underline{X/M}$  и  $up := \overline{X/M}$ .

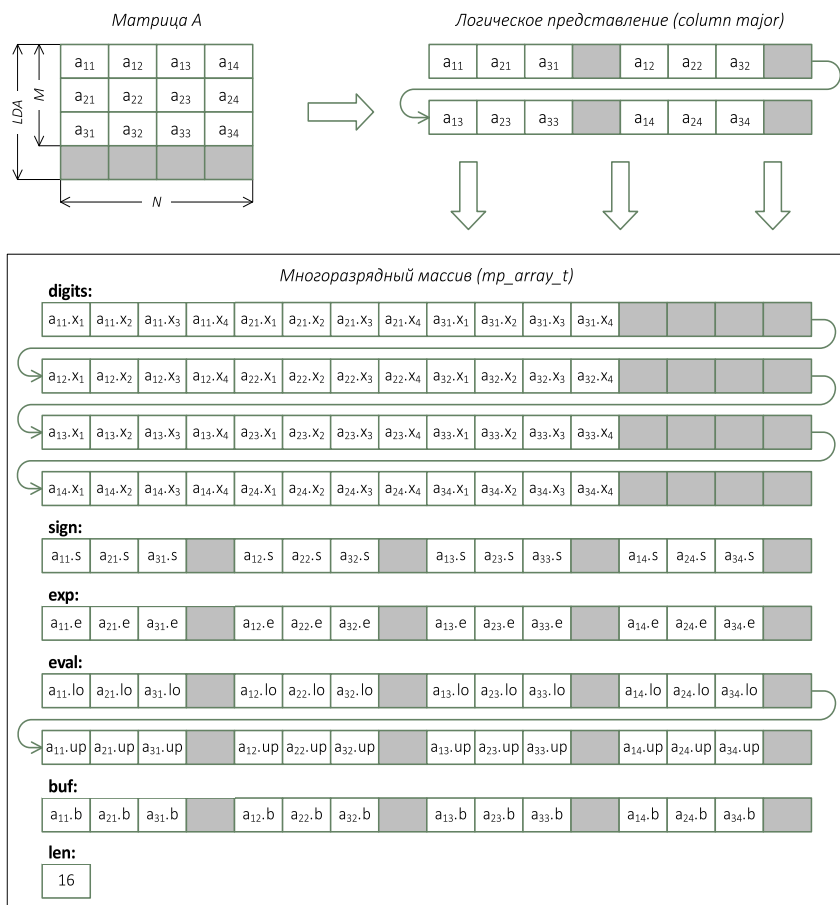


Рисунок 2. Размещение многоэлемента матрицы  $3 \times 4$  ( $LDA = 4$ ) в памяти GPU.

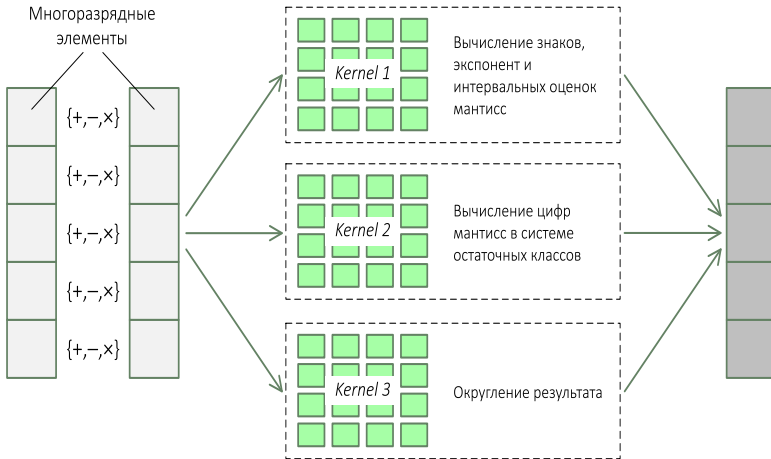


РИСУНОК 3. Выполнение покомпонентных операций с массивами многократной точности на GPU.

#### 4. Алгоритмы реализации GEMV на GPU

В работе [11] авторы данной статьи предложили алгоритмы для вычислений с векторами многократной точности, ориентированные на GPU. В предложенных алгоритмах арифметические операции разбиваются на следующие части, каждая из которых выполняется как отдельное CUDA ядро:

- вычисление знаков, экспонент и интервальных оценок;
- вычисление цифр (остатков) многоразрядных мантисс в СОК;
- округление.

