Эффективное масштабирование в системе остаточных классов с использованием интервальных оценок

К. С. Исупов*, В. С. Князьков, А. С. Куваев
Вятский государственный университет, Киров, Россия
*Контактный e-mail: ks_isupov@vyatsu.ru

Рассмотрены два алгоритма масштабирования чисел в системе остаточных классов (СОК). В этих алгоритмах использована новая процедура определения остатка от деления масштабируемого числа X на коэффициент K, основанная на интервальной оценке относительной величины числа в СОК. Первый алгоритм эффективен для коэффициентов в пределах разрядной сетки компьютера. Второй алгоритм является обобщением первого для быстрого масштабирования степенью двойки в СОК. Он позволяет без реинициализации системы в качестве коэффициента масштабирования использовать любое число $K = 2^D$ в диапазоне от 2 до $M - 1$, где M — произведение всех модулей. В процессе вычислений требуются только стандартные операции с плавающей точкой и целочисленные модулярные операции. Оба алгоритма позволяют эффективно использовать параллелизм СОК.

Представлены CPU- и CUDA-реализации разработанных алгоритмов и рассмотрены их особенности. Эксперименты на CPU Intel Xeon X5650 и GPU NVIDIA Tesla M2050 показали, что предложенные алгоритмы обладают высоким быстродействием по сравнению с известными алгоритмами масштабирования, основанными на преобразовании в двоичную систему счисления и контроле четности.

Ключевые слова: система остаточных классов, масштабирование, параллельные алгоритмы, программирование на CUDA.

Введение

Система остаточных классов (СОК) [1–3] обеспечивает эффективное сложение, вычитание и умножение многоразрядных чисел с распараллеливанием обработки отдельных цифр, благодаря чему является востребованной в современных высокопроизводительных приложениях, таких как криптография [4, 5], цифровая обработка сигналов [6], беспроводные сенсорные сети [7] и цифровая обработка изображений [8]. В основном алгоритмы вычислений в СОК реализуются аппаратно (FPGA, ASIC). В то же время современные параллельные архитектуры общего назначения позволяют эффективно использовать СОК в программном обеспечении, например для реализации криптосистем на эллиптических кривых [5] и в арифметике многократной точности [9].

Однако выполнение некоторых базовых операций в СОК, таких как масштабирование, занимает значительно (до нескольких порядков) больше времени, чем выполнение сложения, вычитания или умножения. В данном случае масштабирование числа X означает сокращение его разрядности путем деления на константу K:

$$Y = \left\lfloor \frac{X}{K} \right\rfloor .$$

В двоичной арифметике K обычно является степенью двойки, поэтому масштабирование реализуется простым отбрасыванием младших разрядов двоичного представления. Однако в СОК нет эквивалентной операции, вследствие чего возрастающая разрядность результата, накапливаемого при выполнении серии умножений или сложений, приводит к переполнению динамического диапазона [10]. Разработка алгоритмов масштабирования, обладающих высоким быстродействием при приемлемых затратах памяти, востребована во многих приложениях СОК. Например, масштабирование степенью двойки широко используется для редукции разрядности результата в системах цифровой обработки сигналов [11].

В данной статье рассматриваются два алгоритма масштабирования в СОК, которые хорошо подходят для программной реализации. Первый алгоритм, являющийся базовым, основан на теореме о делении с нулевым остатком. Для расширения набора модулей с целью определения остатка от деления масштабируемого числа на коэффициент масштабирования используются интервальные аппроксимации относительных величин [12]. Преобразование из СОК во взвешенную систему счисления или использование избыточных модулей не требуется. Второй алгоритм предназначен для масштабирования произвольной степенью двойки. Он является обобщением базового алгоритма и не накладывает ограничений на величину коэффициента масштабирования. Новые алгоритмы реализованы для центральных процессоров (CPU) и графических процессоров видеокарты (GPU) с использованием NVIDIA CUDA.

1. Система остаточных классов

Система остаточных классов определяется набором из n взаимно простых целых чисел $\{m_1, m_2, \ldots, m_n\}$, называемых модулями. Целое число X представляется в СОК набором остатков:

$$X = \langle x_1, x_2, \ldots, x_n \rangle,$$

где $x_i = |X|_{m_i}$ — наименьший неотрицательный остаток от деления X на m_i. В СОК обеспечивается уникальное представление всех целых чисел в интервале $[0, M-1]$, где $M = \prod_{i=1}^{n} m_i$ — динамический диапазон СОК.

Введем обозначения: $M_i = M/m_i$ и $\alpha_i = \left|M_i^{-1}\right|_{m_i}$ — мультипликативная инверсия M_i по модулю m_i. Китайская теорема об остатках (КТО) устанавливает, что

$$|X|_M = \left| \sum_{i=1}^{n} M_i \left| x_i \alpha_i \right|_{m_i} \right|_M . \tag{1}$$

Это выражение задает прямой способ преобразования числа из СОК в двоичную систему счисления, также известный как метод ортогональных базисов.

Если X, Y и Z представлены в СОК наборами остатков $\langle x_1, x_2, \ldots, x_n \rangle$, $\langle y_1, y_2, \ldots, y_n \rangle$ и $\langle z_1, z_2, \ldots, z_n \rangle$ соответственно, то сложение, беззнаковое вычитание и умножение могут выполняться одновременно с n остатками в n параллельных потоках:

$$Z = \begin{cases} z_1 = |x_1 \; op \; y_1|_{m_1}, \\ z_2 = |x_2 \; op \; y_2|_{m_2}, \\ \ldots \\ z_n = |x_n \; op \; y_n|_{m_n}. \end{cases} \qquad op \in \{+, -, \cdot\}$$

Такая высокоскоростная параллельная обработка делает СОК привлекательной системой счисления [10]. Однако в общем случае для деления, масштабирования, сравнения по величине, определения знака, контроля переполнения и ряда других операций в СОК не существует параллельной формы и это является существенным ограничением.

Следующая теорема определяет условия, при которых деление в СОК выполняется простым умножением делимого на мультипликативную инверсию делителя.

Теорема 1 (Division Remainder Zero [2, 13]). *Если* $X = \langle x_1, x_2, \ldots, x_n \rangle$ *и* $|K^{-1}|_{m_i}$ — *мультипликативная инверсия* K *по модулю* m_i, *то*

$$|X/K|_M = \left|X \cdot |K^{-1}|_M\right|_M = \left\langle \left|x_1|K^{-1}|_{m_1}\right|_{m_1}, \left|x_2|K^{-1}|_{m_2}\right|_{m_2}, \ldots, \left|x_n|K^{-1}|_{m_n}\right|_{m_n} \right\rangle$$

тогда и только тогда, когда X *делится без остатка на* K *и* $\gcd(K, m_i) = 1$ *для всех* $i \in \{1, 2, \ldots, n\}$.

Согласно теореме, если K — коэффициент масштабирования, взаимно простой с M, и $Y = \langle y_1, y_2, \ldots, y_n \rangle$ — результат масштабирования X коэффициентом K, то

$$Y = \left\lfloor \frac{X}{K} \right\rfloor = \left(X - |X|_K\right) \cdot \left|K^{-1}\right|_M .$$

Если $|K^{-1}|_{m_i}$ — мультипликативная инверсия K по модулю m_i, то i-й остаток частного будет определяться в соответствии с выражением

$$y_i = \left| \left|x_i - |X|_K\right|_{m_i} \cdot \left|K^{-1}\right|_{m_i} \right|_{m_i} . \tag{2}$$

Таким образом, масштабирование сводится к нахождению остатка $|X|_K$ с последующими модулярными вычитанием и умножением [14].

2. Существующие алгоритмы масштабирования

Классическим методом является преобразование из СОК в двоичную систему, масштабирование в двоичной системе и преобразование результата обратно в СОК. Однако при больших динамических диапазонах этот метод считается затратным с точки зрения времени, так как требует обработки больших чисел и редукции по большому модулю M. Существует целый ряд алгоритмов масштабирования, не требующих преобразования из СОК в двоичную систему [13–19]. Их исчерпывающий обзор приведен в [20].

Во многих алгоритмах в качестве K используется произведение первых S модулей СОК, $K = \prod_{i=1}^{S} m_i$ [14, 16–19]. Это упрощает нахождение $|X|_K$ для старших остатков y_i, $S{+}1 \le i \le n$. Для нахождения младших остатков y_i, $1 \le i \le S$, используется один из алгоритмов расширения набора модулей (base extension). В [17] для этой цели применяется преобразование из СОК в систему со смешанными основаниями (Mixed-Radix System, MRS). Алгоритм Гарсия — Льорис [14] полностью основан на использовании подстановочных таблиц и характеризуется высокой скоростью, однако при большом числе модулей затраты памяти существенно возрастают. Метод масштабирования Шеной — Кумаресан [18] основан на технике расширения набора модулей СОК, использующей один избыточный модуль [21]. Алгоритм Барси — Пинотти [19] позволяет избавиться от избыточности за счет использования дополнительных подстановочных таблиц.

