Масштабируемые алгоритмы целочисленной арифметики и организация поддержки рациональных вычислений в гетерогенных средах

В.А. Голодов, А.В. Панюков

ФГБОУ ВПО «Южно-Уральский государственный университет»
(Национальный исследовательский университет)

Для алгоритмического анализа крупномасштабных проблем, чувствительных к ошибкам округления разрабатывается программное обеспечение, реализующее точные дробно-рациональные вычисления для распределенной вычислительной среды с интерфейсом MPI. Дальнейшее повышение эффективности программного обеспечения возможно за счет применения гетерогенных вычислительных систем, позволяющих выполнять локальные арифметические операции с числами сверхбольшой разрядности параллельно в большом числе процессов. В работе представлено исследование масштабируемости алгоритмов основных арифметических операций и методы ее повышения. Показана возможность повышения эффективности программного обеспечения за счет применения массового параллелизма в гетерогенных вычислительных системах. Использование избыточной позиционной системы счисления, предложенной в работе, позволяет выполнять операцию алгебраического сложения за константное время, что позволяет построить хорошо масштабируемые алгоритмы выполнения всех основных арифметических операций с целыми числами. Масштабируемость основных алгоритмов целочисленной арифметики легко переносится на дробно-рациональную арифметику.

1. Введение

Вычисления повышенной точности, а также безошибочные рациональные вычисления являются необходимым [1–3] и достаточным [4–6] средством алгоритмического анализа крупномасштабных и/или неустойчивых задач. Решение задач дискретной оптимизации методом элипсоидов [7, 8] требует, чтобы число бит в представлении операндов в пять раз превосходило количество итераций.

В настоящее время такие возможности представляет известная библиотека GMP(The GNU Multiple Precision Arithmetic Library) [9]. Библиотека распространяется под лицензией GNU LGPL, актуальная версия библиотеки GMP 6.0.0 доступна для загрузки с официального сайта проекта. Библиотека использует возможности центральных процессоров, в том числе процессоров архитектуры ARM. Программный код оптимизирован с помощью ассемблерных вставок, соответствующих поддерживаемым библиотекой процессорным семействам, но она *не предоставляет* своим объектам возможность их использования в распределенных вычислениях.

В тоже время потенциал современных мультипроцессорных архитектур, таких как архитектуры Fermi, Kepler, Maxwell, Graphics Core Next, Intel Xeon Phi и т.д. позволяют не только разгрузить центральный процессор, но и полностью портировать вычислительную задачу на сопроцессор, оставив центральному процессору управляющие функции.

Тем самым встает вопрос поддержки вычислений повышенной точности и безошибочных рациональных вычислений в данных вычислительных устройствах. В рамках современных десктопных ЭВМ и небольших вычислительных станций может объединяться до 10 сопроцессоров в относительно небольшом корпусе, который не требует специальной инфраструктуры и может охлаждаться штатным набором вентиляторов компонентов. Это выводит на первый план вопросы координации, интеграции сопроцессоров, обмена данных

между устройствами. Производители вычислительных элементов и программного обеспечения уделяют данным вопросам повышенное внимание.

Помимо организации обмена данных между устройствами важнейшей задачей является обеспечение эффективности их использования и конкурентоспособности гетерогенных решений по сравнению с традиционными решениями на базе многопроцессорных комплексов. Эта задача требует существенно переработать алгоритмы базовых арифметических операций, поскольку традиционные последовательные алгоритмы не могут использовать потенциал массового параллелизма устройств, при этом лишены на сопроцессорах возможности использовать высокие тактовые частоты (с среднем тактовые частоты элементов топовых сопроцессоров в 2-3 раза ниже частот топовых центральных процессоров). Требуется глобальный пересмотр алгоритмов базовых арифметических операций для совместимости с архитектурами современных массивно-параллельных сопроцессоров.

Возможности использования современных гетерогенных вычислительных систем уделено внимание при разработке библиотеки «Exact computation 2.0» [10]. Элементами библиотеки являются классы `overlong` и `rational`. Данные классы позволяют производить безошибочные дробно-рациональные вычисления. Дальнейшее повышение эффективности программного обеспечения, использующего «Exact computation 2.0», возможно за счет применения оптимизации локальных операций длинной арифметики с применением распараллеливания на большое количество потоков.

В работе [11] предложены методы построения базовых параллельных вычислительных структур ПЛИС для операций сложения, вычитания, умножения, ориентированных на представлении данных в коде с большой разрядностью. Технология CUDA (Compute Unified Device Architecture) [12] и язык OpenCL дают средства построения современных комплексов программ для гибких гетерогенных систем. В частности существует опыт использования графических процессоров для реализации операций длинной целочисленной арифметики с применением двоично-десятичного представления [13] и системы остаточных классов [14], [15] в задачах криптографии. В данной статье рассматриваются масштабируемые параллельные алгоритмы выполнения базовых арифметических операций.

2. Алгоритмы базовых арифметических операций для устройств с массовым параллелизмом

Сложение неотрицательных целых чисел

Один из возможных способов распараллеливания классического алгоритма сложения n-разрядных чисел в позиционной системе счисления по основанию R столбиком состоит в следующем. На первой стадии выполняется параллельное поразрядное суммирование, в результате которого в некоторых l разрядах возникает переполнение. После этого возможно выполнние l параллельных процессов распространения переноса.

В силу использования двоичного представления данных в ЭВМ выгодно использовать систему счисления с основанием равным степени двойки. Все алгоритмы далее опираются на представление цифры числа как двоичного слова.