Такая декомпозиция (см. рисунок 3) исключает ветвления при выполнении последовательных этапов арифметических операций с многоразрядными числами. Кроме этого, для каждого вычислительного ядра задается своя конфигурация запуска (количество блоков потоков и размер блока), что позволяет эффективно задействовать все имеющиеся ресурсы GPU. Данный подход лежит в основе нашей реализации GEMV, описание которой представлено в алгоритме 1. Каждый шаг алгоритма, кроме шага 4, состоит из запуска трех CUDA ядер. Шаг 4 выполняется одним ядром, так как каждая отдельная операция сложения многократной точности на этом шаге вычисляется последовательно (без распараллеливания по модулям СОК).

---

 АЛГОРИТМ 1. Операция GEMV многократной точности
 

---

- 1: Вектор  $x$  умножается на скаляр  $\alpha$ :  $d \leftarrow \alpha x$ . Для хранения резульатного вектора  $d$  используется буфер в глобальной памяти GPU.
- 2: Вектор  $y$  умножается на скаляр  $\beta$ :  $y \leftarrow \beta y$ .
- 3: Вектор  $d$  рассматривается как главная диагональ матрицы  $D$ :

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

где  $L = N$  для нетранспонированной матрицы и  $L = M$  для транспонированной матрицы. На данном шаге матрица  $A$  масштабируется с левой или с правой стороны (в зависимости от типа выполняемой операции) диагональной матрицей  $D$ . Масштабирование справа ( $B \leftarrow AD$ ) заключается в умножении каждого  $i$ -го столбца матрицы  $A$  на соответствующий диагональный элемент  $d_i$ . Масштабирование слева ( $B \leftarrow DA$ ) заключается в умножении каждой  $i$ -й строки матрицы  $A$  на соответствующий диагональный элемент  $d_i$ . В результате этой операции формируется промежуточная матрица  $B$  размера  $M \times N$ , для хранения которой используется буфер в глобальной памяти GPU.

- 4: Элементы матрицы  $B$  суммируются по строкам или по столбцам, в зависимости от выполняемой операции, и вычисленный вектор сумм складывается с  $\beta y$ .
- 

Таким образом, выполнение операции GEMV многократной точности на GPU заключается в запуске в общей сложности десяти CUDA ядер. Стоит отметить, что накладные расходы, связанные с этими запусками, не вносят существенного вклада в общее время вычислений даже при небольших матрицах, поскольку операции многократной точности сами по себе достаточно затратные.

Умножение векторов на скаляры на шагах 1 и 2 алгоритма 1 может быть выполнено с использованием функции SCAL (уровень 1 BLAS), параллельная CUDA реализация которой с использованием чисел вида (5) рассмотрена в [11]. Поэтому мы более подробно остановимся лишь на шагах 3 и 4. Так как для нетранспонированной и транспонированной матрицы используются различные схемы вычислений, далее эти случаи рассматриваются по отдельности.

### 4.1. Нетранспонированная матрица

Схема вычисления  $y \leftarrow \alpha Ax + \beta y$  представлена на рисунке 4. Каждый столбец матрицы  $A$  умножается на один элемент вектора  $d = \alpha x$  (т.е. выполняется масштабирование диагональной матрицей справа), в результате чего формируется соответствующий столбец матрицы  $B$ . Затем производится суммирование элементов матрицы  $B$  по строкам с прибавлением  $i$ -го элемента вектора  $\beta y$  к  $i$ -й вычисленной сумме.

