УДК 519.688

О программной реализации алгоритма умножения Карацубы

М. Е. Куляс*

Предложен комбинированный рекурсивный алгоритм умножения длинных целых чисел, сочетающий асимптотически быстрый метод Карацубы и метод сдвигов и сложений на нижних уровнях рекурсии. Приведены варианты реализации алгоритма и результаты экспериментального исследования его эффективности.
Ключевые слова: умножение длинных чисел, алгоритм Карацубы, рекурсия, линейная программа, многопоточное программирование.

Умножение длинных целых чисел является одной из наиболее востребованных операций современной компьютерной алгебры. В связи с этим создаются и совершенствуются соответствующие программные библиотеки. Их эффективность часто зависит от скорости вычисления произведения длинных чисел. Возникает задача ускорения данной операции при ее программной реализации на современных вычислительных средствах.

Рассматривая алгоритмы умножения, остановимся на методе сдвигов и сложений [1], а также на асимптотически более быстром алгоритме умножения по методу Карацубы [1 — 4]. Такие методы являются наиболее эффективными при реализации операции умножения длинных целых чисел с диапазоном значений, представляющим наибольший практический интерес в наши дни. Существуют асимптотически еще более быстрые алгоритмы, основанные на быстром преобразовании Фурье, но они имеют большую мультипликативную константу в оценке сложности и показывают существенное преимущество по сравнению с методами сдвигов и сложений и методом Карацубы только для чисел в несколько десятков тысяч десятичных знаков.

* KuliasMY@mpei.ru

Операнды и результаты выполнения математических операций представим в позиционной системе по основанию 2^w в виде массивов целых, беззнаковых, w-битных чисел. Обычно параметр w выбирают равным размеру машинного слова для конкретного процессора. Число бит, необходимых для хранения длинных целых чисел, обозначим n, а число машинных слов, необходимых для хранения таких чисел, как $s = \lceil n/w \rceil$. Так, для представления 513-битного ($n = 513$) числа A в процессоре с разрядностью машинного слова $w = 64$ бит требуется массив из $s = \lceil 513/64 \rceil = 9$ элементов с индексами от 0 до $s - 1$, где $A[0]$ хранит младшие значащие биты числа A, а элемент $A[s - 1]$ — старшие. Диапазон возможных значений каждого элемента $A[i]$ от 0 до $2^w - 1$. Диапазон возможных значений произведений двух таких элементов от 0 до $2^{2w} - 2^{w+1} + 1$, т.е. для представления произведения двух элементов используются два машинных слова. Конкатенацию двух и более элементов обозначим с помощью круглых скобок с перечислением этих элементов. Например, для однобайтовых машинных слов ($w = 8$) запись $(U, V) = (2a3f)_{16}$ означает, что $U = 0x2a$ и $V = 0x3f$ (здесь и далее для представления шестнадцатеричных чисел возьмем общепринятое обозначение с префиксом 0x).

Основу метода сдвигов и сложений (алгоритм 1) составляют 2 вложенных цикла. Во внешнем цикле идет

перебор значений всех разрядов одного операнда, а во внутреннем цикле выполняется операция умножения с накоплением результата произведения текущего разряда одного операнда на все разряды второго операнда. Разрядности промежуточных результатов вычислений определяются операциями, указанными в пункте 2.2.1 алгоритма 1: $(U, V) \leftarrow P[i + j] + A[j]\cdot B[i] + U$. Здесь (U, V) означает конкатенацию двух w-битных чисел U и V. Максимальное значение (U, V) ограничено $2^{2w} - 1$, поскольку значения $A[j]$, $B[i]$, U и $P[i + j]$ не больше $2^w - 1$.

Алгоритм 1. Умножение длинных целых чисел методом сдвигов и сложений [1]

ВХОД: два массива с операндами A и B ($A, B \in [0, 2^{s\cdot w})$) размером s элементов каждый: $A = (A[s - 1], ..., A[0])$, $B = (B[s - 1], ..., B[0])$, где $A[i]$, $B[i]$ принимают значения от 0 до $2^w - 1$; $0 \le i \le s - 1$.

ВЫХОД: массив P из $2s$ элементов, содержащий результаты операции умножения: $P = A\cdot B = (P[2s - 1], \ldots, P[0])$.