В алгоритме [16] используется преобразование из СОК в MRS. Далее масштабирование выполняется в MRS и результат преобразуется обратно в СОК. Преобразование из СОК в MRS требует выполнения $O(n^2)$ операций [2], поэтому является медленным при использовании наборов модулей большого размера.

В 2003 г. Уве Мейер-Безе и Танос Стоураитис предложили итерационный алгоритм масштабирования степенью двойки [13]. На каждой итерации для вычисления $|X|_2$ определяется четность X в СОК с использованием подстановочных таблиц. Масштабирование коэффициентом $K = 2^r$ выполняется за r итераций. Алгоритм также позволяет масштабировать числа со знаком. Эта техника масштабирования применена в работе [15] для сокращения динамического диапазона при реализации полифазного фильтра в СОК, где определение четности X выполняется с использованием метода расширения набора модулей [21] и дробной версии КТО [22].

Существуют методы масштабирования для наборов модулей специального вида, например $\{2^n-1, 2^n, 2^n+1\}$, $\{2^n-1, 2^{n+x}, 2^n+1\}$ или $\{2^n-1, 2^{n+x}, 2^n+1, m_4\}$ [11, 23, 24]. На основе этих методов строятся эффективные вычислительные архитектуры, обладающие высокой производительностью и низкими аппаратурными затратами. Однако, когда требуются большие динамические диапазоны (что типично для высокоточной арифметики), применение таких наборов модулей становится невозможным.

Таким образом, многие известные алгоритмы сводят задачу масштабирования к поиску остатка $|X|_K$ посредством того или иного метода расширения набора модулей. Мы используем для этой цели интервальную оценку относительной величины числа в СОК.

3. Масштабирование с использованием интервальных оценок

Уравнение (1) может быть записано в виде

$$X = \left(\sum_{i=1}^{n} M_i \left| x_i \alpha_i \right|_{m_i} \right) - kM, \tag{3}$$

где k — целое неотрицательное число.

Легко видеть, что $k < n$. Если k известно, то остаток $|X|_K$ вычисляется просто:

$$|X|_K = \left| \left(\sum_{i=1}^{n} \left| M_i \left| x_i \alpha_i \right|_{m_i} \right|_K \right) - |kM|_K \right|_K . \tag{4}$$

Для вычисления k может быть использован избыточный модуль m_r [21]:

$$k = \left| \left| M^{-1} \right|_{m_r} \cdot \left(\left| \sum_{i=1}^{n} M_i \left| x_i \alpha_i \right|_{m_i} \right|_{m_r} - x_r \right) \right|_{m_r}$$

при условии, что $m_r \geq n$. Однако такой подход приводит к накладным расходам, связанным с обработкой остатка $x_r = |X|_{m_r}$ в течение всех расчетов в СОК, чтобы он был известен к моменту вычисления k. Вместо этого мы будем использовать дробную версию КТО [22, 25]. Разделив (3) на произведение модулей M, получим выражение для относительной величины (дробного представления) числа X:

$$\frac{X}{M} = \left(\sum_{i=1}^{n} \frac{\left| x_i \alpha_i \right|_{m_i}}{m_i} \right) - k. \tag{5}$$

С другой стороны, как следует из (1),

$$\frac{X}{M} = \sum_{i=1}^{n} \frac{|x_i \alpha_i|_{m_i}}{m_i} - \left\lfloor \sum_{i=1}^{n} \frac{|x_i \alpha_i|_{m_i}}{m_i} \right\rfloor, \tag{6}$$

где $\lfloor\ \rfloor$ означает округление до ближайшего целого числа, не превышающего данной величины. Подставим (6) в (5) и выразим k:

$$k = \left\lfloor \sum_{i=1}^{n} \frac{|x_i \alpha_i|_{m_i}}{m_i} \right\rfloor. \tag{7}$$

Сумма в (7) должна быть вычислена с высокой точностью, иначе k, являющееся целой частью этой суммы, может отличаться от истинного значения в большую или в меньшую сторону. Точное вычисление (7), при котором гарантируется корректность результата, требует работы с $\lceil \log_2 Mn \rceil$-битными дробными числами [26]. При большом M это влечет существенные накладные расходы, критичные для программной реализации.

В [12] предложен метод интервальной оценки относительной величины, который заключается в вычислении для числа X, представленного в СОК с динамическим диапазоном M, такого интервала $I(X/M) = [\underline{X/M}, \overline{X/M}]$, что $\underline{X/M} \le X/M \le \overline{X/M}$. Здесь $\underline{X/M}$ и $\overline{X/M}$ — обычные числа с плавающей точкой машинной точности, возможно с расширенной экспонентой (при больших M). Для их расчета предложен алгоритм, основанный на поэтапном уточнении, который использует только малоразрядные целочисленные операции и операции с плавающей точкой с направленными округлениями стандарта IEEE-754. Интервал $I(X/M)$ определяет границы изменения числа X, масштабированного относительно M. Это позволяет эффективно выполнять такие трудоемкие для СОК операции, как сравнение, определение знака числа и контроль переполнения диапазона. Данные операции сводятся к анализу границ интервалов, представленных в двоичной системе счисления. Использование интервального подхода для оценки X/M вместо вычисления одного приближенного числа решает проблему обеспечения достоверности результатов вычислений. Например, если в системе с модулями $\{3, 5, 7\}$ заданы числа $X = \langle 2, 3, 1 \rangle$ и $Y = \langle 1, 1, 2 \rangle$, для которых вычислены $I(X/M) = [0.07, 0.09]$ и $I(Y/M) = [0.14, 0.16]$, то гарантируется, что $X < Y$, при этом преобразование X и Y в двоичную систему не требуется.

Мы используем идеи описанного интервального метода для эффективного вычисления k в задаче масштабирования. Вместо повышения разрядности расчетов при вычислении суммы $S = \sum\limits_{i=1}^{n} \dfrac{|x_i \alpha_i|_{m_i}}{m_i}$ в (7) будем выполнять стандартные операции с плавающей точкой с направленными округлениями “вниз” и “вверх”. В результате получим интервальную оценку суммы S, определяемую границами S_{low} и S_{upp}:

$$S_{\text{low}} = \bigtriangledown \sum_{i=1}^{n} \Big(|x_i \alpha_i|_{m_i} \mathbin{\bigtriangledown} m_i \Big), \tag{8}$$

$$S_{\text{upp}} = \bigtriangleup \sum_{i=1}^{n} \Big(|x_i \alpha_i|_{m_i} \mathbin{\bigtriangleup} m_i \Big). \tag{9}$$

Символы $\bigtriangledown$ и $\overline{\bigtriangledown}$ в (8) означают суммирование и деление с плавающей точкой, выполняемые с округлением "вниз"; аналогично $\bigtriangleup$ и $\overline{\bigtriangleup}$ в (9) означают операции с округлением "вверх". Так, если a и b — числа с плавающей точкой, то $a \mathbin{\overline{\bigtriangledown}} b$ — наибольшее число с плавающей точкой, меньшее либо равное a/b; в свою очередь, $a \mathbin{\overline{\bigtriangleup}} b$ — наименьшее число с плавающей точкой, большее либо равное a/b. Направленные округления IEEE-754 поддерживаются большинством современных вычислительных архитектур.

Для вычисления k дробные части S_{low} и S_{upp} отбрасываются:

$$k_{\text{low}} = \lfloor S_{\text{low}} \rfloor, \quad k_{\text{upp}} = \lfloor S_{\text{upp}} \rfloor.$$

При этом возможны два случая (рис. 1):

- $k_{\text{low}} = k_{\text{upp}}$. Это свидетельствует о том, что S_{low} и S_{upp} вычислены с достаточной точностью. Тогда $k = k_{\text{low}} = k_{\text{upp}}$.
- $k_{\text{low}} \neq k_{\text{upp}}$. Это свидетельствует о том, что точность вычисления S_{low} и S_{upp} недостаточна для однозначного определения k. В этом случае вычисляется ранг $r(X)$ — характеристика, показывающая, сколько раз диапазон системы превышен при переходе от представления X в СОК к его позиционному представлению [1]. Ранг связан с искомым k следующим соотношением: $k = r(X) - \sum_{i=1}^{n} \lfloor x_i \alpha_i / m_i \rfloor$.