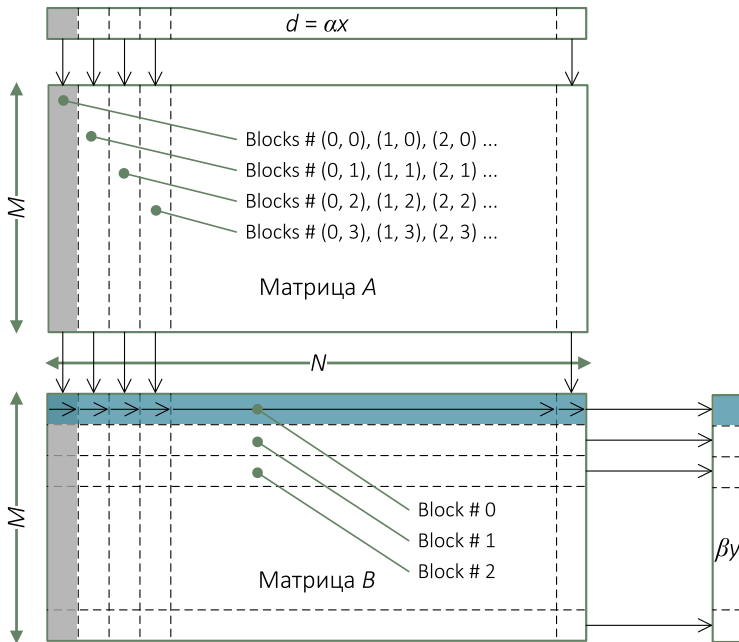


Рисунок 4. Схема выполнения GEMV для нетранспонированной матрицы  $A$

#### Вычисление матрицы $B$

Как указано выше, вычисление матрицы  $B$  производится тремя ядрами CUDA: Kernel 1 (вычисление знаков, экспонент и интервальных оценок), Kernel 2 (вычисление цифр многоразрядных мантисс) и Kernel 3 (округление элементов вычисленной матрицы). Далее рассматриваются принципы работы каждого ядра.

KERNEL 1. Вычисления выполняются на двумерной сетке блоков потоков. Размер сетки —  $gridDim1 \times blockDim1$ ; размер каждого блока равен  $blockDim1$ . Параметры  $gridDim1$  и  $blockDim1$  являются настраиваемыми. Координата  $blockIdx.y$  определяет смещение по столбцам матрицы, т.е. все блоки для которых  $blockIdx.y = i$  (индексация начинается с нуля) участвуют в формировании  $i$ -го столбца матрицы  $B$ , выполняя умножение  $i$ -го столбца матрицы  $A$  и  $i$ -го элемента вектора  $d = \alpha x$ , который предварительно загружается в регистры или локальную память потока. Если  $gridDim1 < N$ , то блоки для которых  $blockIdx.y = i$  вычисляют в цикле столбцы матрицы с индексами  $i$ ,  $(i + blockDim1)$ ,  $(i + 2 \times blockDim1)$  и т.д.

В данном CUDA ядре знак, экспонента и границы интервальной оценки одного многозначного числа вычисляются одним потоком. Благодаря использованию структуры `mp_array_t` для хранения многозначных массивов, доступы потоков к глобальной памяти GPU объединяются в транзакции.

KERNEL 2. Вычисления выполняются на двумерной  $gridDim2 \times blockDim2$  сетке блоков потоков, где  $gridDim2$  — настраиваемый параметр. Назначение блоков столбцам матрицы аналогично описанному выше для Kernel 1, однако в данном случае каждый поток ассоциирован со своим модулем СОК, и таким образом все  $n$  цифр многозначной мантиссы вычисляются одновременно  $n$  потоками в рамках одного блока, причем в каждом блоке потоков могут вычисляться одновременно несколько многозначных мантисс, в зависимости от соотношения между величиной набора модулей системы остаточных классов  $n$  и минимальным размером блока, необходимым для полной загрузки потокового мультипроцессора GPU.

Для данного вычислительного ядра размер блока потоков рассчитывается автоматически. Минимальный размер блока зависит от версии аппаратной совместимости GPU (Compute Capability) и определяется следующим образом [11]:

$$(8) \quad MinBS = MaxThreads / MaxBlocks,$$

где  $MaxThreads$  и  $MaxBlocks$  — максимальное количество резидентных потоков и блоков в мультипроцессоре, соответственно. Например, если  $MinBS = 64$  и  $n = 4$ , то в каждом блоке потоков будут одновременно вычисляться цифры для 16 многозначных мантисс.

KERNEL 3. Данное вычислительное ядро выполняется на одномерной сетке одномерных блоков потоков, причем матрица  $B$  рассматривается как вектор длины  $M \times N$ . Используемый алгоритм округления и его CUDA реализация рассмотрены в [11]. При округлении числа вида (5) необходимо заново вычислять интервальную оценку мантиссы. Актуальный алгоритм вычисления интервальной оценки представлен в [27].

### Редукция матрицы $B$

Суммирование элементов каждой строки матрицы  $B$  выполняется в соответствии с Алгоритмом 6 из [11]. Для этого запускается  $M$  блоков потоков, и в каждом  $i$ -м блоке элементы  $i$ -й строки матрицы загружаются в разделяемую память, после чего выполняется редукция по разделяемой памяти. Максимальный размер каждого блока зависит от точности вычислений (размера модулей СОК) и ограничен доступным объемом разделяемой памяти.

Каждая операция сложения многократной точности выполняется одним потоком. Для хранения многоразрядных чисел в разделяемой памяти используется массив структур `mp_float_t`. В отличие от структуры `mp_array_t`, которая представляет вектор многоразрядных чисел и может быть инициализирована только на стороне хоста, `mp_float_t` представляет одно многоразрядное число и может быть инициализирована на стороне GPU. Преобразование между `mp_array_t` и `mp_float_t` осуществляется непосредственно при выполнении операции сложения и не приводит к накладным расходам.