Алгоритм 1 содержит процедуры Digit_Addition, Carry_Propagation, Add_Process, описывающих локальные процессы для каждой цифры, и процедуру Add, в которой определяются параметры и резервируется требуемое количество параллельных локальных процессов.

Для получения суммы $(t_n \ldots t_0)_R = (a_{n-1} \ldots a_0)_R + (b_{m-1} \ldots b_0)_R,\ n \geq m$ требуется вызвать процедуру Add (a, b, n, m, t).

Algorithm 1 Addition

Require: $R = 2^r$, $n \geq m$, $a_i = \left(a_i^{r-1} \ \ldots a_i^0\right)_2$, $i = 0, 1, 2, \ldots, n-1$ and $b_j = \left(b_j^{r-1} \ \ldots b_j^0\right)_2$, $j = 0, 1, 2, \ldots, m-1$;
Ensure: $t = (t_n, ..., t_0)_R = (a_{n-1} \ \ldots \ a_0)_R + (b_{m-1} \ \ldots \ b_0)_R$, $t_n \in \{0, R\}$.

1: **procedure** DIGIT_ADDITION(In: a, b, i, Out: c_i, t_i)
2: $\left(s_i^r \, s_i^{r-1} \ \ldots \ s_i^1 \, s_i^0\right)_2 \leftarrow \left(a_i^{r-1} \, a_i^{r-2} \ \ldots \ a_i^1 \, a_i^0\right)_2 + \left(b_i^{r-1} \, b_i^{r-2} \ \ldots \ b_i^1 \, b_i^0\right)_2$;
3: $t_i \leftarrow \left(s_i^{r-1} \ \ldots \ s_i^1 \, s_i^0\right)_2$ ▷ i-th digit before carry propagation
4: $c_i \leftarrow s_i^r$ ▷ carry value to $(i+1)$-th digit
5: **end procedure**

6: **procedure** CARRY_PROPAGATION(In: n, i, c_i, t, Out: t)
7: **while** $c_i \neq 0$ **do** ▷ there is no carry if $c_i = 0$
8: $i \leftarrow i + 1$;
9: $\left(s_i^r \, s_i^{r-1} \ \ldots \ s_i^1 \, s_i^0\right)_2 \leftarrow t_i + c$
10: $t_i \leftarrow \left(s_i^{r-1} \ \ldots \ s_i^1 \, s_i^0\right)_2$ ▷ i-th digit after carry propagation
11: $c_i \leftarrow s_i^r$ ▷ carry value to next digit
12: **end while**
13: **end procedure**

14: **procedure** ADD_PROCESS(In: a, b, i, Out: t)
15: **var** c_i ▷ for carry of this local process
16: DIGIT_ADDITION(a, b, i, c_i, t_i)
17: SYNCHRONIZATION
18: CARRY_PROPAGATION(n, i, c_i, t)
19: **end procedure**

20: **procedure** _GLOBAL_ADD(In: a, b, Out: n, m, t) ▷ exec add in parallel
21: $n \leftarrow \max\{\text{sizeof } (a), \text{sizeof } (b)\}$
22: $m \leftarrow \min\{\text{sizeof } (a), \text{sizeof } (b)\}$
23: Fork m parallel processes
24: ADD_PROCESS(a, b, i, t)
25: Join
26: **end procedure**

Оценим затраты времени, требуемые для выполнения алгоритма 1 и возможный выигрыш от распараллеливания. Для выполнения процедуры Digit_Addition каждому из локальных процессов достаточно время s, равное времени пересылки. Время выполнения процедур Carry_Propagation и Add_Process может изменяться в пределах $[0, \ n \cdot s]$. Поэтому время выполнения алгоритма 1 в зависимости от исходных данных может изменяться в пределах $[s, \ 2 \cdot n \cdot s]$. Время выполнения сложения строго последовательным алгоритмом (т.е. цифра за цифрой) равно $3n \cdot s$.

Найдем оценку среднего времени выполнения алгоритма 1 при условии, что цифры слагаемых являются случайными числами, равномерно распределенными на отрезке $[0, \ R-1]$. С целью упрощения выкладок будем считать $n = m$. Вероятность переполнения в любом разряде $i = 0, 1, \ldots, n-1$ равна

$$p = \mathrm{P}\left\{a_i + b_i \geq 2^r\right\} = \sum_{l=1}^{2^r-1} \mathrm{P}\left\{a_i = l\right\} \mathrm{P}\left\{b_i \geq 2^r - l\right\} = \sum_{l=1}^{2^r-1} \frac{1}{2^r} \cdot \frac{l}{2^r} = \frac{1}{2}\left(1 - \frac{1}{2^r}\right).$$

Вероятность получения после сложения цифр величины $2^r - 1$ равна

$$q = \mathrm{P}\left\{a_i + b_i = 2^r - 1\right\} = \sum_{l=0}^{2^r-1} \mathrm{P}\left\{a_i = l\right\} \mathrm{P}\left\{b_i = 2^r - 1 - l\right\} = \sum_{l=0}^{2^r-1} \frac{1}{2^r} \cdot \frac{1}{2^r} = \frac{1}{2^r}.$$

Вероятность отсутствия переноса равна

$$P_0 = \mathrm{P}\left\{(\forall i = 0, 1, \ldots, m-1)\,(a_i + b_i \le 2^r)\right\} = (1 - \mathrm{P}\left\{a_i + b_i \ge 2^r\right\})^m = (1-p)^m \to 0,$$

Вероятность отсутствия цепи сквозного переноса (т.е. цепи из более одного последовательного переноса) равна $P_1 = (1 - pq)^m \to 1$. Следовательно, в асимптотике среднее время выполнения алгоритма будет равно $3s$. Таким образом, рассмотренный алгоритм в сравнении с последовательным имеет среднее быстродействие в n раз выше, и не зависит от длины складываемых чисел.