1. Для всех i от 0 до $s - 1$ установить $P[i] \leftarrow 0$.
2. Для всех i от 0 до $s - 1$ выполнить:
   2.1. Установить $U \leftarrow 0$.
   2.2. Для всех j от 0 до $s-1$ выполнить:
   2.2.1. Установить $(U, V) \leftarrow P[i+j] + A[j]\cdot B[i] + U$.
   2.2.2. Установить $P[i+j] \leftarrow V$.
   2.3. Установить $P[i+s] \leftarrow U$.
3. Вернуть P.

Сложность алгоритма оценивается как $O(s^2)$, поскольку требуется выполнить не более s^2 инструкций умножения w-битных слов [1]. Для эффективной реализации этого метода необходимо процессорная инструкция быстрого умножения целых w-битных чисел с сохранением $2w$-битного результата. Так, в современных версиях языков C/C^{++} имеются расширенные целочисленные типы данных двойной точности (например, целочисленный беззнаковый тип uint128_t в 64-битной операционной системе), позволяющие хранить результат произведения двух машинных слов. В общем случае для реализации алгоритма требуется $2s^2$+s инструкций чтения из памяти в регистр и s^2+s инструкций записи из регистра в память. Если промежуточные значения частичных произведений $P[i+j]$ в пунктах 2.2.1 и 2.2.1 алгоритма 1 хранить не в ячейках памяти конечного результата, а использовать дополнительно s регистров, то общее число обращений к памяти снижается до s^2+3s.

Операция возведения в квадрат требует меньшего числа инструкций, поскольку частичные слагаемые $A[i]\cdot A[j]$ и $A[j]\cdot A[i]$ в алгоритме 1 могут быть заменены удвоенным значением $2A[i]\cdot A[j]$. Таким образом, общее число операций умножения процессорных слов для операции возведения в квадрат снижается до $(s^2+s)/2$.

При рассмотрении метода Карацубы будем полагать, что n (число бит, необходимых для хранения операндов) является степенью двойки, тогда любое n-битное целое число A можно записать по основанию $2^{n/2}$ как сумму двух $n/2$-битных чисел:

$$A = (A_1, A_0) = A_0 + A_1\cdot 2^{n/2}.$$

Тогда по формуле Карацубы [3] произведение двух таких чисел требует 3 операции умножения и несколько дополнительных операций сложения/вычитания, имеющих линейную сложность, и определяется следующим образом:

$$A\cdot B = A_0\cdot B_0\cdot(1 + 2^{n/2}) + (A_1 - A_0)\cdot(B_0 - B_1)\cdot 2^{n/2} + \\ + A_1\cdot B_1\cdot(2^n + 2^{n/2}). \quad (1)$$

В то же время по методу сдвигов и сложений требуется выполнить 4 операции умножения:

$$A\cdot B = A_0\cdot B_0 + (A_0\cdot B_1 + A_1\cdot B_0)\cdot 2^{n/2} + A_1\cdot B_1\cdot 2^n.$$

Формула Карацубы может применяться рекурсивно с глубиной рекурсии не более чем $\log_2(n) - \log_2(w)$ уровней. Рекурсивный вариант реализации данного метода для максимальной глубины вложенности описан в алгоритме 2. Базой рекурсии в данном случае является умножение двух процессорных слов с разрядностью w-бит каждое. Асимптотическая сложность алгоритма оценивается как $O(s^{\log 3}) \approx O(s^{1,585})$ операций умножения базовых слов [2, 3].

Алгоритм 2. Умножение длинных целых чисел методом Карацубы

ВХОД: два массива с операндами A и B ($A, B \in [0, 2^{s\cdot w})$) размером s элементов каждый: $A = (A[s - 1], ..., A[0])$; $B = (B[s - 1], ..., B[0])$, где $A[i]$, $B[i]$ принимают значения от 0 до 2^w-1; $0 \le i \le s - 1$.

ВЫХОД: массив P из $2s$ элементов, содержащий результаты операции умножения: $P = A\cdot B = (P[2s - 1], \ldots, P[0])$.

1. Если $s = 1$, то вернуть $A\cdot B$ (база рекурсии).
2. Установить $s \leftarrow s/2$.
3. Установить $A_0 \leftarrow (A[s - 1], \ldots, A[0])$;
   $A_1 \leftarrow (A[2s - 1], \ldots, A[s])$.
4. Установить $B_0 \leftarrow (B[s - 1], \ldots, B[0])$;
   $B_1 \leftarrow (B[2s-1], \ldots, B[s])$.
5. Вычислить $(A_0\cdot B_0)$, $(A_1\cdot B_1)$, $M = (A_1 - A_0)\cdot(B_0 - B_1)$ по пунктам 1 — 6 данного алгоритма.
6. Вернуть $A_0\cdot B_0\cdot(1 + 2^{n/2}) + M\cdot 2^{n/2} + A_1\cdot B_1\cdot(2^n + 2^{n/2})$.