В результате основного цикла редукции в разделяемой памяти  $i$ -го блока хранится вычисленная сумма элементов  $i$ -й строки матрицы  $B$ , к которой затем прибавляется соответствующий элемент вектора  $y$ , предварительно умноженный на скаляр  $\beta$ . Так как каждый блок потоков ассоциирован со своим элементом вектора  $y$ , то синхронизация между блоками не требуется.

## 4.2. Транспонированная матрица

Схема вычисления  $y \leftarrow \alpha A^T x + \beta y$  представлена на рисунке 5. В данном случае каждый столбец матрицы  $A$  покомпонентно умножается на весь вектор  $d = \alpha x$  (иначе говоря, выполняется масштабирование

матрицы  $A$  диагональной матрицей  $D$  слева), в результате чего формируется соответствующий столбец матрицы  $B$ . Затем производится суммирование элементов матрицы  $B$  по столбцам с прибавлением  $i$ -го элемента вектора  $\beta y$  к  $i$ -й вычисленной сумме.

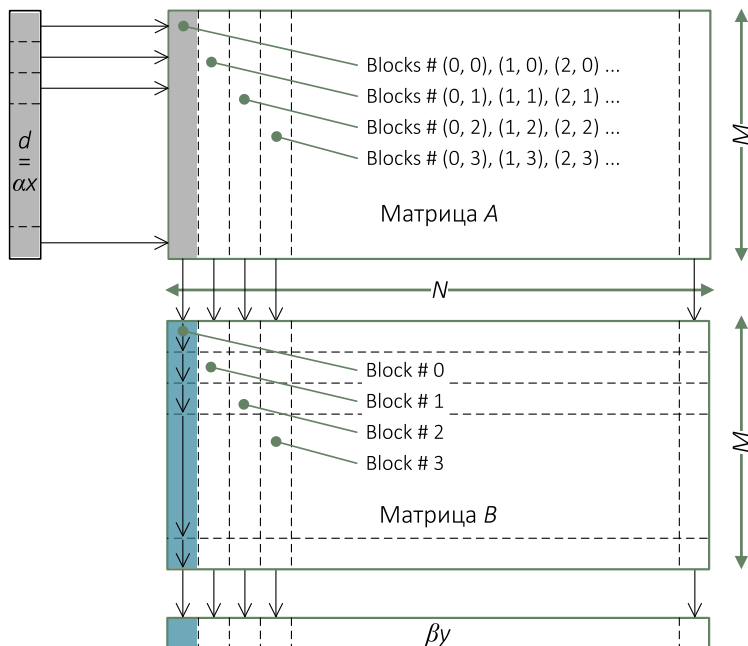


РИСУНОК 5. Схема выполнения GEMV для транспонированной матрицы  $A$

Также как и в первом случае, для вычисления матрицы  $B$  производится запуск трех вычислительных ядер — Kernel 1, Kernel 2 и Kernel 3, причем Kernel 1 и Kernel 2 выполняются на двумерных сетках из блоков потоков, а все блоки с индексом  $blockIdx.y = i$  участвуют в формировании  $i$ -го столбца матрицы  $B$ . Схема обращения потоков к матрице остается прежней. Отличие заключается в том, что каждый поток оперирует соответствующим элементом вектора  $d$ . После того как матрица  $B$  сформирована, вычисление  $i$ -го элемента вектора результата сводится к суммированию элементов  $i$ -го столбца матрицы с прибавлением к вычисленной сумме  $i$ -го элемента вектора  $y$ , предварительно умноженного на скаляр  $\beta$ .



## 5. Оценка точности

Полагая  $M = N$ , получим оценки точности для представленной реализации GEMV. Пусть выполняется операция  $y \leftarrow \alpha Ax + \beta y$ . Обозначим  $y^* = (y_1^*, y_2^*, \dots, y_N^*)$  — точный результат и  $\hat{y} = (\hat{y}_1, \hat{y}_2, \dots, \hat{y}_N)$  — результат применения операции GEMV. Будем оценивать границы абсолютной и относительной ошибки,  $\|y^* - \hat{y}\|$  и  $\|y^* - \hat{y}\|/\|y^*\|$ , соответственно (где  $\|\cdot\|$  —  $\ell_1$ -норма), используя стандартную модель арифметики с плавающей точкой [28, с. 40].