Распространение переносов

Максимальное количество последовательных операций, выполняемых одним процессом, определяется длинной максимального переноса, тем самым в неблагоприятном случае, когда существует как минимум одна цепочка переносов длины более нескольких разрядов, время выполнения сложения можно улучшить за счет совершенствования алгоритма распространения переносов.

Одним из возможных способов такого совершенствования является вычисление цепей распространения переносов параллельно с их распространением. Если перенос попадает на вычисленную цепь, то его распространение по данной цепи осуществляется за один такт. Процедура Carry_Propagation, представленная алгоритмом 2 описывает такое ускоренное распространение сквозных переносов.

Алгоритм состоит в следующем. Первоначально результаты процедуры Digit_Addition (т.е. число t и значения переносов из каждого разряда c) представлены в виде n фрагментов, при этом каждая цифра d_i, $i = 0, 1, \ldots, n-1$ соответствует фрагменту с выделенным процессом i.

На k-й итерации **while** цикла фрагменты ассоциированные с процессами $l2^k$ и $(l+1)2^k$ объединяются в один фрагмент ассоциированный с процессом $(l+1)2^k$.

При объединении младший процесс $l2^k$ посылает старшему процессу $(l+1)2^k$ флаг $NotCarry$ отсутствия сквозной цепи переноса через объединенный фрагмент, значние c переноса из младшего фрагмента, и возможную границу V распространения сквозного переноса в объединенном фрагменте. Старший процесс $(l+1)2^k$ при ненулевом переносе из младшего фрагмента $l2^k$ пересылает его всем ожидающим его цифрам (от $l2^k + 1$-й до $V((l+1)2^k)$-й), устанавливает границу V распространения сквозного переноса в объединенном фрагменте и завершает выполнение процессов, которые далее не учавствуют в распространении переносов.

Оценим затраты времени необходимые для выполнения процедуры Carry_Propagation в алгоритме 2. Данная процедура содержит цикл `''while''`, который выполняется любым и созданных процессов не более $\lceil \log_2 n \rceil$ раз. Каждый из активных процессов выполняет операторы, указанные на строках 2,3,4,5, и 41 указанного цикла. При соответствующей оптимизации гетерогенной среды эти операции могут быть выполнены за один такт. Каждый из активных процессов также выполняет также выполняет не более одной посылки и не более одного приема сообщений объемом $3r$ бит. Подготовка и выполнение данных коммуникаций может быть осуществлено за два такта. Таким образом при соответствующей оптимизации гетерогенной среды, время выполнения алгоритма 2 в среднем не превосходит величины $3s$, а в худшем случае – величины $3s\lceil \log_2 n \rceil$.

Algorithm 2 Improved carry propagation.

```
1: procedure CARRY_PROPAGATION(In: n, i, Out: c, t)
2:     L ← 1, V ← i                                  ▷ length and verge of the joined fragments
3:     while L ≤ n do                                 ▷ there are fragments for joining
4:         M ← i mod 2L
5:         if (M < L) then                            ▷ i belong the low fragment
6:             if (M = L − 1) then                    ▷ i is high digit of the low fragment
7:                 j ← min{i + L, n − 1}              ▷ high digit of joined fragment
8:                 NotCarry ← (t_i ≠ 2^r − 1) ∪ (V ≠ i)   ▷ absent ripple carry
9:                 send {c, NotCarry, V} to process j
10:                if ((c ≠ 0) ∪ NotCarry) then
11:                    terminate process
12:                end if
13:            end if
14:        else                                       ▷ i belong the high fragment
15:            j ← i + L − M − 1                      ▷ j is high digit of the low fragment
16:            flag ← ((M = 2L − 1) ∪ (i = n − 1))
17:            if flag then                           ▷ i is high digit of the high fragment
18:                receive {Cj, NotCarry, Vj} from process j
19:                send {Cj, NotCarry} to processes k = j + 1, ..., V
20:                if (NotCarry) then
21:                    V ← Vj
22:                else if (i = V) then
23:                    (s^r s^{r−1} ... s^1 s^0)_2 ← (c t_i^{r−1} ... t_i^1 t_i^0)_2 + Cj
24:                    t_i ← (s^{r−1} ... s^1 s^0)_2, c ← s^r
25:                end if
26:            else                                   ▷ i is not high digit of the high fragment
27:                if (i ≤ V) then                    ▷ i belong the ripple carry chain
28:                    receive {Cj, NotCarry} from process j + L
29:                    if (Cj ≠ 0) then
30:                        t_i ← 0
31:                        if (i = V) then
32:                            t_{i+1} ← t_{i+1} + 1
33:                        end if
34:                        terminate process
35:                    else if NotCarry then
36:                        terminate process
37:                    end if
38:                end if
39:            end if
40:        end if
41:        L ← 2L
42:    end while
43:    terminate process
44: end procedure
```

Среднее время выполнения операции сложения с применением процедуры ускоренного распространения переносов (т.е. алгоритма 2) оказывается равным $4s$, т.е. превышает среднее время выполнения алгоритма 1 в 4/3 раз. Эффективность использования алгоритма 2 очевидна при наличии цепей сквозного переноса длины более двух цифр.