В качестве примера рассмотрим умножение двух целых чисел $A = (\text{FE98})_{16} = 0\text{xFE98}$ и $B = (\text{DCBA})_{16} = 0\text{xDCBA}$ при $s = 2$, $w = 8$, $n = s\cdot w = 16$ методом Карацубы. Представим числа A и B по основанию $n/2$: $A = (A_1, A_0) = A_0 + A_1\cdot 2^{n/2} = 0\text{x98} + 0\text{xFE}\cdot 2^8$; $B = (B_1, B_0) = B_0 + B_1\cdot 2^{n/2} = 0\text{xBA} + 0\text{xDC}\cdot 2^8$. Вычислим произведения $(A_0\cdot B_0)$, $(A_1 - A_0)\cdot(B_0 - B_1)$, $(A_1\cdot B_1)$: $(A_0\cdot B_0) =$

= 0x6E70; $(A_1 \cdot B_1)$ = 0xDA48; $(A_1 - A_0)\cdot(B_0 - B_1)$ = = −(0x66)·(0x22) = −0x0D8C (при вычислении значения $B_0 - B_1$ необходимо учитывать возникающий заем из старшего разряда, который меняет операцию суммирования на вычитание). Суммируем частичные произведения с учетом знака компонента $(A_1 - A_0)\cdot(B_0 - B_1)$: $A\cdot B = A_0\cdot B_0 + A_1\cdot B_1\cdot 2^n + A_0\cdot B_0\cdot 2^{n/2} + (A_1 - A_0)\cdot(B_0 - B_1)\cdot 2^{n/2} +$ $+ A_1\cdot B_1\cdot 2^{n/2}$ = 0x6E70 + 0xDA480000 + 0x6E7000 − − 0x0D8C00 + 0xDA4800 = 0xDB839A70.

Возведение в квадрат рекурсивным методом Карацубы проводится аналогично, при условии, что из алгоритма 2 исключают шаг 4, а на шаге 5 при вычислении M всегда получается положительный результат, что также несколько снижает сложность всего алгоритма.

При практической реализации данного метода следует учитывать тот факт, что в пункте 5 алгоритма 2 компонент M может принимать как положительные, так и отрицательные значения. Таким образом, если $M > 0$, то при вычислении пункта 4 этот компонент входит со знаком «+», иначе со знаком «–», и операция суммирования заменяется на вычитание. Такое ветвление вносит большую задержку в вычисления даже на современных процессорах, поскольку из-за случайного характера знака компонента M аппаратные блоки предсказания переходов не могут эффективно определить нужную ветвь. В результате неправильного предсказания перехода конвейер процессора вынужден перезапуститься, а подсистеме кэш-памяти следует загрузить необходимый код команд и данных. Кроме того, для программных приложений в области криптографии критически важно избегать таких ветвлений, так как они могут привести к утечке секретной информации вследствие разности времен обработки различных ветвей.

Чтобы избежать ветвления при вычислении M, можно вместо определения разностей $(A_1 - A_0)$ и $(B_0 - B_1)$ вычислять их абсолютные разности $|A_1 - A_0|$ и $|B_0 - B_1|$ с одновременным определением бита знака (t) компонента M. Такая операция реализуется следующим образом:

вычисляются s разрядов разности $(A_1 - A_0)$ и бит заема из старшего разряда (t_a);

из значения t_a формируется w-битная маска $mask_a = - t_a$;

выполняется операция «исключающее или» со всеми разрядами разности $(A_1 - A_0)$ и маской $mask_a$; если $t_a = 0$ и $A_1 \geq A_0$, то операция «исключающее или» не меняет значения разности $(A_1 - A_0)$; если же $t_a = 0$, то все биты маски устанавливаются в «1», и в результате такой операции получается значение $|A_1 - A_0|$, записанное в форме дополнения до единицы;

чтобы иметь представление $|A_1 - A_0|$ в форме дополнения до двойки, необходимо к нему прибавить бит заема t_a с корректной обработкой возникающих при этом переносов.