Для чисел вида (5) стандартной модели (в ограниченной форме) соответствует соотношение (6). Для упрощения промежуточных выражений будем использовать нотацию  $\theta_n$ , которая определяется следующим образом [28, с. 63]:

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

Здесь  $\delta_i$  — относительная ошибка, вносимая  $i$ -й операцией с плавающей точкой из последовательности операций, входящих в исследуемое выражение, причем  $|\delta_i| \leq \mathbf{u}$ . Целью нашего анализа является получение границы итоговой ошибки GEMV и нет необходимости различать вклад отдельных арифметических операций. Поэтому примем далее  $\delta_i \equiv \delta$ ,  $|\delta| \leq \mathbf{u}$ . В соответствии с (6) имеем  $\mathbf{u} < 4/\sqrt{M}$ .

Вначале вычисляется вектор  $d = \alpha x$ , элементы которого в соответствии со стандартной моделью выражаются следующим образом:

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

С учетом используемых обозначений, вычисленные элементы матрицы  $B$  могут быть записаны в следующей форме:

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

Далее вычисляется сумма элементов каждой строки матрицы  $B$ . Для  $i$ -й строки обозначим эту сумму  $s_i$ . Хорошо известно, что точность суммирования зависит от выбранного алгоритма. Как отмечено выше,  $s_i$  вычисляется одним блоком потоков. И хотя используемый Алгоритм 6 из [11] реализует попарное суммирование, максимальный размер блока (количество параллельных потоков, выполняющих суммирование) зависит от величины набора модулей СОК, т.е. от разрядности вычислений.

Когда размер блока равен единице,  $s_i$  вычисляется последовательно, что приводит к классическому рекурсивному алгоритму:

```

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

```

Для этого алгоритма вычисленное значение  $s_i$  может быть представлено в следующем виде [29]:

$$(12) \quad s_i = (b_{i1} + b_{i2})(1 + \theta_{N-1}) + b_{i3}(1 + \theta_{N-2}) + b_{i4}(1 + \theta_{N-3}) + \dots + b_{iN}(1 + \theta_1),$$

или то же самое:

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

Сложение  $s_i$  и  $\beta y_i(1 + \theta_1)$  дает

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

и после подстановки получаем финальное выражение для  $i$ -го элемента вычисленного вектора:

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

Для получения верхней границы ошибки следует исходить из того, что отдельные компоненты  $\theta_i$  взаимно не компенсируются. Будем считать все  $\theta_i > 0$ . Тогда приведенное выше выражение примет вид

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

Для дальнейшего упрощения перейдем к записи  $\gamma_n$  [28, с. 63], которая определяется следующим образом:

$$(14) \quad \gamma_n := \frac{n\mathbf{u}}{1 - n\mathbf{u}} \quad \text{при } n\mathbf{u} < 1.$$

Одним из свойств этой записи является  $\gamma_n \leq \gamma_{n+1}$  (см. [28, с. 67]). Учитывая это свойство, представленное выше выражение для  $\hat{y}_i$  можно переписать в следующем виде:

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

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

Поскольку  $i$ -й элемент точного вектора-результата  $y^*$  выражается в форме  $y_i^* = \beta y_i + \sum_{j=1}^N \alpha a_{ij} x_j$ , имеем следующую верхнюю границу абсолютной ошибки:

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

и верхнюю границу относительной ошибки:

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

где  $\mathbf{u} < 4/\sqrt{\mathcal{M}}$  и  $\mathcal{M}$  — произведение модулей СОК.

То же самое в более лаконичной форме:

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

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

При  $M = N$  полученные оценки справедливы также и для операции с транспонированной матрицей,  $y \leftarrow \alpha A^T x + \beta y$ .

## 6. Оценка производительности

В данном разделе представлены выполненные эксперименты и полученные результаты. Все эксперименты выполнялись на видеокарте NVIDIA GeForce GTX 1080 GPU (8 GB GDDR5X, 2560 CUDA cores, Compute Capability 6.1). Хост имел следующую конфигурацию: Intel Core i5 7500 (3.40 GHz) / 16 GB DDR4 RAM / Ubuntu 18.04.4 LTS / CUDA 10.2 / NVIDIA Driver 440.33.01. Исходный код компилировался с использованием компилятора nvcc с опциями `-O3 -use_fast_math -Xcompiler=-O3,-fopenmp,-ffast-math`.