Бинарные отношения

Для проверки истинности бинарных отношений $a\rho b \ : \rho \in \{<, \leq, =, \geq, >, \neq\}$ достаточна проверка истинности отношений $\rho \in \{=, >\}$. Действительно, $(a \leq b) = \neg(a > b)$, $(a \neq b) = \neg(a = b)$, $(a \geq b) = (a > b) \vee (a = b)$, $(a < b) = \neg(a \geq b)$.

Алгоритм 3 вычисляет истинность бинарных отношений $(a = b)$ and $(a > b)$ для заданных неотрицательных целых a, and b.

Algorithm 3 Checking of truth of the binary relations $(a = b)$ and $(a > b)$.

Require: $a = (a_{n-1} \ldots a_0)_R$, and $b = (b_{n-1} \ldots b_0)_R$, $R = 2^r$, $n > 0$, $a_i = \left(a_i^{r-1} \ \ldots a_i^0\right)_2$, $b_i = \left(b_i^{r-1} \ \ldots b_i^0\right)_2$, $i = 0, 1, 2, \ldots, n-1$

Ensure: p is truth of relation $a = b$, and q represents truth of relation $a > b$.

```
 1: procedure EqG_Process(In: a, b, n, i, Out: p, q)
 2:     p ← (a = b)
 3:     q = (a > b)
 4:     L ← n/2
 5:     while (L > 1) do
 6:         if (i > 0) then
 7:             send {p, q} to process i/2
 8:         end if
 9:         if i < L then
10:             if there is sending from process 2i then
11:                 receive {p_0, q_0}
12:             else p_0 = true, q_0 = false
13:             end if
14:             if there is sending from process 2i + 1 then
15:                 receive {p_1, q_1}
16:             else p_1 = true, q_1 = false
17:             end if
18:             read timing
19:             p ← p_1 ∧ p_0
20:             q ← q_1 ∨ (p_1 ∧ q_0)
21:         else
22:             terminate process
23:         end if
24:         L ← L/2
25:     end while
26: end procedure

27: procedure _global_EqG(In: a, b, n, Out: p, q)
28:     Fork n parallel processes
29:     ExecInParallel EqG_Process(a_i, b_i, n, i, p_i, q_i)      ▷ i is number of process
30:     Join
31:     p ← p_0, q ← q_0
32: end procedure
```

Сущность алгоритма состоит в следующем. Первоначально данные представляют в виде n фрагментов с выделенными процессами $i = 0, 1, \ldots, n-1$, а локальные переменные p_i and q_i процесса i отображают истинность отношений $(a_i = b_i)$ и $(a_i > b_i)$. При k-м выполнении цикла **for** осуществляется слияние фрагментов $l2^k$ и $(l+1)2^k$ в один фрагмент,

ассоциированный с процессом $l2^{k-1}$. При этом вычисляются значения p_i и q_i объединенного фрагмента $i = l2^{k-1}$, процессы $l2^k$ и $(l+1)2^k$ завершаются. Легко заметить, что алгоритм 3 выполняется не менее чем в $n/\log_2 n$ быстрее строго последовательного алгоритма.

Определение количества значащих цифр

Для рационального распределения вычислительных ресурсов для результата выполнения любой арифметической операции необходимо знать количество значащих цифр ее операндов. В таких операциях как сложение, умножение и деление количество значащих цифр результата операции определяется по количеству значащих цифр операндов с погрешностью в одну цифру. Количество значащих цифр результата операции вычитания можно определить только после ее выполнения. Поэтому, рационализация использования вычислительного ресурса требует наличия алгоритма определения количества значащих цифр числа в используемой системе счисления. Алгоритм определния количества значащих цифр во многом схож с алгоритмом для бинарных отношений.

Умножение многозначного числа на однозначное

Алгоритм 4 вычисляет произведение $(c_n, ..., c_0)_R$ заданных неотрицательных целых чисел $a = (a_{n-1}, ..., a_0)_R$ и $b = (b_0)_R$.

Algorithm 4 calculates the product of a and digit b

Require: $a = (a_{n-1} \ \dots \ a_0)_R$, $n > 0$, $b = (b_0)_R$, $R = 2^r$.
Ensure: $(t_n \ t_{n-1} \ ... \ t_0)_R$ is product of a and b.

```
 1: procedure M_Process(In: ad, b, i, n, dt)
 2:     if (i < n) then
 3:         (x_1 x_0)_R ← ad · b
 4:         send x_1 to process (i + 1)
 5:     else
 6:         x_0 ← 0
 7:     end if
 8:     if (i > 0) then
 9:         receive x_1 from process (i − 1)
10:         (s^r s^{r−1} ... s^1 s^0)_2 ← x_0 + x_1
11:         c ← s^r,   dt ← (s^{r−1} ... s^1 s^0)
12:         send c to process (i + 1)
13:         receive c from process (i − 1)
14:         dt ← dt + c
15:     else
16:         dt ← x_0
17:     end if
18: end procedure

19: procedure _global_ M(a, b, n, t)
20:     Fork n parallel processes
21:     ExecInParallel M_Process(a_i, b, i, n, t_i)        ▷ i is number of process
22:     Join
23: end procedure
```

Из описания алгоритма 4 видно, что время его выполнения не превосходит величины

$4s$. Таким образом затраты времени на выполнение алгоритма 4 в n раз меньше времени работы последовательного алгоритма и не зависит от длины операндов.