В первом случае для последовательного вычисления k потребуется $O(n)$ элементарных операций (целочисленных операций по модулям $m_1, m_2, \ldots, m_n$ и операций с плавающей точкой машинной точности). В свою очередь, при распараллеливании вычислений по модулям СОК потребуется только $O(\log n)$ элементарных операций.

Во втором случае для вычисления ранга может быть использован алгоритм Rank из [12]. Он позволяет вычислить $r(X)$ за n итераций, где n — размер набора модулей СОК. Сложность каждой итерации линейна при полностью последовательных расчетах и постоянна при распараллеливании вычислений по модулям. На каждой итерации выполняются только целочисленные операции небольшой разрядности.

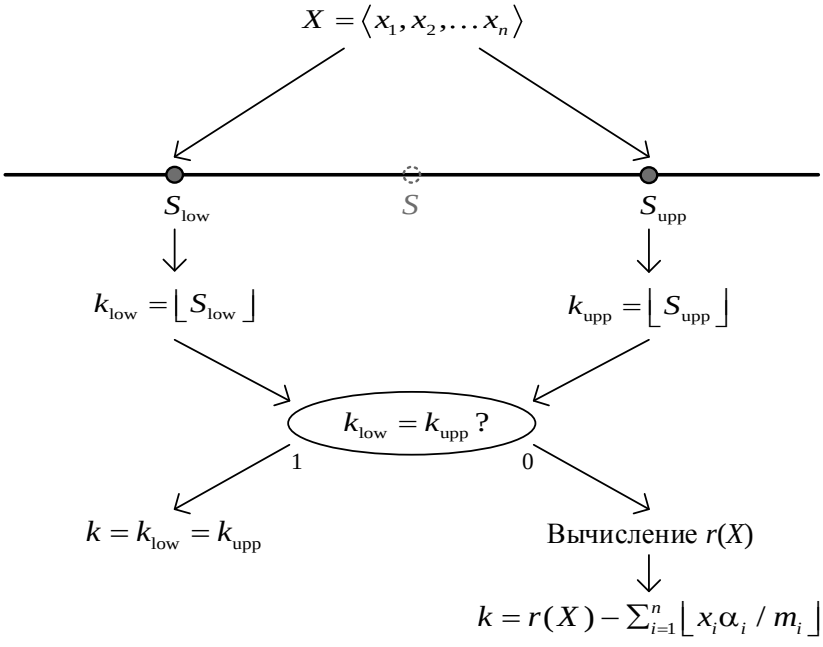

Рис. 1. Расчет k с использованием интервальных оценок

Второй случай возникает, только если масштабируемое X число лежит вблизи границ диапазона $[0, M-1]$. Для равномерного распределения вероятность этого очень мала [12].

Если k известно, то искомый остаток $|X|_K$ вычисляется просто. Так как коэффициент масштабирования K является постоянным, то уравнение (4) может быть записано в виде

$$|X|_K = \left| \left(\sum_{i=1}^{n} \left| |M_i|_K \, |x_i \alpha_i|_{m_i} \right|_K \right) - \left| k \, |M|_K \right|_K \right|_K . \tag{10}$$

В отличие от (4), в (10) не требуется обработка больших чисел M_i и M.

Алгоритм 1 масштабирует число X коэффициентом K, взаимно простым с M. В результате возвращается частное $Y = \lfloor X/K \rfloor$, представленное остатками $y_i = |Y|_{m_i}$. В процессе вычислений используются только стандартные операции с плавающей точкой и целочисленные операции по модулям $m_1, m_2, \ldots, m_n$ и K. Заранее вычисляются следующие константы: $\alpha_i = \left| M_i^{-1} \right|_{m_i}$, $|M_i|_K$ и $|K^{-1}|_{m_i}$ для $i = 1, 2, \ldots, n$, а также $|M|_K$.

Алгоритм 1 Масштабирование $X = \langle x_1, x_2, \ldots, x_n \rangle$ коэффициентом K

1: $c_i \leftarrow |x_i \alpha_i|_{m_i}$ **for all** i from 1 to n
2: $k_{\text{low}} \leftarrow \left\lfloor \bigtriangledown \sum_{i=1}^{n} \left(c_i \mathbin{\overline{\bigtriangledown}} m_i \right) \right\rfloor$
3: $k_{\text{upp}} \leftarrow \left\lfloor \bigtriangleup \sum_{i=1}^{n} \left(c_i \mathbin{\overline{\bigtriangleup}} m_i \right) \right\rfloor$
4: **if** $k_{\text{low}} = k_{\text{upp}}$ **then**
5: $\quad k \leftarrow k_{\text{low}}$
6: **else**
7: $\quad r(X) \leftarrow \textsc{Rank}(X)$ ▷ Алгоритм из [12]
8: $\quad h \leftarrow \sum_{i=1}^{n} \left\lfloor x_i \alpha_i \,/\, m_i \right\rfloor$
9: $\quad k \leftarrow r(X) - h$
10: **end if**
11: $|X|_K \leftarrow \left| \left(\sum_{i=1}^{n} \left| |M_i|_K \, c_i \right|_K \right) - \left| k \, |M|_K \right|_K \right|_K$
12: $y_i \leftarrow \left| \left| x_i - |X|_K \right|_{m_i} \cdot \left| K^{-1} \right|_{m_i} \right|_{m_i}$ **for all** i from 1 to n
13: **return** $\langle y_1, \ldots, y_n \rangle$

Количество операций в алгоритме 1 зависит от выполнения условия $k_{\text{low}} = k_{\text{upp}}$ на шаге 4. Если оно выполняется, то последовательная реализация алгоритма потребует выполнения $9n$ операций для масштабирования числа в n-модульной СОК. Параллельная реализация, в свою очередь, потребует только $3 \log_2 n$ операций. Если $k_{\text{low}} \neq k_{\text{upp}}$, то для определения k используется алгоритм вычисления ранга из [12]. В этом случае алгоритм 1 выполнит $n(2n+13)$ и $3(3n + \log_2 n)$ элементарных операций при последовательных и параллельных вычислениях соответственно.

Пример 1. Рассмотрим модули $m_1 = 7$, $m_2 = 9$, $m_3 = 11$, $m_4 = 13$. Пусть число $X = 5308 = \langle 2, 7, 6, 4 \rangle$ необходимо масштабировать коэффициентом $K = 23$. Для заданных модулей и коэффициента масштабирования заранее рассчитываются: $|M|_K = 16$,

$$\alpha_1 = 6, \qquad |M_1|_K = 22, \qquad \left|K^{-1}\right|_{m_i} = 4,$$
$$\alpha_2 = 5, \qquad |M_2|_K = 12, \qquad \left|K^{-1}\right|_{m_i} = 2,$$
$$\alpha_3 = 9, \qquad |M_3|_K = 14, \qquad \left|K^{-1}\right|_{m_i} = 1,$$
$$\alpha_4 = 10, \qquad |M_4|_K = 3, \qquad \left|K^{-1}\right|_{m_i} = 4.$$

Вычисляем $c_1 = |2 \cdot 6|_7 = 5$, $c_2 = |7 \cdot 5|_9 = 8$, $c_3 = |6 \cdot 9|_{11} = 10$, $c_4 = |4 \cdot 10|_{13} = 1$. В результате расчета k_{low} и k_{upp} с округлением до двух десятичных разрядов имеем

$$k_{\text{low}} = \left\lfloor \bigtriangledown\big(5 \bigtriangledown 7 + 8 \bigtriangledown 9 + 10 \bigtriangledown 11 + 1 \bigtriangledown 13\big)\right\rfloor = \left\lfloor \bigtriangledown\big(0.71 + 0.88 + 0.90 + 0.07\big)\right\rfloor = 2,$$

$$k_{\text{upp}} = \left\lfloor \bigtriangleup\big(5 \bigtriangleup 7 + 8 \bigtriangleup 9 + 10 \bigtriangleup 11 + 1 \bigtriangleup 13\big)\right\rfloor = \left\lfloor \bigtriangleup\big(0.72 + 0.89 + 0.91 + 0.08\big)\right\rfloor = 2,$$

следовательно, $k = 2$. Находим $|X|_K$:

$$|X|_K = \big||22 \cdot 5|_{23} + |12 \cdot 8|_{23} + |14 \cdot 10|_{23} + |3 \cdot 1|_{23} - |2 \cdot 16|_{23}\big|_{23} = 18.$$

Масштабированные остатки y_i вычисляются в соответствии с (2):

$$y_1 = \big||2 - 18|_7 \cdot 4\big|_7 = 6, \qquad y_2 = \big||7 - 18|_9 \cdot 2\big|_9 = 5,$$
$$y_3 = \big||6 - 18|_{11} \cdot 1\big|_{11} = 10, \qquad y_4 = \big||4 - 18|_{13} \cdot 4\big|_{13} = 9.$$

Таким образом, $Y = \langle 6, 5, 10, 9\rangle = 230 = \lfloor 5308/23 \rfloor$.

