УДК 004.051/056
Ковтун В.Ю., Охрименко А.А. (НАУ)

ПОДХОДЫ К ПОВЫШЕНИЮ ПРОИЗВОДИТЕЛЬНОСТИ ПРОГРАММНОЙ РЕАЛИЗАЦИИ ОПЕРАЦИИ УМНОЖЕНИЯ В ПОЛЕ ЦЕЛЫХ ЧИСЕЛ

Авторами предлагается подход к увеличению производительности программной реализации алгоритма умножения в поле чисел для 32-х и 64-х разрядных платформ, который состоит в использовании механизма отложенного учета переноса из старшего разряда при накоплении суммы, что позволяет избежать необходимости учета переноса из старшего разряда на каждой итерации цикла накопления суммы. Отложенный перенос дает возможность уменьшить общее число операций сложения и эффективно применять существующие технологии распараллеливания.

Ключевые слова: умножение целых чисел, программная реализация, криптографические преобразования, криптосистема, поле целых чисел, распараллеливание.

Введение. Криптографические преобразования с открытым ключом прошли долгий путь с момента их предложения Диффи и Хеллманом [1] до современных криптосистем на алгебраических кривых, однако неизменным в них оставалось одно – операции в поле целых чисел $\mathbf{GF}(p)$, среди которых особое место занимает операция умножения (рис. 1). Среди актуальных задач дальнейшего развития криптосистем с открытым ключом, фигурирует и повышение производительности их программной и аппаратной реализации. Одним из подходов к увеличению быстродействия криптосистем, является увеличение быстродействия арифметических преобразований в поле чисел – операции умножения.

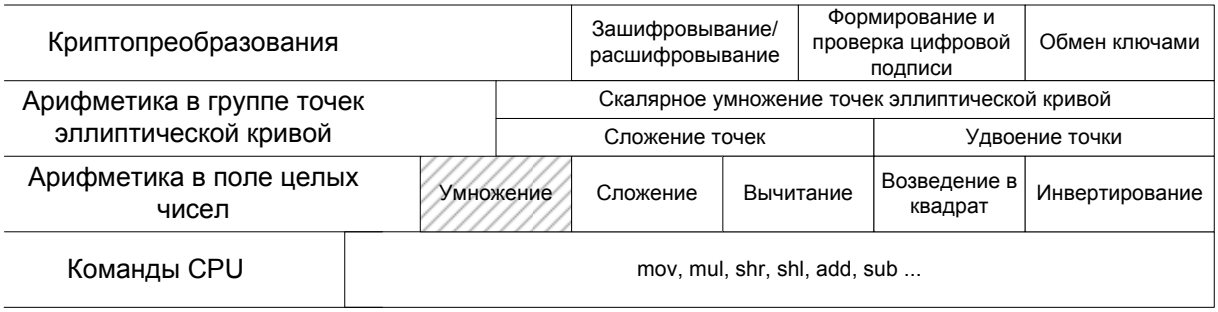

Рис. 1. Иерархия операций в криптосистеме на эллиптической кривой

Заметим, что вопрос повышения производительности арифметических операций над целыми числами в поле активно изучался многими учеными, о чем свидетельствует значительное число публикаций по данному направлению [2-8]. Кроме самих алгоритмов арифметических операций, интерес представляют подходы к архитектуре самих библиотек [9-18], которые позволяют существенно сократить накладные расходы на реализацию операций над числами в общем.

Проведенный анализ публикаций [2-8], позволил выделить наиболее эффективные алгоритмы умножения Comba [2, 3] и Карацубы [3, 8, 10]. Однако алгоритм Comba показывает лучшие результаты тестов производительности программной реализации на современных платформах [3-9]. В работе [8] рассматривается алгоритм Карацубы-Comba – интересная модификация алгоритма Comba для RISC-процессоров, использующего алгоритм Карацубы лишь для умножения машинных слов. В связи с этим, **целью работы** является предложение подходов к повышению эффективности программной реализации операции умножения чисел (возведения в квадрат) в поле $\mathbf{GF}(p)$, уже хорошо известного алгоритма Comba [2, 3, 8]. Кроме всего прочего, подобные исследования вызваны необходимостью подтверждения эффективности программных реализаций известных алгоритмов при непрерывном развитии современных 32-х и 64-х разрядных аппаратных платформ. Отметим, что в последнее десятилетие наблюдается развитие в сторону многоядерости процессоров и многопроцессорности вычислительных систем [8, 9].