Умножение многозначных чисел

Алгоритм 5 вычисляет произведение двух неотрицательных чисел.

Algorithm 5 calculates the product $c = a \cdot b$

Require: $a = (a_{n-1} \ldots a_0)_R$, $b = (b_{m-1} \ldots b_0)_R$, $n \geq m > 0$, $R = 2^r$
Ensure: $(c_{n+m-1}, ..., c_0)_R$ is product of a and b

```
 1: procedure MM_Process(a, b, i, n, m, c)
 2:     _global_M(a, b_i, n, z)
 3:     L ← m/2, B ← R
 4:     while (L > 1) do
 5:         if (i > 0) then
 6:             send z to process i/2
 7:         end if
 8:         send timing
 9:         if i < L then
10:             if there is sending from process 2i then
11:                 receive value for s_0
12:             else s_0 ← 0
13:             end if
14:             if there is sending from process 2i + 1 then
15:                 receive value for s_1
16:             else s_1 ← 0
17:             end if
18:             s_1 ← s_1 · B
19:             _global_Add(s_0, s_1, n, m, z)
20:         else
21:             terminate process
22:         end if
23:         L ← L/2, B ← B^2
24:     end while
25:     c ← z
26: end procedure

27: procedure _global_MM(a, b, n, m c)
28:     Fork m parallel processes
29:     ExecInParallel MM_Process(a, b, i, n, m, z)        ▷ i is number of process
30:     Join
31:     c ← z
32: end procedure
```

Время выполнения процедуры MM равно $4s$. Тело цикла **while** (строки 4 – 23) выполняется не более $\lceil \log_2 m \rceil$ раз. Оно содержит не более одной отправляющей (строка 6) и двух принимающих коммуникации (строки 11 и 15) коммуникаций "точка – точка", и одно сложение $(n+m-2L)$-разрядных чисел. Выполнение остальных операторов можно организовать за один такт. Таким образом, в среднем время необходимое для вычисления произведения не превосходит величины $(4 + 3\log_2 m)\, s$. В наихудшем случае время вычисления произведения не превосходит величины $(4 + 3(n+m)\log_2 m)\, s$.

Процедура M of алгоритма 5 порождает m процессов, каждый из которых вызывает процедуру M (строка 2), которая создает n процессов. Следовательно, общее количество процессов равно mn.

В наихудшем случае с увеличением разрядности слагаемых время выполнения растет немного быстрее линейной функции, тогда как последовательный алгоритм умножения столбиком имеет квадратичную зависимость времени выполнения от разрядности сомножителей.

Деление

Классический алгоритм деления «столбиком» не является масштабируемым. В соответствии с данным алгоритмом при его реализации необходимо последовательно выполнить $(n + m - 1)$ операций умножения-вычитания m-разрядных чисел. В работах [16], [17] предложено решение проблемы повышения эффективности алгоритма операции деления посредством применения метода Ньютона.

Чтобы разделить целое число

$$u = (u[n-1]\ u[n]\ \ldots\ u[1]\ u[0])_R$$

на целое число

$$v = (v[m-1]\ \ldots\ v[0])_R,$$

сначала предлагается найти достаточно точное приближение к числу $1/v$, затем умножить его на u, что даст приближение к u/v. Очевидно, что длина целочисленного ответа будет не более $n-m+1$. Число $1/v$ содержит не более m незначащих нулей в старших разрядах, для получения правильного результата деления достаточно, чтобы приближенное значение $1/v$ содержало еще хотя бы $n - m + 1$ значащих цифр. Таким образом, достаточная точность вычисления величины $1/v$ определяется величиной R^{-n+1}.

Применение метода Ньютона к задаче нахождения корня уравнения $f(x) = 0$, где $f(x) = v - 1/x$, состоит в последовательном вычислении

$$x_{k+1} = (2 - v \cdot x_k) \cdot x_k,\ k = 0, 1, 2, \ldots,$$

где x_0 – начальное приближение, вычисленное с достаточной точностью. При $x \geq 1$ функция $f(x)$ является дважды непрерывно дифференцируемой и строго выпуклой. В этом случае метод Ньютона обладает квадратичной скоростью сходимости, т.е. количество правильно вычисленных разрядов после выполнения очередной итерации будет удваиваться. Начальное приближение $x_0 = 1/(v[m-1]$ величины $1/v$ имеет погрешность

$$\frac{1}{v[m-1] \cdot R^{m-1}} - \frac{1}{v} = \frac{v - v[m-1] \cdot R^{m-1}}{v \cdot v[m-1] \cdot R^{m-1}} \leq \frac{1}{v \cdot v[m-1]} \leq R^{-m+1},$$

т.е. в нем правильно вычислено m разрядов. Таким образом, количество итераций, которые потребуется выполнить по методу Ньютона, будет не более $4\log_2(n+1) - \log_2 m$.

Алгоритм 6 вычисляет частное двух неотрицательных целых.

Algorithm 6 calculates the quotient $c = a/b$, a and b are non-negative integers

Require: $a = (a_{n-1} \dots a_0)_R$, $b = (b_{m-1} \dots b_0)_R$, $n \geq m > 0$ are represented in the radix notation on the base $R = 2^r$

Ensure: $(c_{n+m-1}, \dots, c_0)_R$ is the quotient $c = a/b$.