4. Масштабирование произвольной степенью двойки

В случае, когда разрядность коэффициента масштабирования K выходит за пределы машинного слова, операции редукции по модулю K, выполняемые на шаге 11 алгоритма 1, становятся неэффективными, так как требуют использования многоразрядной арифметики, что на порядок увеличивает временные затраты. Поэтому алгоритм 1 применим к использованию при сравнительно небольших коэффициентах масштабирования, не превышающих разрядности вычислительной системы (например, $K \le 2^{64}$ для 64-битной архитектуры). Далее рассматривается алгоритм, пригодный для масштабирования чисел в СОК степенями двойки произвольной величины.

Предполагаем, что все модули $\{m_1, m_2, \ldots, m_n\}$ нечетные. Обозначим символом T пороговую величину (целое неотрицательное число), при которой применение алгоритма 1 с коэффициентом 2^T эффективно. Пусть дано число X, определяемое остатками $\langle x_1, x_2, \ldots, x_n\rangle$, которое требуется масштабировать коэффициентом 2^D, получив результат $Y = \lfloor X/2^D \rfloor$, определяемый остатками $y_i = |Y|_{m_i}$. Мы не накладываем дополнительных ограничений на коэффициент масштабирования, т. е. предполагаем, что 2^D может изменяться в диапазоне от 2 до $M - 1$. Тогда, если $D \le T$, то для масштабирования используется алгоритм 1. Если $D > T$, то определяются $t = \lfloor D/T \rfloor$, $d = |D|_T$ и далее выполняются итерационные вычисления:

$$X_j = \lfloor X_{j-1}/2^T \rfloor, \quad j = 1, 2, \ldots, t. \tag{11}$$

В качестве начального значения принимается $X_0 = X$. В результате будет получено $X_t = \lfloor X/2^{T\lfloor D/T \rfloor} \rfloor$. Затем выполняется уточнение результата:

$$Y = \lfloor X/2^D \rfloor = \lfloor X_t/2^d \rfloor.$$

Первая итерация в (11) аналогична алгоритму 1. Во всех последующих расчетах применяется упрощенная процедура вычисления k:

$$k = \left\lfloor \triangle \sum_{i=1}^{n} \Big(|x_i \alpha_i|_{m_i} \triangleq m_i \Big) \right\rfloor. \tag{12}$$

Так как $X < M$ и $T \geq 1$, то после первой итерации (11) будет получено число $X_1 < M/2$, для которого $X_1/M < 0.5$. Поэтому k, вычисляемое в соответствии с (12), будет корректным при любом разумном соотношении точности и числа модулей. В частности, при вычислениях в арифметике IEEE-754 с точностью (разрядностью мантиссы) p бит уравнение (12) обеспечивает правильность результата при $n < \sqrt{2^{p-2}}$. Так, при одинарной точности ($p = 24$) допустимо использовать до 2047 модулей, а при удвоенной точности ($p = 53$) максимальный размер набора модулей составляет 47 453 132.

Алгоритм 2 выполняет масштабирование числа X коэффициентом 2^D. В результате возвращается частное $Y = \lfloor X/2^D \rfloor$, представленное остатками $y_i = |Y|_{m_i}$. Этот алгоритм требует вычисления следующих констант: $\alpha_i = \left| M_i^{-1} \right|_{m_i}$ для $i = 1, 2, \ldots, n$, $|M|_{2^j}$ для $j = 1, 2, \ldots, T$,

$$H = \begin{bmatrix} |M_1|_2 & |M_2|_2 & \ldots & |M_n|_2 \\ |M_1|_4 & |M_2|_4 & \ldots & |M_n|_4 \\ \vdots & \vdots & \ddots & \vdots \\ |M_1|_{2^T} & |M_2|_{2^T} & \ldots & |M_n|_{2^T} \end{bmatrix}, \quad P = \begin{bmatrix} |2^{-1}|_{m_1} & |2^{-1}|_{m_2} & \ldots & |2^{-1}|_{m_n} \\ |4^{-1}|_{m_1} & |4^{-1}|_{m_3} & \ldots & |4^{-1}|_{m_n} \\ \vdots & \vdots & \ddots & \vdots \\ \left|(2^T)^{-1}\right|_{m_1} & \left|(2^T)^{-1}\right|_{m_2} & \ldots & \left|(2^T)^{-1}\right|_{m_n} \end{bmatrix}.$$

Пороговая величина T задается так, чтобы операции по модулю 2^T выполнялись эффективно на данной вычислительной системе. При этом, если $D \leq T$, то сложность алгоритма 2 равна сложности алгоритма 1. Если $D \gg T$, то потребуется в среднем $7n\lceil D/T \rceil$ операций при последовательной реализации алгоритма и $2\log_2 n \lceil D/T \rceil$ операций при распараллеливании вычислений по модулям СОК. В худшем случае, когда использование интервальных оценок (8) и (9) не позволяет однозначно определить значение k, потребуется однократное (вне зависимости от числа итераций алгоритма 2) вычисление ранга. Это влечет дополнительно $n(2n + 7)$ и $9n$ целочисленных операций при последовательных и параллельных расчетах соответственно.

Пример 2. Пусть в СОК с модулями $\{7, 9, 11, 13\}$ задано число $X = 3413 = \langle 4, 2, 3, 7 \rangle$, которое требуется масштабировать коэффициентом $K = 2^8$. Примем $T = 3$. Предварительно вычисляются: $\alpha_1 = 6,\ \alpha_2 = 5,\ \alpha_3 = 9,\ \alpha_4 = 10,\ |M|_{2^1} = 1,\ |M|_{2^2} = 1,\ |M|_{2^3} = 1,$

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 3 & 1 & 3 & 1 \\ 7 & 1 & 3 & 5 \end{bmatrix}, \quad P = \begin{bmatrix} 4 & 5 & 6 & 7 \\ 2 & 7 & 3 & 10 \\ 1 & 8 & 7 & 5 \end{bmatrix}.$$

Алгоритм 2 Масштабирование $X = \langle x_1, x_2, \ldots, x_n \rangle$ коэффициентом 2^D

$t \leftarrow \lfloor D/T \rfloor$
if $t > 0$ **then**
  $c_i \leftarrow |x_i \alpha_i|_{m_i}$ **for all** i from 1 to n
  Вычислить k так же, как в алгоритме 1
  $\langle x_1, \ldots, x_n \rangle \leftarrow \textsc{Scale}(k, T, x_1 \ldots x_n, c_1 \ldots c_n)$
  $t \leftarrow t - 1$
end if
while $t > 0$ **do**
  $c_i \leftarrow |x_i \alpha_i|_{m_i}$ **for all** i from 1 to n
  $k \leftarrow \left\lfloor \triangle \sum_{i=1}^{n} \left(c_i \triangleq m_i\right) \right\rfloor$
  $\langle x_1, \ldots, x_n \rangle \leftarrow \textsc{Scale}(k, T, x_1 \ldots x_n, c_1 \ldots c_n)$
  $t \leftarrow t - 1$
end while
$y_i \leftarrow x_i$ **for all** i from 1 to n
$d \leftarrow |D|_T$
if $d > 0$ **then**
  $c_i \leftarrow |x_i \alpha_i|_{m_i}$ **for all** i from 1 to n
  if $d < D$ **then**
    $k \leftarrow \left\lfloor \triangle \sum_{i=1}^{n} \left(c_i \triangleq m_i\right) \right\rfloor$
  else
    Вычислить k так же, как в алгоритме 1
  end if
  $\langle y_1, \ldots, y_n \rangle \leftarrow \textsc{Scale}(k, d, x_1 \ldots x_n, c_1 \ldots c_n)$
end if
return $\langle y_1, \ldots, y_n \rangle$

function $\textsc{Scale}(k, b, x_1 \ldots x_n, c_1 \ldots c_n)$
  С использованием b-й строки из H вычислить:
  $|X|_{2^b} \leftarrow \left| \left(\sum_{i=1}^{n} \left| |M_i|_{2^b} c_i \right|_{2^b} \right) - \left| k \, |M|_{2^b} \right|_{2^b} \right|_{2^b}$
  С использованием b-й строки из P вычислить:
  $y_i \leftarrow \left| \left| x_i - |X|_{2^b} \right|_{m_i} \cdot \left| (2^b)^{-1} \right|_{m_i} \right|_{m_i}$ **for all** i from 1 to n
  return $\langle y_1, \ldots, y_n \rangle$
end function