Описание алгоритма-прототипа умножения и его модификация. Основу алгоритма Comba [2, 3, 8] составляет цикл п.2 и вложенный цикл 2.1. На низшем уровне иерархии, в цикле п. 2.1 выполняется умножения $(uv)^{(64)}$, результат является 64-х разрядным целым, которое затем разделяется на два 32-х разрядных $u^{(32)}$ и $v^{(32)}$. Накопление суммы производится в 32-х разрядных временных переменных r_0, r_1 и r_2, на каждой итерации п. 2.1..2 и п. 2.1.3. Присвоение конечного результата, а также изменение аккумуляторов суммы r_0, r_1 и r_2, происходит на каждой итерации п. 2.2.

Алгоритм 1. Умножение целых Comba

Вход: целое $a, b \in \mathbf{GF}(p)$, $w = 32$, $n = \log_{2^w} a$, $nk = 2n - 1$.

Выход: $c = a \cdot b$

1. $r_0^{(32)} \leftarrow 0$, $r_1^{(32)} \leftarrow 0$, $r_2^{(32)} \leftarrow 0$.
2. For $k \leftarrow 0$, $k < n$, $k{+}{+}$ do
2.1. For $i \leftarrow 0$, $j \leftarrow k$, $i \le k$, $i{+}{+}$, $j{-}{-}$ do
2.1.1. $(uv)^{(64)} \leftarrow a_i^{(32)} \cdot b_j^{(32)}$.
2.1.2. $r_0^{(32)} \leftarrow r_0^{(32)} + v^{(32)}$, $r_1^{(32)} \leftarrow r_1^{(32)} + u^{(32)} + carry$, $carry \leftarrow 0$.
2.1.3. $r_2^{(32)} \leftarrow r_2^{(32)} + carry$, $carry \leftarrow 0$.
2.2. $c_k^{(32)} \leftarrow r_0^{(32)}$, $r_0^{(32)} \leftarrow r_1^{(32)}$, $r_1^{(32)} \leftarrow r_2^{(32)}$, $r_2^{(32)} \leftarrow 0$.
3. For $k \leftarrow n$, $l \leftarrow 1, k < nk$, $k{+}{+}$, $l{+}{+}$ do
3.1. For $i \leftarrow l$, $j \leftarrow k - l$, $i < n$, $i{+}{+}$, $j{-}{-}$ do
3.1.1. $(uv)^{(64)} \leftarrow a_i^{(32)} \cdot b_j^{(32)}$.
3.1.2. $r_0^{(32)} \leftarrow r_0^{(32)} + v^{(32)}$, $r_1^{(32)} \leftarrow r_1^{(32)} + u^{(32)} + carry$, $carry \leftarrow 0$.
3.1.3. $r_2^{(32)} \leftarrow r_2^{(32)} + carry$, $carry \leftarrow 0$.
3.2. $c_k^{(32)} \leftarrow r_0^{(32)}$, $r_0^{(32)} \leftarrow r_1^{(32)}$, $r_1^{(32)} \leftarrow r_2^{(32)}$, $r_2^{(32)} \leftarrow 0$.
4. $c_{nk}^{(32)} \leftarrow r_0^{(32)}$.
5. Return (c).

Рассмотрим основные недостатки алгоритма 1:

- Во вложенных циклах п. 2.1 и п. 3.1 происходит накопление суммы с переносом в 32-х разрядных временных переменных r_0, r_1 и r_2, п. 2.1.2, 2.1.3 и п. 3.1.2, 3.1.3:

  2.1.2. $r_0^{(32)} \leftarrow r_0^{(32)} + v^{(32)}$, $r_1^{(32)} \leftarrow r_1^{(32)} + u^{(32)} + carry$, $carry \leftarrow 0$.