Аналогичным образом вычисляются значения $|B_0 - B_1|$ и t_b. Результирующий знак произведения $M_{abs} = |A_1 - A_0|\cdot|B_0 - B_1|$ определяется по значениям t_a и t_b: $t = t_a \wedge t_b$.

Для иллюстрации описанного метода можно привести следующий пример: положим, A_1 = 0x2, A_0 = 0x7, B_0 =0x7, B_1 = 0x2, s = 1, w = 4, тогда: $(A_1 - A_0)$ = 0xB, t_a = 1, $mask_a$ = 0xF, $(A_1 - A_0) \wedge mask_a$ = 0x4, $|A_1 - A_0|$ = = $(A_1 - A_0) \wedge mask_a + t_a$ = 0x5; $(B_0 - B_1)$ = 0x5, t_b = 0, $mask_b$ = 0x0, $(B_0 - B_1) \wedge mask_b$ = 0x0, $|B_0 - B_1|$ = = $(B_0 - B_1) \wedge mask_b + t_a$ = 0x5; $t = t_a \wedge t_b$ = 0x1.

Несколько меняется и пункт 6 алгоритма 2: требуется выполнить операцию обращения знака слагаемого M_{abs} при t = 1 и вернуть значение $A_0\cdot B_0\cdot(1 + 2^{n/2}) +$ $+ (-1)^t M_{abs}\cdot 2^{n/2} + A_1\cdot B_1\cdot(2^n + 2^{n/2})$.

Более детальный анализ формулы (1) позволяет дополнительно сократить число операций сложения, т.к. некоторые слагаемые в ней вычисляются дважды [3]. Представим слагаемые $(A_0\cdot B_0)$ и $(A_1\cdot B_1)$ по основанию $n/2$ и введем соответствующие обозначения:

$$(A_0\cdot B_0) = L_0 + 2^{n/2}L_1;\ (A_1\cdot B_1) = H_0 + 2^{n/2}H_1;\ T = L_1 + H_0,$$

тогда

$$\begin{aligned} A\cdot B &= (L_0 + 2^{n/2}L_1)\cdot(1 + 2^{n/2}) + (-1)^t M_{abs}\cdot 2^{n/2} + \\ &+ (H_0 + 2^{n/2}H_1)\cdot(2^n + 2^{n/2}) = 2^{3n/2}H_1 + 2^n(L_1 + H_0 + H_1) + \\ &+ 2^{n/2}(L_1 + H_0 + L_0 + (-1)^t M_{abs}) + L_0 = 2^{3n/2}H_1 + \\ &+ 2^n(T + H_1) + 2^{n/2}(T + L_0 + (-1)^t M_{abs}) + L_0. \end{aligned} \quad (2)$$

Таким образом, заранее вычисленное значение T позволяет сократить число операций сложения на величину $2^{n/2}/w$.

Следует заметить, что существует вариант формулы Карацубы, в котором знаки среднего члена всегда определены [4]:

$$A\cdot B = A_0\cdot B_0\cdot(1 - 2^{n/2}) + (A_1 + A_0)\cdot(B_0 + B_1)\cdot 2^{n/2} + A_1\cdot B_1\cdot(2^n - 2^{n/2}).$$

При практической реализации вычислений по этой формуле требуется дополнительный объем памяти (по сравнению с рассмотренным выше вариантом), поскольку при вычислении значений $(A_1 + A_0)$ и $(B_0 + B_1)$ требуется уже $(n/2 + 1)$ разряд. Соответственно увеличивается время вычисления произведения $(A_1 + A_0)\cdot(B_0 + B_1)$ и размер памяти, необходимый для хранения результата этой операции. Кроме того, нарушается условие применимости рекурсии, т.к. размер операндов n уже не является степенью двойки (данное ограничение введено ранее). Вследствие этого усложняется рекурсивная реализация комбинированного метода, что является предметом дополнительного исследования, выходящего за рамки настоящей работы.

Следующим подходом к повышению быстродействия вычисления произведения длинных целых чисел может служить синтез комбинированного рекурсив-

ного алгоритма, сочетающего метод Карацубы по (2) и метод сдвигов и сложений на нижних уровнях. В комбинированном алгоритме на этапе прямого хода рекурсии происходит понижение разрядностей сомножителей до некоторой величины n_0, называемой порогом рекурсии. По достижению этого порога разрядности сомножителей равны n_0, и их произведение можно вычислить по методу сдвигов и сложений. Такой подход позволяет уменьшить мультипликативную константу в оценке сложности алгоритма Карацубы для некоторого диапазона значений входных данных.