### 6.1. Эффективность отдельных CUDA ядер

В данном эксперименте с использованием средства профилирования NVIDIA Visual Profiler мы исследовали эффективность использования ресурсов GPU отдельными вычислительными ядрами (глобальными функциями), запускаемыми в рассматриваемой реализации GEMV. Эксперимент выполнялся с нетранспонированной матрицей размера  $M = N = LDA = 1000$  с 424-битной точностью вычислений. Для достижения указанной точности был использован набор из 32 модулей СОК, каждый 32 бита, с динамическим диапазоном  $\mathcal{M} = 850$  бит. Точность вычислений с числами вида (5) составляет  $\lfloor \log_2 \sqrt{\mathcal{M}} \rfloor - 1$  бит [11]. Необходимо отметить, что хотя множественные округления могут оказать существенное влияние на общую производительность нашей реализации, мы всегда можем снизить их количество, используя набор модулей СОК большего размера. В связи с этим исходные данные для эксперимента были такими, что множественные округления не требовались.

В таблице 3 представлен вклад отдельных CUDA ядер в общее время выполнения операции. При указанных параметрах эксперимента наиболее трудоемким является суммирование элементов матрицы  $B$  по строкам, на которое приходится 71.5% от общего времени вычислений. Также существенный вклад вносит вычисление цифр многоразрядных мантисс при формировании  $B$ .

ТАБЛИЦА 3. Вклад отдельных CUDA ядер в общее время выполнения GEMV многократной точности

Функция	Описание	%
mp_vec2scal_mul_esi	Умножение $x$ на $\alpha$ , $y$ на $\beta$ : вычисление знаков, экспонент, интервальных оценок	0.2
mp_vec2scal_mul_digits	Умножение $x$ на $\alpha$ , $y$ на $\beta$ : вычисление цифр многоразрядных мантисс	0.1
mp_mat2vec_right_scal_esi	Формирование матрицы $B$ : вычисление знаков, экспонент, интервальных оценок	5.6
mp_mat2vec_right_scal_digits	Формирование матрицы $B$ : вычисление цифр многоразрядных мантисс	20.8
mp_vector_round	Округление промежуточных результатов	1.7
mp_matrix_row_sum	Построчная редукция матрицы $B$ и сложение с вектором $\beta y$	71.5

В таблице 4 представлены значения фактической загрузки доступных ресурсов GPU (occupancy), эффективной пропускной способности глобальной памяти (bandwidth) и ветвления логики исполнения (divergence) для двух наиболее трудоемких CUDA ядер. Разбиение операций многократной точности на части (`mp_mat2vec_right_scal_digits`) приводит к существенно более эффективным реализациям по сравнению с традиционным подходом, где каждая операция многократной точности выполняется одним потоком GPU (`mp_matrix_row_sum`).

ТАБЛИЦА 4. Показатели эффективности наиболее затратных вычислительных ядер

Ядро	Occupancy, %	Divergence, %	Bandwidth, ГБ/с
<code>mp_mat2vec_right_scal_digits</code>	95.5	0	163.3
<code>mp_matrix_row_sum</code>	26.8	66.8	49.9

Отметим, что NVIDIA 1080 GPU использует память GDDR5X с рабочей частотой 1251 МГц и шириной шины 256 бит, следовательно теоретическая пиковая пропускная способность памяти составляет

$$(20) \quad 1251 \times 10^6 \times (256/8) \times 8/10^9 = 320.3 \text{ ГБ/с.}$$

Для ядра `mp_mat2vec_right_scal_digits` эффективная пропускная способность составляет 51% от пиковой в связи с тем, что вычисление каждой цифры мантиссы сопровождается нахождением остатка от деления по соответствующему модулю (*mod*), но, как известно, операция *mod* является одной из наиболее трудоемких арифметических операций. Поэтому дальнейшее повышение быстродействия вычислений с цифрами многоразрядных мантис в СОК возможно за счет ускорения операции *mod*. Этого можно достичь, например, используя редукцию Барретта. Для ядра `mp_matrix_row_sum` фактическая пропускная способность составляет 16% от пиковой, так как все промежуточные результаты хранятся в разделяемой памяти, а в глобальную память записывается лишь финальный результат работы каждого блока потоков. Однако разделяемая память в данном случае является ограничивающим фактором для полного использования доступных ресурсов GPU.

## 6.2. Сравнение с другими реализациями

В ходе экспериментов быстродействие представленной реализации GEMV сравнивалось с реализациями, основанными на CUDA библиотеках многократной точности GARPPEC, CAMPARY и CUMP, которые рассматриваются в разделе 1. Все эти библиотеки основаны на позиционной системе счисления. Для того, чтобы оценить практический эффект от использования структуры `mp_array_t` и разбиения операций многократной точности на несколько CUDA ядер, мы реализовали классический алгоритм GEMV, в котором каждая арифметическая операция с многоразрядными числами вида (5) выполняется одним потоком GPU, а для хранения массива многоразрядных чисел используется массив структур `mp_float_t`. В дальнейшем эта версия GEMV называется “базовая реализация”.