  2.1.3. $r_2^{(32)} \leftarrow r_2^{(32)} + carry$, $carry \leftarrow 0$.

  Отметим, что в случае успешной проверки, происходит 3 операции сложения 32-х разрядных целых (две из них с учетом переноса), 3 присвоения 32-х разрядных переменных r_0, r_1 и r_2. Накопление суммы с учетом переноса производится на каждой итерации цикла 2.1.
- Во вложенных циклах п. 2.1 и п. 3.1 при накоплении суммы учитываются переносы между r_0, r_1 и r_2, используя вставки на языке ассемблера, что в свою очередь не позволяет выполнять спаривание и распараллеливание таких операций [11], как следствие – неэффективное использование ресурсов процессора.
- Высокая внутренняя связность, в связи с учетом переносов, не дает возможности эффективно распараллелить циклы п. 2 и п. 3.

- Не учитывается возможность использования современными процессорами поддержки 64-х разрядных операций.

Не сложно получить вычислительную сложность алгоритма Comba:

$$I_{mul}^{Comba} = 4I_{assign}^{32} + \left(\frac{n+1}{2}n + \frac{1+n-1}{2}(n-1)\right)\left(1I_{mul}^{32} + 3I_{add}^{32} + 6I_{assign}^{32}\right) + 4(2n-1)I_{assign}^{32} =$$
$$= 4I_{assign}^{32} + n^2\left(1I_{mul}^{32} + 3I_{add}^{32} + 6I_{assign}^{32}\right) + 4(2n-1)I_{assign}^{32},$$

где I_{assign}^{32} - операция присвоения 32-х разрядных чисел, I_{add}^{32} - операция сложения 32-х разрядных чисел, I_{mul}^{32} - операция умножения 32-х разрядных чисел, I_{amp}^{32} - операция сравнения 32-х разрядных чисел.

На рис. 2 проиллюстрируем, для $n=3$, недостатки алгоритма 1 и их влияние на вычислительную сложность алгоритма.

В верхней части указаны два больших числа a и b, представленные тремя 32-х разрядными целыми $a=(a_2, a_1, a_0)$ и $b=(b_2, b_1, b_0)$, где a_i и b_i имеют размер машинного слова. Под верхней чертой указаны итерации алгоритма. Обратим внимание, что алгоритм 1 реализует подход известный со школы – «умножение в столбик», с небольшим отличием: умножается часть множителя a_i $i=\overline{1,n}$ на все части другого множителя b_j $j=\overline{1,n}$, в порядке выполнения условия $(i+j==k)$ (по столбцам), а не последовательно части множителя a_i $i=\overline{1,n}$ на все части другого множителя b_j $j=\overline{1,n}$ (по строкам).