Алгоритм 3. Комбинированный рекурсивный метод умножения

ВХОД: два массива с операндами A и B ($A, B \in [0, 2^{s \cdot w})$) размером s элементов каждый: $A = (A[s-1], ..., A[0])$; $B = (B[s-1], ..., B[0])$, где $A[i]$, $B[i]$ принимают значения от 0 до $2^w - 1$; $0 \le i \le s-1$; порог рекурсии n_0, массив P из $6s$ элементов разрядности w.

ВЫХОД: массив P из $2s$ элементов, содержащий результаты операции умножения: $P = A \cdot B = (P[2s-1], \ldots, P[0])$.

1. Если $s \le n_0$, то вернуть вычисленное методом сдвигов и сложений значение $(A \cdot B)$.

2. Установить $s \leftarrow s/2$.

3. Установить $A_0 \leftarrow (A[s-1], \ldots, A[0])$;
$A_1 \leftarrow (A[2s-1], \ldots, A[s])$.

4. Установить $B_0 \leftarrow (B[s-1], \ldots, B[0])$;
$B_1 \leftarrow (B[2s-1], \ldots, B[s])$.

5. Вычислить абсолютные разности $|A_1 - A_0|$, $|B_0 - B_1|$ и их знаки t_a, t_b.

6. Вычислить $(A_0 \cdot B_0)$, $(A_1 \cdot B_1)$ по пунктам 1 — 10 данного алгоритма.

7. Вычислить $M_{abs} = |A_1 - A_0| \cdot |B_0 - B_1|$ по пунктам 1 — 10 настоящего алгоритма; $t = t_a \wedge t_b$.

8. Вычислить $(-1)^t M_{abs}$.

9. Вычислить $T = L_1 + H_0$ при $(A_0 \cdot B_0) = L_0 + 2^{n/2} L_1$ и $(A_1 \cdot B_1) = H_0 + 2^{n/2} H_1$.

10. Вернуть $2^{3n/2} H_1 + 2^n (T + H_1) + 2^{n/2} (T + L_0 + (-1)^t M_{abs}) + L_0$.

Для корректной работы предложенного алгоритма требуется выделить массив P из $6s$ элементов с разрядностью w-бит каждый. Младшие $2s$ элементов массива P предназначены для хранения результатов операции умножения, а оставшиеся $4s$ элементов — для промежуточных вычислений и организации вложенных уровней рекурсии (рис. 1). Значение T предлагается записывать в массив P, начиная с адреса $3s$.

На каждом уровне рекурсии произведения $(A_0 \cdot B_0)$, $(A_1 \cdot B_1)$ и M_{abs} занимают по s элементов каждое и записываются с нулевым смещением относительно текущего базового адреса массива P. Абсолютные разности $|A_1 - A_0|$ и $|B_0 - B_1|$ занимают по $s/2$ элементов каждая и записываются в конец массива P, как показано на рис. 1. Следующий по вложенности уровень рекурсии требует $6s/2$ элементов P, причем вычисленные на верхнем уровне значения $|A_1 - A_0|$ и $|B_0 - B_1|$ становятся исходными данными для рекурсивного вычисления M_{abs}.

При практической реализации алгоритмов умножения для увеличения скорости вычислений эффективным будет построение программы в виде последовательности элементарных операций (присваивание, умножение, сложение машинных слов и т.д.), без использования циклов, ветвлений и рекурсии [5]. В этом случае декомпозиционная схема вычислений может строиться в автоматическом режиме. Для этого вместо непосредственных вычислений по пунктам алгоритма требуется выполнять последовательную запись соответствующих элементарных операций над текущими элементами массивов A, B и P. Ее результатом станет линейная программа, которая для некоторых размеров разрядностей входных операндов будет работать быстрее итеративного или рекурсивного аналога, т.к. в ней отсутствуют накладные расходы на организацию циклов и рекурсии.