1: **procedure** _GLOBAL_D(a, b, n, m, c)
2: $x \leftarrow \left\lfloor \frac{R-1}{b[m-1]} \right\rfloor_R$, $\tilde{R} \leftarrow R^m$ ▷ Initial approximation
3: **for** $i = 0, 1, ?, \lceil \log_2 \frac{n+1}{m} \rceil$ **do** ▷ More precise definition
4: $d \leftarrow x$, $x \leftarrow \left(2 \cdot \tilde{R} - b \cdot d\right) \cdot d$, $\tilde{R} \leftarrow \tilde{R} \cdot \tilde{R}$
5: **end for**
6: $z = a \cdot x$ ▷ Multiplication
7: $c = z/\tilde{R}$ ▷ Answer forming
8: **end procedure**

На итерации $k = 0, 1, 2, \dots, l < \log_2(n+1) - \log_2 m$ цикла **for** переменная x представляет целое $(2^{k+1} - 1)$-разрядное число. Поэтому в теле цикла с помощью параллельных алгоритмов выполняется одна операция умножения переменной x на m-разрядное число b, одна операция вычитания (2^k)-разрядных чисел и одна операция перемножения (2^k)-разрядных чисел. Следовательно, время необходимое для выполнения шага тела цикла не превосходит величины $11+3(\log_2 m+k)]\cdot s$ в среднем случае и величины $\left[3 \cdot 2^k k + 2.5 \cdot 2^k + 6k + 3m\log_2 m + 10\right] s$ в наихудшем случае.

Поскольку

$$\sum_{k=0}^{l} k = \frac{l(l+1)}{2}, \quad \sum_{k=0}^{l} 2^k = 2^{l+1} - 1,$$

$$\sum_{k=0}^{l} \left(k \cdot 2^k\right) \leq \sqrt{\sum_{k=0}^{l} k^2 \sum_{k=0}^{l} 4^k} = \sqrt{\frac{2l^3 + 3l^2 + l}{6} \cdot \frac{4^{l+1} - 1}{3}} \leq 2^{l+1}\sqrt{\frac{l^3}{3}},$$

то время выполнения тела цикла **"for"** в среднем и наихудшем случаях не превысит соответственно величин $O\left(\log_2 n \cdot \log_2 \frac{n}{m}\right)$ and $O\left(\frac{n}{m} \log_2{}^{3/2} \frac{n}{m}\right)$.

Перемножение n-значных чисел завершает выполнение алгоритма D. Время выполнения данного шага не больше величины $O(\log_2 n)$ в среднем и величины $O(n \cdot \log_2 n)$ в наихудшем случаях. Таким образом, окончательрые оценки времени выполнения алгоритма 6 в среднем и наихудшем случаях равны соответственно

$$O\left(\log_2 n \cdot \log_2 \frac{n}{m}\right) \text{ и } O\left(\frac{n}{m} \log_2{}^{3/2} \frac{n}{m} + n \log_2 n\right).$$

3. Применение знаковых позиционных систем счисления

Приведенная выше система счисления является незнаковой, цифрами в позиционной системе по основанию R являются числа 0, 1, 2, …, $R-2$, $R-1$. Ее недостатком является сложность реализации операций алгебраического сложения и вычитания, т.к. она требует выполнения операции сравнения чисел. Устранить отмеченный недостаток позволяет применение знаковых позиционных систем счисления. Цифрами в знаковой позиционной системе по основанию R являются числа

$$-\left\lfloor \frac{R}{2} \right\rfloor, -\left\lfloor \frac{R}{2} \right\rfloor + 1, \dots, -1, 0, 1, \dots, \left\lceil \frac{R}{2} \right\rceil - 2, \left\lceil \frac{R}{2} \right\rceil - 1.$$

Отметим, что при нечетном R количество положительных и отрицательных цифр одинаково, а при четном R количество положительных на одно меньше количества отрицательных.

Представление числа в знаковой позиционной системе счисления по основанию $R = 2^r$ имеет вид $(a_{n-1}, ..., a_0)_{\pm R}$, а его цифры как $a_i = (a_i^{r-1}\, a_i^{r-2}\, \ldots\, a_i^1\, a_i^0)_{\pm 2}$, $i = 0, 1, \ldots, n-1$. Старший бит в представлении цифры определяет знак цифры (0 для положительных, 1 для отрицательных). В этом случае цифрами системы счисления являются C++-объекты типа `integer`.

Заметим, что все основные алгоритмы, для незнаковых систем счисления, за исключением алгоритмов сложения/вычитания, переносятся на знаковые системы без изменения. Алгоритмы сложения/вычитания объединяются в общий алгоритм алгебраического сложения.

Применение знаковых позиционных систем упрощает алгоритм алгебраического сложения, но не решает проблему повышения эффективности вычислений при наличии сквозных переносов. Преимущества и недостатки ускоренного распространения переносов такие же как в случае безнаковых систем.

Истинность бинарных отношений в знаковых системах реализуется посредством операции вычитания. Знак числа определяется знаком старшего разряда. Алгоритмы определения количества значащих цифр, умножения и деления такие же как в безнаковых системах.

4. Применение избыточных позиционных систем счисления

Изложенный выше анализ показывает на высокую в среднем эффективность применения распараллеливания в алгоритмах всех арифметических операций. При этом среднее время вычисления результатов сложения, вычитания, умножения на одноразрядное число и бинарных отношений имеет величину $O(1)$, среднее время вычисления умножения и деления чисел разрядности n не превосходит величины $O(\log_2^2 n)$.