Так как $t = \lfloor 8/3 \rfloor = 2$ и $d = |8|_3 = 2$, потребуются два шага масштабирования с коэффициентом 2^3 и один шаг с коэффициентом 2^2. На первом шаге $c_i = \{3, 1, 5, 5\}$, $k_{\text{low}} = k_{\text{upp}} = 1$, поэтому $k = 1$. Остаток от деления X на 2^3 вычисляется с использованием третьей строки матрицы H : $|X|_8 = \left| |7 \cdot 3|_8 + |1 \cdot 1|_8 + |3 \cdot 5|_8 + |5 \cdot 5|_8 \right|_8 = 5$. Масштабированные остатки вычисляются с использованием третьей строки из P:

$$x_1 = \left| |4 - 5|_7 \cdot 1 \right|_7 = 6, \qquad x_2 = \left| |2 - 5|_9 \cdot 8 \right|_9 = 3,$$
$$x_3 = \left| |3 - 5|_{11} \cdot 7 \right|_{11} = 8, \qquad x_4 = \left| |7 - 5|_{13} \cdot 5 \right|_{13} = 10.$$

Таким образом, $X_1 = \langle 6, 3, 8, 10\rangle$. На втором шаге $c_i = \{1, 6, 6, 9\}$ и в соответствии с (12) $k = 2$. Поэтому $|X_1|_8 = 2$ и масштабированные остатки

$$x_1 = \big||6-2|_7 \cdot 1\big|_7 = 4, \qquad x_2 = \big||3-2|_9 \cdot 8\big|_9 = 8,$$
$$x_3 = \big||8-2|_{11} \cdot 7\big|_{11} = 9, \qquad x_4 = \big||10-2|_{13} \cdot 5\big|_{13} = 1.$$

Получили $X_2 = \langle 4, 8, 9, 1\rangle$. Остался последний шаг масштабирования с коэффициентом 2^2, на котором $c_i = \{3, 4, 4, 10\}$ и $k = 2$. Вычисление $|X_2|_4$ с использованием второй строки из H приводит к $|X_2|_4 = 1$. С использованием второй строки из P вычисляются искомые остатки y_i:

$$y_1 = \big||4-1|_7 \cdot 2\big|_7 = 6, \qquad y_2 = \big||8-1|_9 \cdot 7\big|_9 = 4,$$
$$y_3 = \big||9-1|_{11} \cdot 3\big|_{11} = 2, \qquad y_4 = \big||1-1|_{13} \cdot 10\big|_{13} = 0.$$

Получен результат $Y = \langle 6, 4, 2, 0\rangle = 13 = \lfloor 3413/2^8 \rfloor$.

5. Программная реализация алгоритмов масштабирования

Новые алгоритмы масштабирования реализованы на языке C для центральных и графических процессоров (CPU и GPU соответственно). Для CPU реализованы последовательные версии. Для GPU реализованы последовательные и параллельные версии алгоритмов с использованием архитектуры параллельных вычислений NVIDIA CUDA. Рассмотрим подробнее вопросы реализации алгоритмов на GPU.

5.1. Технология Compute Unified Device Architecture

Технология CUDA, анонсированная компанией NVIDIA в 2007 г., позволяет использовать GPU для высокопроизводительных вычислений общего назначения. В основе вычислительной архитектуры CUDA лежит блочно-сеточная организация, согласно которой параллельные GPU-потоки объединяются в сетки (grid). Внутри сетки происходит разделение потоков на блоки (block), а внутри блока — на варпы (warp) — группы из 32 потоков, выполняющихся синхронно в режиме SIMD (single instruction, multiple data). Ветвления в варпе могут приводить к дивергенции потоков (branch divergence). При дивергенции все возможные пути ветвления обрабатываются последовательно, что негативно сказывается на производительности [27].

В CUDA определены следующие типы памяти: глобальная память, константная память, локальная память, текстурная память, разделяемая память, регистровый файл. Глобальная (global) память — это основная абстракция CUDA, позволяющая обмениваться данными между CPU и GPU. К ней имеют доступ все потоки. Разделяемая (shared) память используется для эффективного обмена данными между потоками на уровне блоков. В терминах быстродействия разделяемая память приблизительно в 10 раз медленнее регистров и также в 10 раз быстрее глобальной памяти [27]. Разделяемая память разбита на 32 блока, называемых банками. Запросы всех потоков варпа к разным банкам памяти выполняются параллельно. Одновременное обращение нескольких потоков к одному банку памяти для выборки нескольких различных значений называется конфликтом банков памяти. При возникновении конфликта обращения выполняются последовательно, что приводит к снижению пропускной способности.

При написании программного кода CUDA используется специальный упрощенный диалект языка C. Интерфейс программирования приложений (CUDA Runtime API) определяет три типа функций, используемых в GPU-подпрограммах:

- `host`-функции, вызываемые с CPU и выполняемые на CPU;
- `global`-функции, вызываемые с CPU и выполняемые на GPU (при использовании динамического параллелизма могут вызываться с GPU);
- `device`-функции, вызываемые с GPU и выполняемые на GPU.

При вызове `global`-функций, которые также называются ядрами (CUDA kernel), задается конфигурация вычислительной сетки, в соответствии с которой создается необходимое количество параллельных GPU-потоков. При вызове `device`-функций создания новых потоков не происходит. Функции данного типа могут вызываться непосредственно из CUDA ядра либо из других `device`-функций. В новых версиях CUDA введена концепция динамического параллелизма, позволяющая ядрам CUDA запускать другие ядра [27]. Это в некоторых случаях обеспечивает повышение производительности, избавляя от необходимости в промежуточных обменах данными между CPU и GPU.

5.2. Особенности CUDA-реализации алгоритмов масштабирования

Поскольку алгоритмы масштабирования предназначены для использования в других программах, выполняющих вычисления в СОК, CUDA-подпрограммы разработаны в виде `device`-функций. Для параллельных реализаций не используется динамический параллелизм, приводящий в данном случае (многократные запуски при небольшом объеме работы параллельных потоков) к существенным временным задержкам, связанным с управлением дочерними вычислительными сетками. Вместо этого индекс потока передается в качестве аргумента вызываемой функции, а взаимодействие между потоками для вычисления k и $|X|_K$ осуществляется через разделяемую память. Таким образом, все параллельные потоки должны выполняться в одном и том же блоке потоков. Если число модулей СОК превышает максимальный размер блока потоков, то взаимодействие между потоками должно выполняться через глобальную память.

Одной из основных операций в представленных алгоритмах является суммирование, которое требуется для вычисления k_{low}, k_{upp}, h и $|X|_K$. От эффективности данной операции существенно зависит быстродействие параллельных CUDA-реализаций. Классическая схема параллельной редукции относительно сложения на примере вычисления суммы $S = \sum_{i=1}^{8} a_i$ представлена на рис. 2, *а*. В контексте эффективной реализации на GPU эта схема обладает недостатками: во-первых, возникает дивергенция потоков; во-вторых, такая схема приводит к конфликтам банков памяти; в-третьих, число слагаемых (модулей СОК) при использовании классической схемы должно быть кратно степени двойки, что делает алгоритмы менее универсальными. Поэтому в реализованных алгоритмах масштабирования используется модифицированная схема параллельной редукции, основанная на технике из [28]. Она представлена на рис. 2, *б* на примере вычисления суммы $S = \sum_{i=1}^{10} a_i$. Суть этой схемы заключается в использовании реверсивного цикла с последовательной `threadID`-адресацией. В схеме применяется предустановленное смещение $\texttt{offset} = \max\{2^i \mid i < \log_2 n\}$, где n — количество модулей СОК. На первой итерации редукции GPU-поток с номером `threadID` суммирует элементы с индексами `threadID` и (`offset + threadID`) при условии, что $\texttt{offset + threadID} < n$ (нумерация начинается с нуля). Элемент с индексом `threadID` переносится на следующий уровень, если $\texttt{offset + threadID} \geq n$. При переходе к следующей итерации