Еще одним методом программирования, позволяющим увеличить скорость вычислений по описанным алгоритмам, является создание многопоточной программы. Современные средства и языки программирования позволяют создавать многопоточные приложения, существенно увеличивающие скорость вычислений на многоядерных процессорах с общей памятью [6]. Одним из способов разработки подобных приложений является метод декомпозиции задач. Так, анализ информационной зависимости данных и операций в (2) позволяет построить алгоритм умножения, в котором независимые промежуточные вычисления выполняются в несколько потоков. В частности промежуточные слагаемые $(A_0 \cdot B_0)$, $(A_1 \cdot B_1)$ и M_{abs} в (2) могут быть вычислены параллельными потоками. Существуют некоторые ограничения на число используемых программных потоков. Они вытекают как из анализа информационной зависимости данных и операций, так и из величины накладных расходов, появляющихся в результате создания, синхронизации и взаимодействия нескольких потоков. Одним из методов создания многопоточных программ является использование

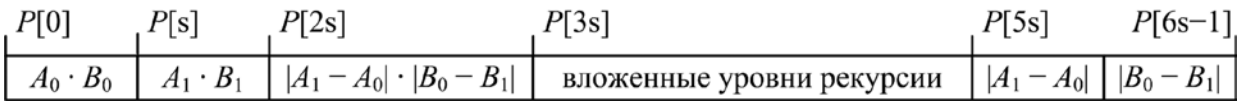

Рис. 1. Выделение памяти для рекурсивного алгоритма Карацубы

технологии OpenMP [6]. Данная технология позволяет создавать многопоточные приложения путем добавления в исходный код специальных директив, явно указывающих компилятору, какие части исходного кода программы могут распределяться между потоками и вычисляться одновременно на многоядерной вычислительной системе. При запуске такого приложения создается один главный мастер-поток, который далее запускает, останавливает и синхронизирует подчиненные потоки, задачи в которых могут выполняться одновременно.

Для определения критерия применимости метода сдвигов и сложений в комбирированном методе умножения на тестовом стенде (Intel Core i7-2600 3,4 ГГц, Linux 3.0 64-bit, gcc 4.5.4-O2-m64) при фиксированных значенях параметров n и w проводились замеры среднего числа тактов процессора при различных значениях параметра порога остановки рекурсии n_0. В результате экспериментов выяснилось, что оптимальным значением n_0 является величина, равная $16w$. Так, при $w = 32$, $n = 4096$, $n_0 = 512$ комбинированный метод показывает наилучшие результаты по скорости вычислений (рис. 2). Аналогичная зависимость наблюдалась и для $w = 64$ (рис. 3).

Как следует из рис. 2, 3, при достижении порога остановки рекурсии значения $n_0 = 16w$ число процессорных инструкций минимальное для данной реализации, и при дальнейшем увеличении глубины рекурсии заметен их значительный рост. Это свидетельствует о том, что рекурсивная имплементация формулы Карацубы до максимальной глубины рекурсии (когда базой рекурсии является умножение двух процессорных слов) малоэффективна для рассматриваемого диапазона значений операндов.

Для оценки эффективности различных вариантов реализации рассматриваемых методов замерялось среднее число тактов процессора при выполнении подпрограмм умножения больших целых чисел различной разрядности (n от 128 до 8192 бит) для $w = 32$ (базовый тип представления данных uint32_t) и $w = 64$ (базовый тип представления данных uint64_t). Для метода сдвигов и сложений и метода Карацубы (2) были созданы как линейные версии программ (без использования циклов и ветвлений), так и их итеративные (метод сдвигов

Рис. 2. Среднее число тактов процессора при вычислении произведения целых чисел разрядности 4096 бит при $w = 32$ бит для различных значений порога остановки рекурсии n_0 по алгоритму 3