Однако в наихудшем случае время вычисления результатов любой операции с числами разрядности n оказывается не меньше $O(n)$ при обычном распространении переносов и не меньше величины $O(\log_2 n)$ при ускоренном распространении переносов. Причиной отклонений от среднего значения является то, что при выполнении сложения (вычитания) для распространения переносов появляются цепи длины более единицы.

Исключить отмеченные прецеденты можно за счет использования избыточных позиционных систем счисления, в которых разрешаются цепи переносов длины не более единицы. В качестве недостатка отметим множественность представления чисел в избыточной системе счисления, что позволяет эффективно вычислять бинарные отношения и количество значащих цифр только после распространения переносов, устраняющих избыточность представления. Напомним, что в асимптотике вероятность появления дополнительных переносов стремится к нулю.

5. Реализация поддержки дробно-рациональной арифметики для гетерогенной вычислительной среды на языке C++

Объектами класса `rational` являются обыкновенные дроби p/q, где p, q - объекты класса `overlong`. Класс `overlong` предназначен для расширения логических возможностей целочисленных вычислений на компьютере. Объем памяти, занимаемый такими объектами, определяется значениями представляемых чисел, их диапазон ограничен только объемом адресуемой памяти. Диапазон чисел, представляемых объектами класса `overlong`, расширен до $(-2^{(r\cdot 2^{r-1})}, 2^{(r\cdot 2^{r-1})})$, где r – разрядность используемых цифр. Минимальный шаг дискретизации чисел, представляемых объектами класса `rational`, может достигать $2^{-r2^{r-1}}$. Для объектов классов `overlong` и `rational` определены все операторы, операции и бинарные отношения, используемые для стандартных числовых типов данных, а так-

же интерфейс MPI. Применение библиотеки «Exact computation 2.0» для решения плохо обусловленных систем линейных алгебраических уравнений [18], [5] и задач линейного программирования [4] показало ее высокую эффективность.

Для решения задач, требующих применения точной дробно-рациональной арифметики наиболее эффективным является применение позиционной системы счисления с основанием $R = 2^r$. В работе изложены основные алгоритмы целочисленной арифметики в позиционной системе счисления по основанию R для гетерогенных вычислительных систем с анализом их масштабируемости анонсированные на конференциях [5, 17, 19, 20]. Для более полноценного использования современных вычислительных архитектур классы `overlong` и `rational` хранят и оперируют числами по основанию 2^{32}, это позволяет оптимизировать проведение операций над числами с помощью быстрых бинарных логических операций.

Оптимизации применяются также при работе с памятью. Поскольку в C++ нет автоматического сборщика мусора, то избыточные перевыделения памяти приводят её фрагментации и снижению быстродействия приложения в целом. Краткое описание современных реализаций классов для процессоров семейств x86 и x86-64 дано в [21] описание особенностей реализации классов в гетерогенной среде приводится также в [22].

Операции с памятью инкапсулированы в отдельный класс `MemHandle`, а выполнение базовых арифметических операций с разрядами полностью производится в рамках класса `ArifRealization` (см. фрагмент листинга 1). Реализации классов отделены от интерфейсов, для каждой из используемых архитектур написаны соответствующим образом оптимизированные реализации, учитывающие большое количество особенностей применяемой архитектуры.

```
1  class overlongNM {
2   private:
3   static ArifRealization realization;
4   MemHandle mhandle;
5   ...
6   public:
7   inline int32 size() const;//length
8   inline int32 sign() const;//sign
9   ...
10  //addition
11  template<typename Type>
12  friend const overlongNM operator+
13    (const overlongNM &num,Type v)
14  {overlongNM rez(num); return (rez+=v);}
15  friend const overlongNM operator+
16    (const overlongNM&,const overlongNM&);
17  ...
18 }
```

Листинг 1: Фрагмент класса `overlong`

Объект класса `overlong` содержит в себе объект типа `MemHandle`, и все действия с памятью происходят через интерфейс класса `MemHandle`. Все арифметические операции с данными осуществляются вызовом соответствующих методов класса `ArifRealization`. Таким образом данные всегда хранятся в памяти того же вычислительного устройства на котором обрабатываются. Для CPU используется оперативная память, для GPU - глобальная память графического ускорителя.

Эти особенности реализации позволяют в момент запуска приложения выбрать оптимальное вычислительное устройство и, тем самым, место хранения разрядов чисел. В рамках MPI приложения все процессы выбирают вычислительное устройство независимо друг от друга, то есть приложение запущенное на разнородном кластере будет использовать наиболее предпочтительное устройство для каждого потока. Особенностью GPU реализаций

является требование к масштабируемости алгоритмов, применяемых для работы с разрядами чисел. В силу меньшей тактовой частоты ядер чипа GPU и их большого числа, применение параллельных масштабируемых алгоритмов является не только оптимизационной особенностью, но и необходимостью.

6. Заключение

Повышение эффективности программного обеспечения операций дробно-рациональной арифметики возможно за счет применения массового параллелизма в гетерогенных вычислительных средах. Эффективное использование массового параллелизма GPU требует переработки алгоритмов основных арифметических операций в их масштабируемые варианты. Реализация аппарата поддержки арифметики высокой точности требует, помимо проработки пользовательского интерфейса, существенно учитывать возможность применения различных устройств. Дальнейшее повышение эффективности возможно, например, за счет использования предложенной в работе избыточной позиционной системы счисления, которая позволяет строить хорошо масштабируемые алгоритмы выполнения основных арифметических операций.