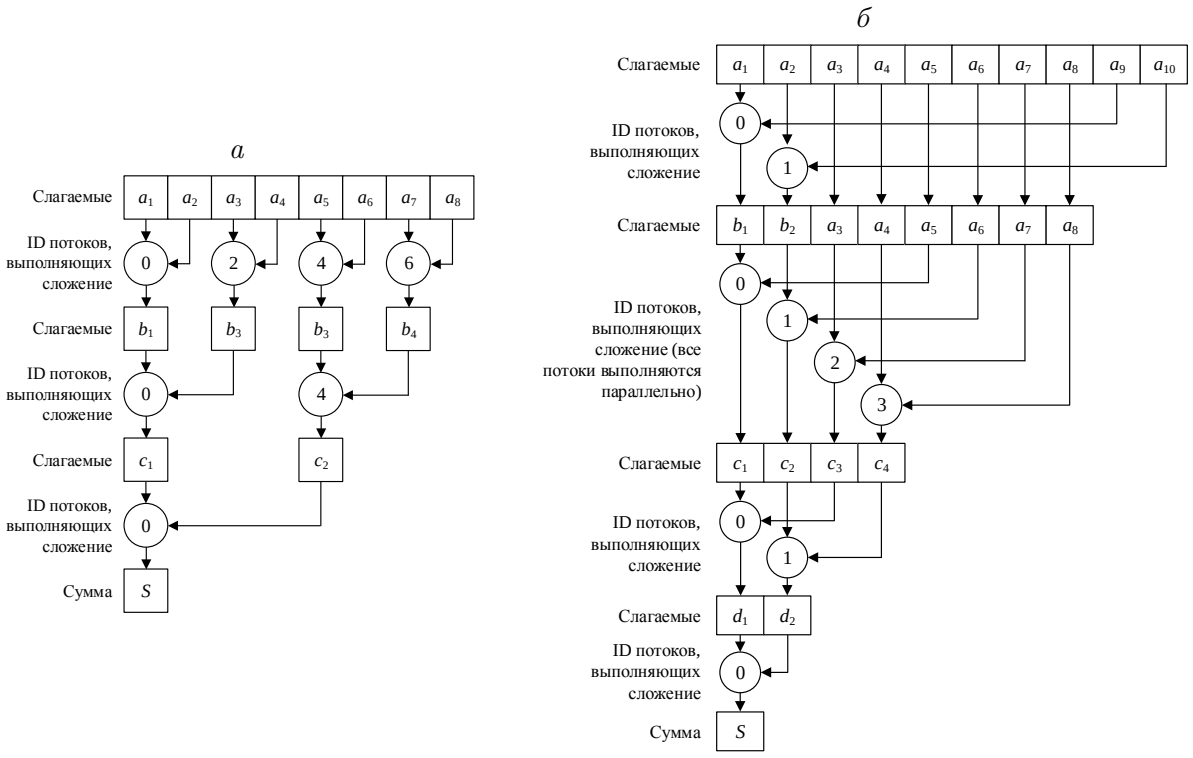

Рис. 2. Параллельная редукция на GPU: *а* — классическая схема; *б* — используемая модифицированная схема

редукции `offset` сдвигается на один разряд вправо. Итерации выполняются до тех пор, пока `offset` > 0. Такая схема позволяет "пропускать" часть элементов с верхних уровней редукции на нижние, тем самым суммировать произвольное число слагаемых.

6. Оценка эффективности

Быстродействие алгоритмов масштабирования оценивалось на различных наборах нечетных модулей. Их параметры представлены в таблице. Тестовая система: Intel Xeon X5650, 12M Cache, 2.66 GHz, 70 GB RAM / GPU Tesla M2050, 515 GFlops (double precision), 3GB GDDR5 / CentOS 6.5 / gcc 4.4.7 / CUDA 5.5. Исходные данные для масштабирования равномерно распределены в диапазоне от 0 до $M - 1$.

Наборы модулей СОК для экспериментов

Число модулей n	Динамический диапазон, M	
	Значение	Разрядность, биты
8	$4.30067 \cdot 10^{16}$	55
16	$2.31783 \cdot 10^{34}$	114
24	$8.19294 \cdot 10^{52}$	175
32	$1.35986 \cdot 10^{72}$	239
40	$8.31452 \cdot 10^{91}$	305
48	$1.94847 \cdot 10^{112}$	373
56	$1.26368 \cdot 10^{133}$	442
64	$2.00869 \cdot 10^{154}$	512

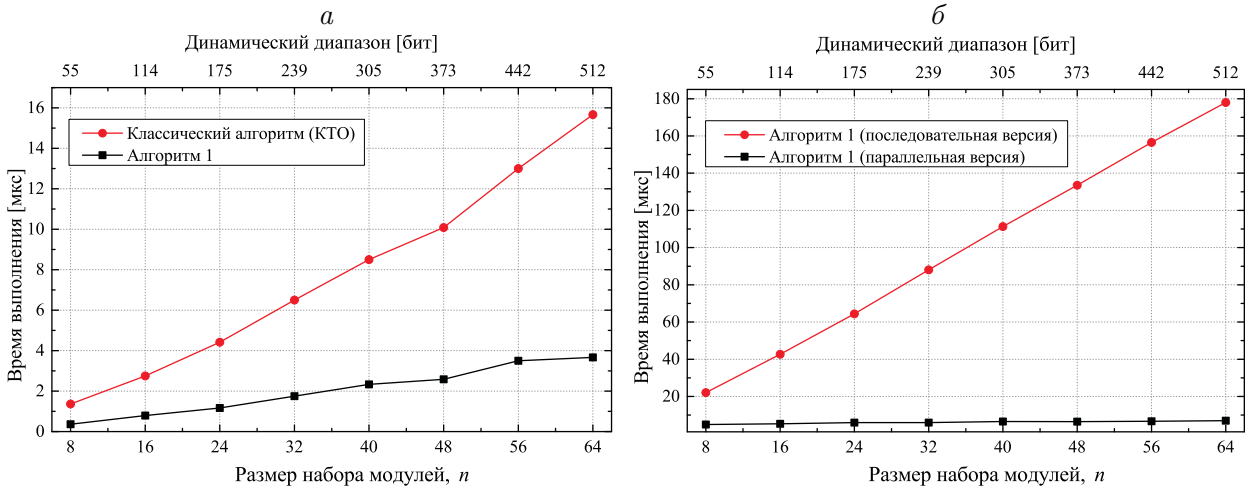

Рис. 3. Эффективность алгоритмов масштабирования в СОК: *а* — CPU Intel Xeon X5650; *б* — GPU NVIDIA Tesla M2050

Эксперимент 1. Исследовались CPU- и CUDA-реализации алгоритма 1 при фиксированном коэффициенте масштабирования $K = 727$, взаимно простом со всеми модулями. Производительность CPU-реализации сравнивалась с классическим алгоритмом, основанным на преобразовании из СОК в двоичную систему с использованием КТО, масштабировании в двоичной системе и преобразовании результата обратно в СОК. Для обработки больших чисел использовалась библиотека длинной арифметики GMP.

На рис. 3, *а* представлены результаты оценки эффективности CPU-реализации алгоритма 1. Как видно, новый алгоритм существенно быстрее классического алгоритма, и с увеличением числа модулей выигрыш в производительности нового алгоритма возрастает. Реализации обоих алгоритмов выполнены без распараллеливания вычислений.

На рис. 3, *б* представлены результаты оценки CUDA-реализаций алгоритма 1. Они свидетельствуют о высокой эффективности распараллеливания. При изменении n от 8 до 64 время последовательной реализации возрастает линейно (с 22 до 178 мкс), тогда как время параллельной реализации остается практически неизменным (5 мкс при $n = 8$ и 7 мкс при $n = 64$). При этом ускорение параллельной версии возрастает с четырех до 25 раз.

Эксперимент 2. Исследовалась производительность алгоритма 2 для масштабирования чисел в СОК степенью двойки при различных значениях T: 1, 15 и 30. Оценивались последовательная CPU-реализация и параллельная CUDA-реализация алгоритма (соотношение между последовательной и параллельной версиями алгоритма 2 такое же, как для алгоритма 1). Коэффициенты масштабирования выбирались из интервалов $[2^1, 2^{32}]$ и $[2^1, 2^{\lfloor \log_2 \sqrt{M} \rfloor}]$. В качестве аналогов рассматривались:

— классический алгоритм на основе КТО (только для CPU);

— алгоритм итерационного деления на двойку с контролем четности на каждом шаге масштабирования [13] (в дальнейшем именуется алгоритмом проверки четности). Для контроля четности масштабируемого числа использовался метод из [22], согласно которому четность числа X, представленного в СОК остатками $x_i = |X|_{m_i}$, определяется следующим образом:

$$P = LSB\big(|x_1\alpha_1|_{m_1}\big) \oplus LSB\big(|x_2\alpha_2|_{m_2}\big) \oplus \cdots \oplus LSB\big(|x_n\alpha_n|_{m_n}\big) \oplus LSB\big(k\big), \qquad (13)$$

где LSB — младший бит числа, а k вычислялось так же как и в алгоритме 2. Для масштабирования коэффициентом 2^D требуется D итераций. Алгоритм был реализован для CPU (последовательная версия) и GPU (параллельная версия).