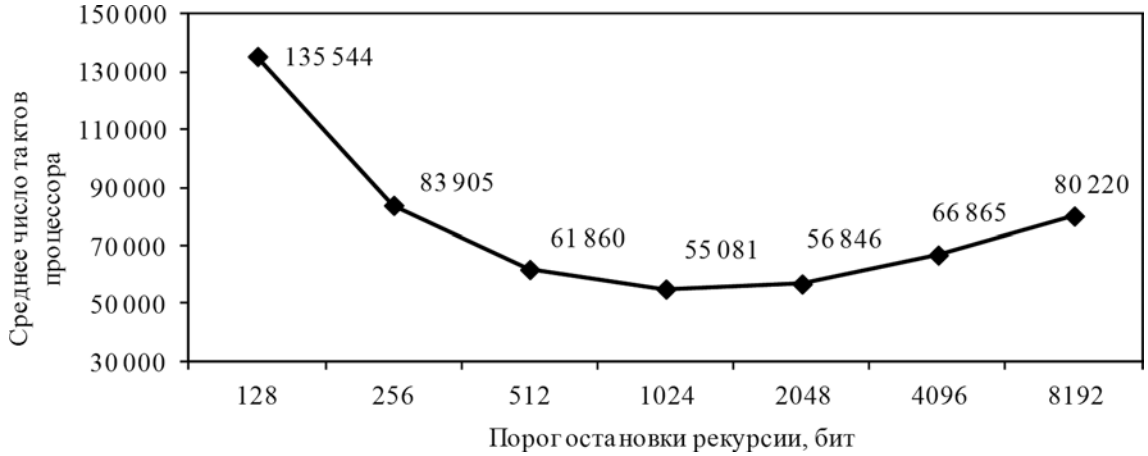

Рис. 3. Среднее число тактов процессора при вычислении произведения целых чисел разрядности 8192 бит при $w = 64$ бит для различных значений порога остановки рекурсии n_0 по алгоритму 3

и сложений) и рекурсивные (метод Карацубы) аналоги. Результаты экспериментов по определению критериев применимости методов показаны на рис. 4, 5.

Кроме того, на языке Си с использованием технологии OpenMP была создана многопоточная версия программы вычисления умножения длинных целых чисел с применением рекурсивного метода Карацубы и порогом рекурсии $n_0 = n/2$, без использования предвычисленного значения T. Для вычисления частичных слагаемых $(A_0 \cdot B_0)$, $(A_1 \cdot B_1)$ и M_{abs} использовалось три потока. Поскольку имеются значительные расходы на создание, взаимодействие и синхронизацию потоков, дальнейшее увеличение их числа менее эффективно для рассматриваемого диапазона значений разрядностей входных операндов.

Результаты экспериментов показали, что при небольших размерах операндов наиболее эффективной является линейная реализация метода сдвигов и сложений (для $w = 32$). Начиная с $n = 512$ бит более эффективной становится линейная реализация метода Карацубы. При $n = 2048$ лидирует рекурсивная реализация метода Карацубы, поскольку размер линейной программы при этом превышает размер кэша процессора и возникают дополнительные накладные расходы на его обновление. Многопоточная версия метода Карацубы показала свою эффективность только при $n = 4096$ и более. Это связано с тем, что накладные расходы на организацию и взаимодействие трех потоков требуют более 7000 тактов процессора. Интересным является тот факт, что линейная реализация метода Карацубы остается эффективнее своей рекурсивной реализации вплоть до $n = 1024$ бит включительно, в то время как линейная реализация метода сдвигов и сложений уже

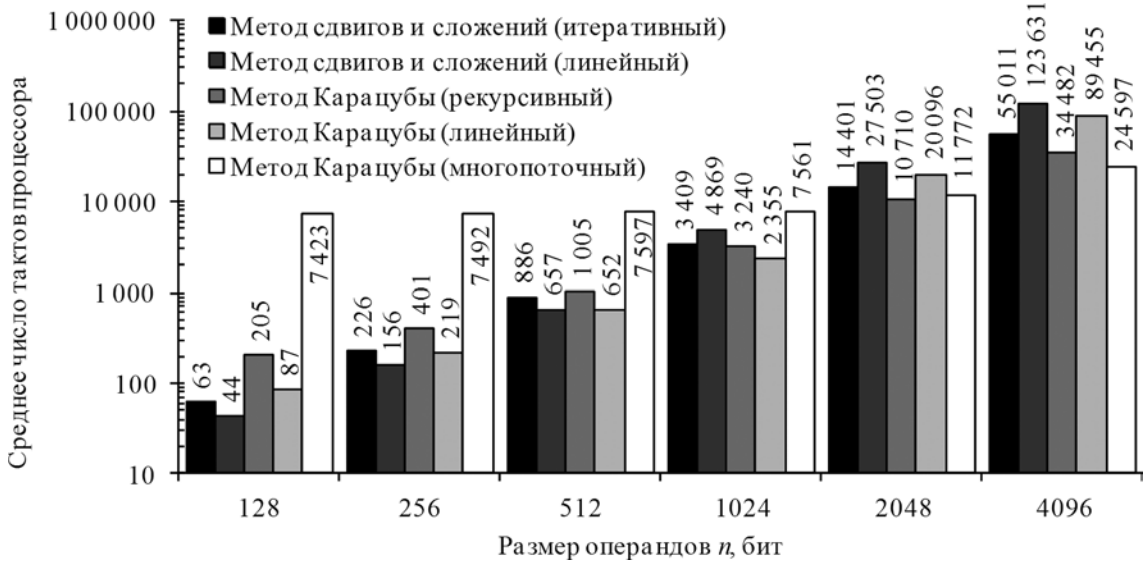

Рис. 4. Среднее число тактов процессора при вычислении произведения целых чисел различной разрядности для $w = 32$ бит