Эксперименты выполнялись при фиксированном размере матрицы  $M = N = LDA = 1000$ . Рассматривались различные уровни арифметической точности, от 106 до 1696 бит. Исходные наборы данных были представлены псевдослучайными многоразрядными числами с плавающей точкой, равномерно распределенными в интервале  $[-1, 1]$ . Чтобы уменьшить влияние шума, оцениваемые реализации GEMV повторялись в цикле несколько раз, измерялось общее время выполнения всех итераций в миллисекундах, после чего рассчитывалось среднее время выполнения одной итерации.

Таблица 5. Время выполнения (в миллисекундах) GEMV многократной точности на GeForce GTX 1080.

	Точность в битах				
	106	212	424	848	1696
Предложенная реализация	3.1	5.6	8.5	12.9	24.6
Предложенная реализация (трансп.)	2.9	4.6	7.1	11.2	19.4
Базовая реализация	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
GARPPEC	26.5	37.4	70.2	206.0	689.1

Результаты экспериментов, представленные в таблице 5, показывают, что во многих случаях быстродействие предложенной реализации существенно выше, чем у аналогов, а также всегда выше чем у базовой

реализации. С ростом точности время предложенной реализации возрастает линейно. Операция с транспонированной матрицей оказалась незначительно быстрее операции с нетранспонированной матрицей, что обусловлено лучшим паттерном чтения матрицы  $B$  при ее редукции по столбцам.

При точности вычислений 106 бит (четверная точность) библиотека SAMPARY оказывается быстрее нашей реализации, однако при увеличении точности время SAMPARY существенно возрастает, что соответствует приведенным в разделе 1 оценкам сложности арифметических операций для данной библиотеки.

## 7. Выводы

В этой статье мы представили параллельную реализацию операции GEMV многократной точности для вычислительных систем с CUDA-совместимыми графическими процессорами. В основе нашей реализации лежит двухуровневая схема распараллеливания вычислений: на верхнем уровне осуществляется параллельная обработка элементов вектора/матрицы, а на нижнем уровне параллельно обрабатываются отдельные цифры многоразрядных мантисс элементов. Последнее стало возможным за счет перехода от традиционной двоичной арифметики к системе остаточных классов (СОК).









Однако простое использование СОК на GPU не дает значительного эффекта, так как операции с плавающей точкой многократной точности содержат последовательные и параллельные секции, что приводит к ветвлению путей выполнения. Для преодоления этого недостатка в предложенной реализации GEMV использована оригинальная схема вычислений, основанная на разбиении арифметических операций многократной точности на составные части, каждая из которых выполняется отдельным CUDA ядром. Такая схема увеличивает количество обращений к глобальной памяти устройства, так как все промежуточные результаты вычислений должны храниться в глобальной памяти. Однако это компенсируется тем, что ветвления внутри вычислительных ядер исключаются и обеспечивается более полная загрузка GPU. Эффективность такой схемы подтверждается экспериментами.

Разработанная реализация GEMV входит в состав новой GPU-библиотеки многократной точности MPRES-BLAS, которая доступна

на GitHub. Отметим, что функциональность MPRES-BLAS не ограничена базовыми операциями линейной алгебры. Фактически MPRES-BLAS является также арифметической библиотекой общего назначения, поскольку предоставляет CPU и CUDA реализации основных арифметических операций многократной точности. Кроме этого, в MPRES-BLAS реализованы оптимизированные алгоритмы для вычислений в СОК, такие как сравнение по величине и масштабирование степенью двойки. Также библиотека поддерживает арифметику расширенного диапазона (extended-range) для CPU и GPU, которая позволяет исключить переполнение и потерю значимости в процессе вычислений с разномасштабными величинами.










Направлениями дальнейших исследований является автоматический подбор оптимальной конфигурации вычислительных ядер в зависимости от точности вычислений и размера задачи. Также мы планируем адаптировать использованные подходы для реализации разреженного матрично-векторного умножения (SpMV) многократной точности на GPU.