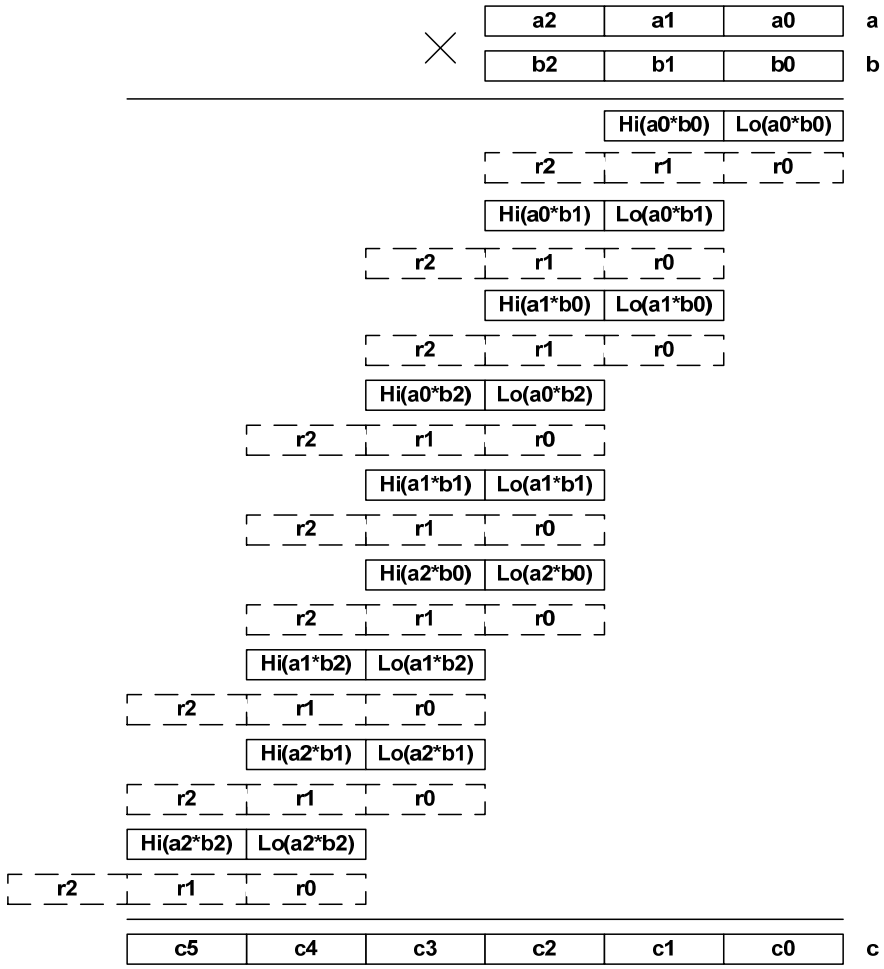

Рис. 2. Графическая интерпретация алгоритма 1

Такой подход приводит не к сложению строк (промежуточных результатов умножения), как в «умножении в столбик», а к сложению всех промежуточных результатов по столбцам, что позволяет сразу получать часть результирующего произведения c_i (под нижней чертой). Из рис. 2 видно, что после каждого умножения следует накопление суммы, с учетом переноса.

Для $n=3$, вычислительная сложность составит: $I_{mul}^{Comba}=4I_{assign}^{32}+9\left(1I_{mul}^{32}+3I_{add}^{32}+6I_{assign}^{32}\right)+$ $+20I_{assign}^{32}=78I_{assign}^{32}+9I_{mul}^{32}+27I_{add}^{32}$.

Рассмотрим теперь подходы, предложенные авторами, направленные на устранение указанных недостатков:

- Современные 32-х разрядные процессоры эффективно реализуют операции сложения 32-х и 64-х разрядных целых чисел, используя 64-х либо 32-х разрядные команды и переменные. Это позволяет реализовать накопление переноса в результате сложения 32-х разрядных значений в 64-х разрядной переменной, что избавит от необходимости после каждого сложения с переменными r_0, r_1 и r_2, выполнять учет и корректировку переноса. Накопленный перенос будет учитываться на финальных итерациях цикла п. 2 и п. 3.
- Современные процессоры обладают многоядерной архитектурой, что позволяет им параллельно выполнять несколько потоков команд. Это позволяет выполнить итерации циклов п. 2 и п. 3 параллельно используя реализацию стандарта OpenMP [11-13].

Введем следующие обозначения: через $t^{(64)}$ будем обозначать 64-х разрядные переменные, а через $t^{(32)}$ обозначим 32-х разрядные; операция $\mathrm{hi}_{(32)}\left(t^{(64)}\right)$ позволяет выделить старшие 32 разряда у 64-х разрядной переменной, а $\mathrm{low}_{(32)}\left(t^{(64)}\right)$ позволяет выделить младшие 32 разряда у 64-х разрядной переменной.

Алгоритм 2. Модифицированное умножение целых Comba

Вход: целое $a,b\in\mathbf{GF}(p)$, $w=32$, $n=\log_{2^w}a$, $nk=2n-1$.

Выход: $c=a\cdot b$

1. $r_0^{(64)}\leftarrow 0$, $r_1^{(64)}\leftarrow 0$, $r_2^{(64)}\leftarrow 0$.