Результаты экспериментов представлены на рис. 4 и 5. При относительно небольших коэффициентах масштабирования (рис. 4) и пороге $T = 30$ новый алгоритм обеспечивает высокую скорость масштабирования в СОК по сравнению с рассмотренными аналогами, поскольку для выполнения операции требуется в среднем только одна итерация. Так, при $n = 64$ его последовательная CPU-реализация обеспечивает ускорение в 5 и 8 раз по сравнению с классическим алгоритмом и алгоритмом проверки четности соответственно. В свою очередь, параллельная CUDA-реализация алгоритма 2 обеспечивает ускорение в 8 раз по сравнению с параллельной CUDA-реализацией алгоритма проверки четности. При $T = 15$ для масштабирования требуется в среднем две итерации,

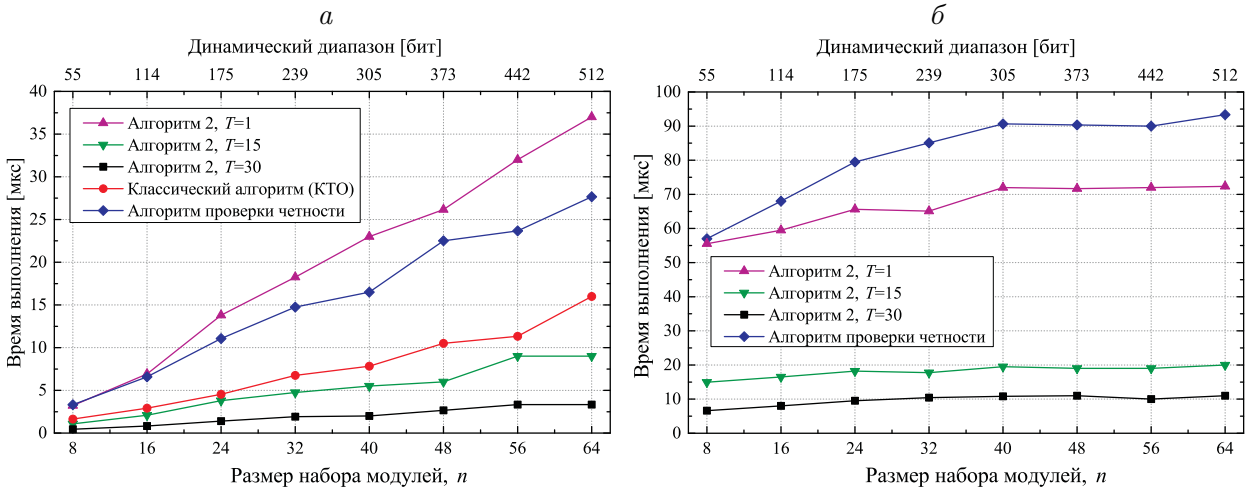

Рис. 4. Эффективность алгоритмов масштабирования в СОК степенью двойки. Коэффициенты масштабирования распределены равномерно в интервале $[2^1, 2^{32}]$: *а* — CPU Intel Xeon X5650; *б* — GPU NVIDIA Tesla M2050

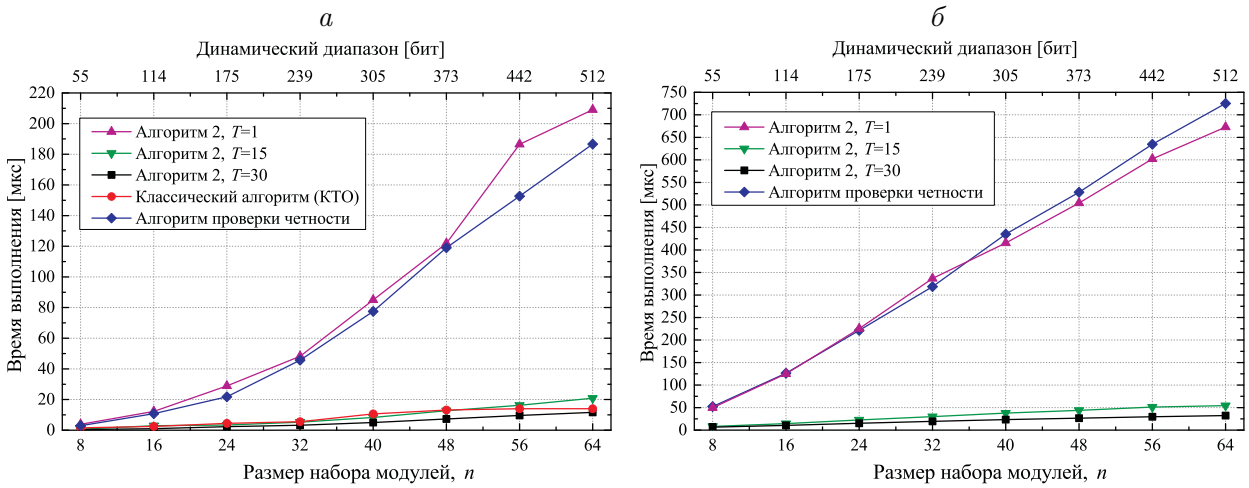

Рис. 5. Эффективность алгоритмов масштабирования в СОК степенью двойки. Коэффициенты масштабирования распределены равномерно в интервале $[2^1, 2^{\lfloor \log_2 \sqrt{M} \rfloor}]$: *а* — CPU Intel Xeon X5650; *б* — GPU NVIDIA Tesla M2050

что ожидаемо приводит приблизительно к двукратному замедлению работы алгоритма по сравнению с $T = 30$. При $T = 1$ требуется D итераций для масштабирования коэффициентом 2^D, поэтому производительность алгоритма 2 сравнима с производительностью алгоритма проверки четности и существенно ниже производительности классического алгоритма на основе КТО.

Параллельная CUDA-реализация алгоритма 2 оказалась эффективнее параллельной реализации алгоритма проверки четности даже при $T = 1$ (см. рис. 4, *б*). Одной из причин этого является необходимость в дополнительной параллельной редукции для вычисления четности числа в соответствии с (13). В алгоритме 2 после вычисления k во всех оставшихся операциях на каждой итерации какие-либо зависимости по данным отсутствуют. Второй вероятной причиной замедления параллельной версии алгоритма проверки четности является дивергенция GPU-потоков, возникающая при анализе четности числа на каждой итерации масштабирования.

При увеличении коэффициента масштабирования время работы алгоритма 2 возрастает (см. рис. 5), что связано с выполнением большего числа итераций для масштабирования каждого числа. Тем не менее при $T = 30$ алгоритм остается быстрее аналогов. Дальнейшее улучшение производительности возможно за счет увеличения T.

Заключение

Рассмотрены алгоритмы масштабирования в СОК, ориентированные на произвольные наборы модулей с большими динамическими диапазонами. Основное их отличие от известных алгоритмов состоит в использовании новой процедуры определения остатка от деления масштабируемого числа на коэффициент масштабирования, основанной на интервальной арифметике, благодаря чему представленные алгоритмы требуют выполнения только стандартных операций машинной точности и эффективно реализуются на различных вычислительных архитектурах, включая CPU и GPU.

Первый алгоритм применим для коэффициентов масштабирования произвольного вида, взаимно простых с модулями и не превышающих машинного слова. Второй алгоритм предназначен для масштабирования в СОК степенью двойки произвольной величины. Он позволяет во время выполнения программы выбирать в качестве коэффициента масштабирования любое число 2^D в пределах диапазона СОК.

Алгоритмы реализованы для CPU- и CUDA-совместимых GPU. Параллельные CUDA-подпрограммы оформлены в виде `device`-функций и используют разделяемую память для обмена данными между потоками. В них применена эффективная схема параллельной редукции, исключающая конфликты банков памяти. Эксперименты показывают, что новые алгоритмы эффективно распараллеливаются, а также существенно быстрее аналогов, основанных на Китайской теореме об остатках и проверке четности.

Предложенный алгоритм масштабирования степенью двойки используется для быстрого округления и выравнивания порядков в гибридной CPU — GPU-библиотеке для вычислений с плавающей точкой многократной точности на основе СОК [9].