## Список литературы

- [1] M. Courbariaux, Y. Bengio, J. David. *Training deep neural networks with low precision multiplications*, 2014. arXiv  1412.7024  <sup>33</sup>
- [2] D. H. Bailey, J. M. Borwein. “High-precision arithmetic in mathematical physics”, *Mathematics*, **3**:2 (2015), pp. 337–367.  <sup>34</sup>
- [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.  <sup>34</sup>
- [4] Y. Feng, J. Chen, W. Wu. “The PSLQ algorithm for empirical data”, *Math. Comp.*, **88**:317 (2019), pp. 1479–1501.  <sup>34</sup>
- [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.  <sup>34</sup>
- [6] M. Kyung, E. Sacks, V. Milenkovic. “Robust polyhedral Minkowski sums with GPU implementation”, *Comput. Aided Des.*, **67–68** (2015), pp. 48–57.  <sup>34</sup>
- [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.  <sup>34</sup>



- [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](#) ↑<sub>34</sub>
- [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](#) ↑<sub>34</sub>
- [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](#) ↑<sub>35,36</sub>
- [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](#) ↑<sub>35,36,38,40,43,44,46,47,49,52</sub>
- [12] A. Omondi, B. Premkumar. *Residue number systems: theory and implementation*, Imperial College Press, London, UK, 2007. ↑<sub>35,39</sub>
- [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. ↑<sub>35</sub>
- [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](#) ↑<sub>36,40</sub>
- [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](#) ↑<sub>36</sub>
- [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](#) ↑<sub>36</sub>
- [17] *Quadruple precision BLAS routines for GPU: QPBLAS-GPU ver.1.0. User’s manual*, 2013 (accessed 19 May 2019), 58 pp. [URL](#) ↑<sub>36,38</sub>
- [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](#) ↑<sub>36,38</sub>
- [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](#) ↑<sub>36,37,38</sub>

- [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.  [↑<sub>36</sub>](#)
- [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. [↑<sub>36</sub>](#)
- [22] J. R. Shewchuk. “Adaptive precision floating-point arithmetic and fast robust geometric predicates”, *Discrete & Computational Geometry*, **18**:3 (1997), pp. 305–363.  [↑<sub>36</sub>](#)
- [23] T. Ogita, S. M. Rump, S. Oishi. “Accurate sum and dot product”, *SIAM J. Sci. Comput.*, **26**:6 (2005), pp. 1955–1988.  [↑<sub>37</sub>](#)
- [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.  [↑<sub>37, 38</sub>](#)
- [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.  [↑<sub>37, 38</sub>](#)
- [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.  [↑<sub>38</sub>](#)
- [27] K. Isupov. “Using floating-point intervals for non-modular computations in residue number system”, *IEEE Access*, **8** (2020), pp. 58603–58619.  [↑<sub>39, 40, 47</sub>](#)
- [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.  [↑<sub>49, 50, 51</sub>](#)
- [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.  [↑<sub>50</sub>](#)


Поступила в редакцию 29.04.2020  
 Переработана 24.07.2020  
 Опубликована 20.08.2020


Рекомендовал к публикации

*к.ф.-м.н. С. А. Романенко*

Пример ссылки на эту публикацию:

К. С. Исупов, В. С. Князьков. «Матрично-векторное умножение многократной точности на графическом процессоре». *Программные системы: теория и приложения*, 2020, **11:3(46)**, с. 33–59.

 10.25209/2079-3316-2020-11-3-33-59

 [http://psta.psiras.ru/read/psta2020\\_3\\_33-59.pdf](http://psta.psiras.ru/read/psta2020_3_33-59.pdf)


Эта же статья по-английски:  10.25209/2079-3316-2020-11-3-61-84

Об авторах:

### Константин Сергеевич Исупов



Кандидат технических наук, доцент кафедры электронных вычислительных машин Вятского государственного университета. Область научных интересов: высокоточные вычисления, система остаточных классов, компьютерная арифметика, параллельные алгоритмы, программирование для графических процессоров, GPGPU.


 0000-0003-0239-0404

e-mail: [ks\\_isupov@vyatsu.ru](mailto:ks_isupov@vyatsu.ru)

### Владимир Сергеевич Князьков




Доктор технических наук, профессор, главный научный сотрудник научно-исследовательского института фундаментальных и прикладных исследований, Пензенский государственный университет. Область научных интересов: параллельные вычислительные архитектуры, высокопроизводительные вычисления, реконфигурируемые вычислительные системы.

 0000-0003-3820-6541

e-mail: [kniazkov@pnzgu.ru](mailto:kniazkov@pnzgu.ru)

Sample citation of this publication:

Konstantin S. Isupov, Vladimir S. Knyazkov. “Multiple-precision matrix-vector multiplication on graphics processing units”. *Program Systems: Theory and Applications*, 2020, **11:3(46)**, pp. 33–59. (In Russian).  10.25209/2079-3316-2020-11-3-33-59

 [http://psta.psiras.ru/read/psta2020\\_3\\_33-59.pdf](http://psta.psiras.ru/read/psta2020_3_33-59.pdf)

The same article in English:  10.25209/2079-3316-2020-11-3-61-84