Список литературы

1. Alt, R. On the accuracy of the solution of linear problems on the CELL processor / R. Alt, J.-L. Lamotte, S. Markov // Reliable Computing. — 2011. — Vol. 15. — Pp. 1–12.

2. Beaumont, O. Linear interval tolerance problem and linear programming techniques / O. Beaumont, B. Philippe // Reliable Computing. — 2001. — Vol. 6, no. 4. — Pp. 365–390.

3. Coxson, G. E. Computing exact bounds on elements of an inverse interval matrix is NP-hard / G. E. Coxson // Reliable Computing. — 1999. — Vol. 5, no. 2. — Pp. 137–142.

4. Panyukov, A. V. Using massively parallel computations for absolutely precise solution of the linear programming problems / A. V. Panyukov, V. V. Gorbik // Automation and Remote Control. — 2012. — Vol. 73, no. 2. — Pp. 276–290.

5. Panyukov, A. V. Exact and guaranteed accuracy solutions of linear programming problems by distributed computer systems with mpi / A. V. Panyukov, V. V. Gorbik // Tambov University Reports. Series: Natural and Technical Sciences. — 2010. — Vol. 15, no. 4. — Pp. 1392–1404.

6. Panyukov, A. V. Computing the best possible pseudo-solutions to interval linear systems of equations // 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numeric (SCAN'2012, Novosibirsk, Russia, September 23-29, 2012): Book of abstracts. — Institute of Computational Technologies Publisher, 2012. — Pp. 134–135.

7. Panyukov, A. Polynomial solvability of n np-complete problems / A. Panyukov // ScienceOpen Research. — 2015.

8. Grötschel, M. The ellipsoid method and its consequences in combinatorial optimization / M. Grötschel, L. Lovász, A. Schrijver // Combinatorica. — 1981. — Vol. 1, no. 2. — Pp. 169–197.

9. The gnu mp bignum library. — February 2013. http://gmplib.org/.

10. Голодов, В. Библиотека классов «Exact Computation 2.0» свидетельство о государственной регистрации программы для ЭВМ №2013612818 от 14 марта / В. Голодов, А. В.

Панюков // Программы для ЭВМ, базы данных, топологии интегральных микросхем. Официальный бюллетень Российского агентства по патентам и товарным знакам. — М.: ФИПС, 2013. — № 3. — С. 251.

11. Золотовский, В. Реализация «длинных вычислений» на параллельных структурах / В. Золотовский, М. Ф. Гильванов // Известия высших учебных заведений. Северо-Кавказский регион. Серия: Технические науки. — 2008. — № 4. — С. 9–13.

12. Parallel programming and computing platform | cuda | nvidia. — Febbuary 2013. htpp://www.nvidia.com/object/cuda_home.html.

13. Желтов, С. А. Реализация арифметических операций с "длинными"числами на устройствах gpgpu / С. А. Желтов // Вопросы защиты информации. — 2012. — № 3. — С. 2–4.

14. Орлов, Д. Реализация арифметики повышенной разрядности на графических процессорах / Д. Орлов // Программная инженерия. — 2012. — № 4. — С. 33–43.

15. Дзегеленок, И. И. Алгебраизация числовых представлений для обеспечения высокоточных суперкомпьютерных вычислений / И. И. Дзегеленок, Ш. А. Оцопков // Вестник Московского энергетического института. — 2010. — № 3. — С. 107–116.

16. Knuth, D. E. The Art of Computer Programming / D. E. Knuth. — 2 edition. — Addison-Wesley Longman, 1981. — Vol. 2. — P. 688.

17. Panyukov, A. V. Application of redundant positional notations for increasing of arithmetic algorithms scalability // 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numeric (SCAN'2012, Novosibirsk, Russia, September 23-29, 2012): Book of abstracts. — Institute of Computational Technologies Publisher, 2012.

18. Панюков, А. В. Сложность нахождения гарантированной оценки решения приближенно заданной системы линейных алгебраических уравнений / А. В. Панюков, М. И. Германенко // Известия Челябинского научного центра УрО РАН. — 2000. — № 4. — С. 21–30.

19. Панюков, А. В. Реализация базовых операций целочисленной арифметики в гетерогенных системах. // Параллельные вычислительные технологии (ПаВТ'2012). – Труды международной научной конференции (Новосибирск, 26 марта – 30 марта 2012 г.). — Челябинск: Издательский центр ЮУрГУ., 2012. — С. 77–84.

20. Панюков, А. В. Применение массивно-параллельных вычислений для реализации основных операций целочисленной арифметики // Высокопроизводительные параллельные вычисления на кластерных системах (HPC - 2010). Материалы X Международной конференции (г. Пермь, 1 - 3 ноября 2010 г.). В 2-х томах. — Т. 2. — Пермь: Изд-во ПермГТУ, 2010. — С. 77–84.

21. Golodov, V. A. Distributed symbolic rational-fractional calculations on the processors of series of x86 and x64 // Proceeding of international conference "Parallel computational technologies"(Novosibirsk, 2012, on March 26 to 30). — Chelyabinsk: Publishing center of SUSU, 2012. — P. 774.

22. Панюков, А. В. Техника программной реализации алгоритма решения системы линейных алгебраических уравнений с интервальной неопределенностью в исходных данных // "Параллельные вычисления и задачи управления"PACO'2012. Шестая международная конференця, Москва, 24-26 октября. Труды в 3 т. — Т. 2. — Москва: ИПУ РАН, 2012. — С. 155–166.