Благодарности. Работа выполнена при финансовой поддержке РФФИ (грант № 16-37-60003 мол_а_дк).

Список литературы / References

[1] **Акушский И.Я., Юдицкий Д.И.** Машинная арифметика в остаточных классах. М.: Сов. радио, 1968. 440 с.
Akushskiy, I.Ya., Yuditskiy, D.I. Machine arithmetic in residual classes. Moscow: Sov. Radio, 1968. 440 p. (In Russ.)

[2] **Szabo, N.S., Tanaka, R.I.** Residue arithmetic and its application to computer technology. N.Y.: McGraw-Hill, 1967. 236 p.

[3] **Omondi, A., Premkumar, B.** Residue number systems: theory and implementation. London: Imperial College Press, 2007. 312 p.

[4] **Bajard, J.-C., Eynard, J., Merkiche, N.** Montgomery reduction within the context of residue number system arithmetic // J. Cryptogr. Eng. 2017. P. 1–12. https://doi.org/10.1007/s13389-017-0154-9

[5] **Antão, S., Bajard, J.-C., Sousa, L.** RNS-based elliptic curve point multiplication for massive parallel architectures // Comput. J. 2012. Vol. 55, No. 5. P. 629–647.

[6] **Cardarilli, G.C., Nannarelli, A., Re, M.** RNS applications in digital signal processing // Embedded Systems Design with Special Arithmetic and Number Systems. Springer Intern. Publ., 2017. P. 181–215. https://doi.org/10.1007/978-3-319-49742-6_8

[7] **Chen, J., Yatskiv, V., Sachenko, A., Su, J.** Wireless sensor networks based on modular arithmetic // Radioelectron. Commun. Syst. 2017. Vol. 60, No. 5. P. 215–224.

[8] **Younes, D., Steffan, P.** Efficient image processing application using residue number system // Proc. of the 20th Intern. Conf. Mixed Design of Integrated Circuits and Systems (MIXDES). Gdynia: IEEE, 2013. P. 468–472.

[9] **Isupov, K., Kuvaev, A., Popov, M., Zaviyalov, A.** Multiple-precision residue-based arithmetic library for parallel CPU-GPU architectures: data types and features // Lecture Notes in Computer Science. 2017. Vol. 10421. P. 196–204.

[10] **Kong, Y., Phillips, B.** Fast scaling in the residue number system // IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 2009. Vol. 17, No. 3. P. 443–447.

[11] **Low, J.Y.S., Chang, C.H.** A VLSI efficient programmable power-of-two scaler for $\{2^n - 1, 2^n, 2^n + 1\}$ RNS // IEEE Trans. Circuits Syst. I. 2012. Vol. 59, No. 12. P. 2911–2919.

[12] **Isupov, K., Knyazkov, V.** Interval estimation of relative values in residue number system // J. of Circuits, Systems and Computers. 2018. Vol. 27, No. 1. P. 1850004.

[13] **Meyer-Bäse, U., Stouraitis, T.** New power-of-2 RNS scaling scheme for cell-based IC design // IEEE Trans. Very Large Scale Integr. Syst. 2003. Vol. 11, No. 2. P. 280–283.

[14] **García, A., Lloris A.** A look up scheme for scaling in the RNS // IEEE Trans. Comput. 1999. Vol. 48, No. 7. P. 748–751.

[15] **Cardarilli, G.C., Del Re, A., Nannarelli, A., Re, M.** Programmable power-of-two RNS scaler and its application to a QRNS polyphase filter // Proc. of the 2005 IEEE Intern. Symp. on Circuits and Systems. Kobe: IEEE, 2005. P. 1102–1105.

[16] **Czyżak, M., Smyk R., Ulman Z.** Pipelined scaling of signed residue numbers with the mixed-radix conversion in the programmable gate array // Poznan Univ. of Technology Acad. J. Electrical Eng. 2013. No. 76. P. 89–99.

[17] **Jullien, G.A.** Residue number scaling and other operations using ROM arrays // IEEE Trans. Comput. 1978. Vol. 27, No. 4. P. 325–336.

[18] **Shenoy, A.P., Kumaresan, R.** A fast and accurate RNS scaling technique for high speed signal processing // IEEE Trans. Acoustics. Speech. Slgnal Process. 1989. Vol. 37, No. 6. P. 929–937.

[19] **Barsi, F., Pinotti, M.C.** Fast base extension and precise scaling in RNS for look-up table implementation // IEEE Trans. Signal Process. 1995. Vol. 43, No. 10. P. 2427–2430.

[20] **Ananda Mohan, P.A.** Scaling, base extension, sign detection and comparison in RNS // Residue Number Systems. 2016. P. 133–162. https://doi.org/10.1007/978-3-319-41385-3_6

[21] **Shenoy, A.P., Kumaresan, R.** Fast base extension using a redundant modulus in RNS // IEEE Trans. Comput. 1989. Vol. 38, No. 2. P. 292–297.

[22] **Lu, M., Chiang, J.-S.** A novel division algorithm for the residue number system // IEEE Trans. Comput. 1992. Vol. 41, No. 8. P. 1026–1032.

[23] **Sousa, L.** 2^n RNS scalers for extended 4-moduli sets // IEEE Trans. Comput. 2015. Vol. 64, No. 12. P. 3322–3334.

[24] **Hiasat, A.** Efficient RNS scalers for the extended three-moduli set $(2^n - 1, 2^{n+p}, 2^n + 1)$ // IEEE Trans. Comput. 2017. Vol. 66, No. 7. P. 1253–1260.

[25] **Soderstrand, M., Vernia, C., Chang, J.-H.** An improved residue number system digital-to-analog converter // IEEE Trans. Circuits Syst. 1983. Vol. 30, No. 12. P. 903–907.

[26] **Vu, T.V.** Efficient implementations of the chinese remainder theorem for sign detection and residue decoding // IEEE Trans. Comput. 1985. Vol. 34, No. 7. P. 646–651.

[27] **Wilt, N.** The CUDA handbook: a comprehensive guide to GPU programming. N.J.: Addison-Wesley, 2013. 528 p.

[28] **Harris, M.** Optimizing parallel reduction in CUDA. Available at: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf (accessed 11.10.2017).

Поступила в редакцию 31 октября 2017 г.,
с доработки — 26 февраля 2018 г.

Efficient scaling in RNS using interval estimations

Isupov, Konstantin S.*, Knyazkov, Vladimir S., Kuvaev, Alexander S.

Vyatka State University, Kirov, 610000, Russia

*Corresponding author: Isupov, Konstantin S., e-mail: ks_isupov@vyatsu.ru

Purpose. This paper addresses an acceleration of the scaling operation in the residue number system (RNS). In RNS, addition, subtraction and multiplication are concurrently performed on the n digits (residues) within n parallel channels, and this is the primary advantage of RNS over the binary system. However, some basic operations are more complex in RNS than in binary system. Scaling, i.e. division by a constant factor, is one of such operations. The impossibility of effective scaling prevents a more widespread use of parallel RNS arithmetic.

Methodology. We developed two RNS scaling algorithms, focused on arbitrary moduli sets. In these algorithms, in order to determine the remainder when dividing

the number to be scaled by the scaling factor, interval estimation for the relative value of an RNS representation is used. The first algorithm is applicable for scaling factors that do not exceed the machine word size. The second algorithm is designed for scaling by an arbitrary power of two. Both algorithms require only machine-precision integer and floating-point operations, so they are well suited for implementation on many computing architectures.

Findings. The developed algorithms are implemented in C language for CPU and GPU using NVIDIA CUDA, and are tested on moduli sets of sizes from 8 to 64, which provide the dynamic ranges with bit sizes from 55 to 512, respectively. Performance of new algorithms is shown to be much higher than for the algorithms based on the Chinese remainder theorem and parity checking. In addition, new algorithms are well parallelized: increasing the number of RNS moduli leads only to a slight decrease in the performance of parallel CUDA implementations.

Originality/value. The proposed scaling algorithms can be used in many RNS applications that require a dynamic range reduction, for example, in digital signal processing and multiple-precision arithmetic. In particular, the proposed power-of-two scaling algorithm is used for fast rounding and exponent alignment in the hybrid CPU-GPU software library for multiple-precision floating-point computations based on RNS, which is developed by the authors.

Keywords: residue number system, scaling, parallel algorithms, CUDA programming.

Acknowledgements. This research was supported by RFBR (grant No. 16-37-60003 mol_a_dk).

Received 31 October 2017
Received in revised form 26 February 2018