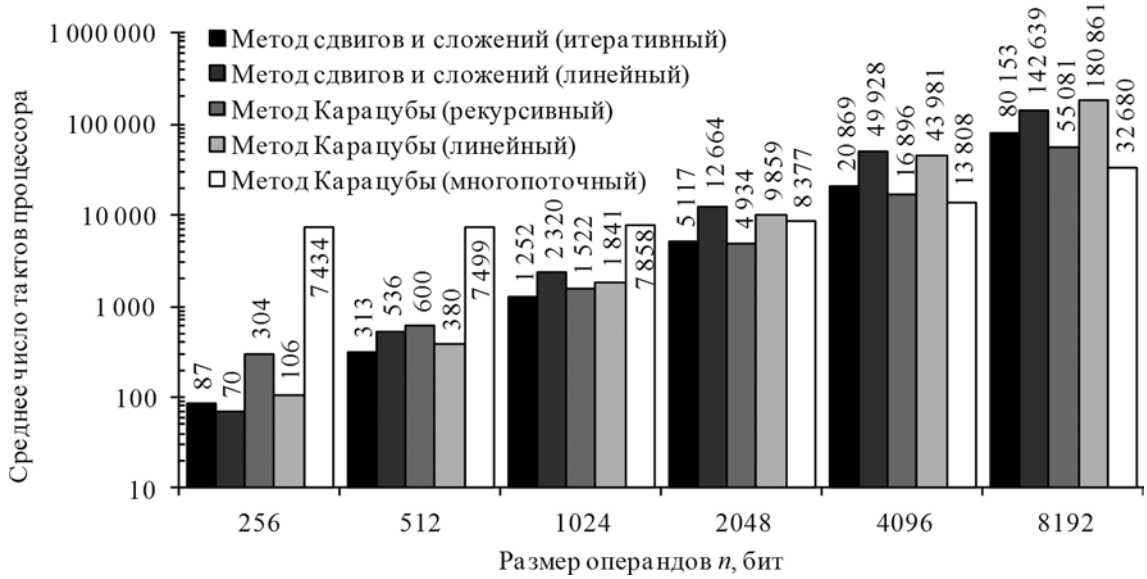

Рис. 5. Среднее число тактов процессора при вычислении произведения целых чисел различной разрядности для $w = 64$ бит

менее эффективна, чем итеративная при $n = 1024$ бит, что также объясняется соотношением размера программы и кэша процессора. Аналогичные выводы можно сделать и при $w = 64$ бит.

Таким образом, предложенные варианты алгоритмов умножения могут быть эффективно использованы при создании программных библиотек компьютерной алгебры. Проведенные эксперименты позволили определить критерии применимости рассматриваемых методов в зависимости от размера операндов. Было показано влияние порога остановки прямого хода рекурсии в предложенном комбинированном алгоритме на скорость вычислений. Определены оптимальные значения этого параметра.

Дальнейшим направлением исследований в области эффективной реализации алгоритмов умножения больших целых чисел может служить анализ рассмотренных алгоритмов с учетом применения векторных инструкций (расширения SSE и AVX для Intel и NEON для процессоров ARM). Такие расширения позволяют распараллеливать однотипные операции и значительно увеличивают скорость вычислений, но при этом сильно ограничивают переносимость кода на различные вычислительные средства.

Литература

1. **Кнут Д.** Искусство программирования. Получисленные алгоритмы. Т. 2. М.: Вильямс, 2004.

2. **Карацуба А.А., Офман Ю.П.** Умножение многозначных чисел на автоматах // ДАН СССР. 1962. № 2. Т. 145. С. 293 — 294.

3. **Гашков С.Б.** Занимательная компьютерная арифметика. Быстрые алгоритмы операций с числами и многочленами. М.: УРСС, Либроком, 2012.

4. **Карацуба А.А.** Сложность вычислений // Труды математического института имени В. А. Стеклова. 1995. № 211. С. 186 — 202,

5. **Фролов А.Б., Винников А.М.** О машинном синтезе некоторых линейных программ // Программная инженерия. 2011. № 6. С. 24 — 30.

6. **Эхтер Ш.** Многоядерное программирование. СПб.: Питер, 2010.

Статья поступила в редакцию 05.11.2015