2. For $k\leftarrow 0$, $k<n$, $k++$ do

2.1. For $i\leftarrow 0$, $j\leftarrow k$, $i\le k$, $i++$, $j--$ do

2.1.1. $(uv)^{(64)}\leftarrow a_i^{(32)}\cdot b_j^{32}$.

2.1.2. $r_0^{(64)}\leftarrow r_0^{(64)}+v^{(32)}$, $r_1^{(64)}\leftarrow r_1^{(64)}+u^{(32)}$.

2.2. $r_1^{(64)}\leftarrow r_1^{(64)}+\mathrm{hi}_{(32)}\left(r_0^{(64)}\right)$, $r_2^{(64)}\leftarrow r_2^{(64)}+\mathrm{hi}_{(32)}\left(r_1^{(64)}\right)$.

2.3. $c_k^{(32)}\leftarrow \mathrm{low}_{(32)}\left(r_0^{(32)}\right)$, $r_0^{(64)}\leftarrow \mathrm{low}_{(32)}\left(r_1^{(32)}\right)$, $r_1^{(64)}\leftarrow \mathrm{low}_{(32)}\left(r_2^{(32)}\right)$, $r_2^{(64)}\leftarrow 0$.

3. For $k\leftarrow n$, $l\leftarrow 1$, $k<nk$, $k++$, $l++$ do

3.1. For $i\leftarrow l$, $j\leftarrow k-l$, $i<n$, $i++$, $j--$ do

3.1.1. $(uv)^{(64)}\leftarrow a_i^{(32)}\cdot b_j^{32}$.

3.1.2. $r_0^{(64)}\leftarrow r_0^{(64)}+v^{(32)}$, $r_1^{(64)}\leftarrow r_1^{(64)}+u^{(32)}$.

3.2. $r_1^{(64)}\leftarrow r_1^{(64)}+\mathrm{hi}_{(32)}\left(r_0^{(64)}\right)$, $r_2^{(64)}\leftarrow r_2^{(64)}+\mathrm{hi}_{(32)}\left(r_1^{(64)}\right)$.

3.3. $c_k^{(32)}\leftarrow \mathrm{low}_{(32)}\left(r_0^{(32)}\right)$, $r_0^{(64)}\leftarrow \mathrm{low}_{(32)}\left(r_1^{(32)}\right)$, $r_1^{(64)}\leftarrow \mathrm{low}_{(32)}\left(r_2^{(32)}\right)$, $r_2^{(64)}\leftarrow 0$.

4. $c_{nk}^{(32)}\leftarrow \mathrm{low}_{(32)}\left(r_0^{(32)}\right)$.

5. Return (c).

Не сложно получить вычислительную сложность модифицированного алгоритма Comba:

$$I_{mul}^{Mod.\,Comba} = 4I_{assign}^{64} + \left(\tfrac{n+1}{2}n + \tfrac{1+n-1}{2}(n-1)\right)\left(1I_{mul}^{32} + 2I_{add}^{64|32} + 2I_{assign}^{64}\right) + (2n-1)\left(2I_{add}^{64|32} + 1I_{assign}^{64} + 1I_{assign}^{32}\right) =$$

$$= 4I_{assign}^{64} + n^2\left(1I_{mul}^{32} + 2I_{add}^{64|32} + 2I_{assign}^{64}\right) + (2n-1)\left(2I_{add}^{64|32} + 1I_{assign}^{64} + 1I_{assign}^{32}\right),$$

где I_{assign}^{32} - операция присвоения 32-х разрядных чисел, I_{assign}^{64} - операция присвоения 64-х разрядных чисел, I_{add}^{32} - операция сложения 32-х разрядных чисел, $I_{add}^{64|32}$ - операция сложения 32-х и 64-х разрядных чисел, I_{mul}^{32} - операция умножения 32-х разрядных чисел.

На рис. 3, 4 проиллюстрируем алгоритм 2 при $n = 3$. Вычислительная сложность для $n = 3$ составит: $I_{mul}^{Mod.\,Comba} = 27I_{assign}^{64} + 9I_{mul}^{32} + 28I_{add}^{64|32} + 5I_{assign}^{32}$.

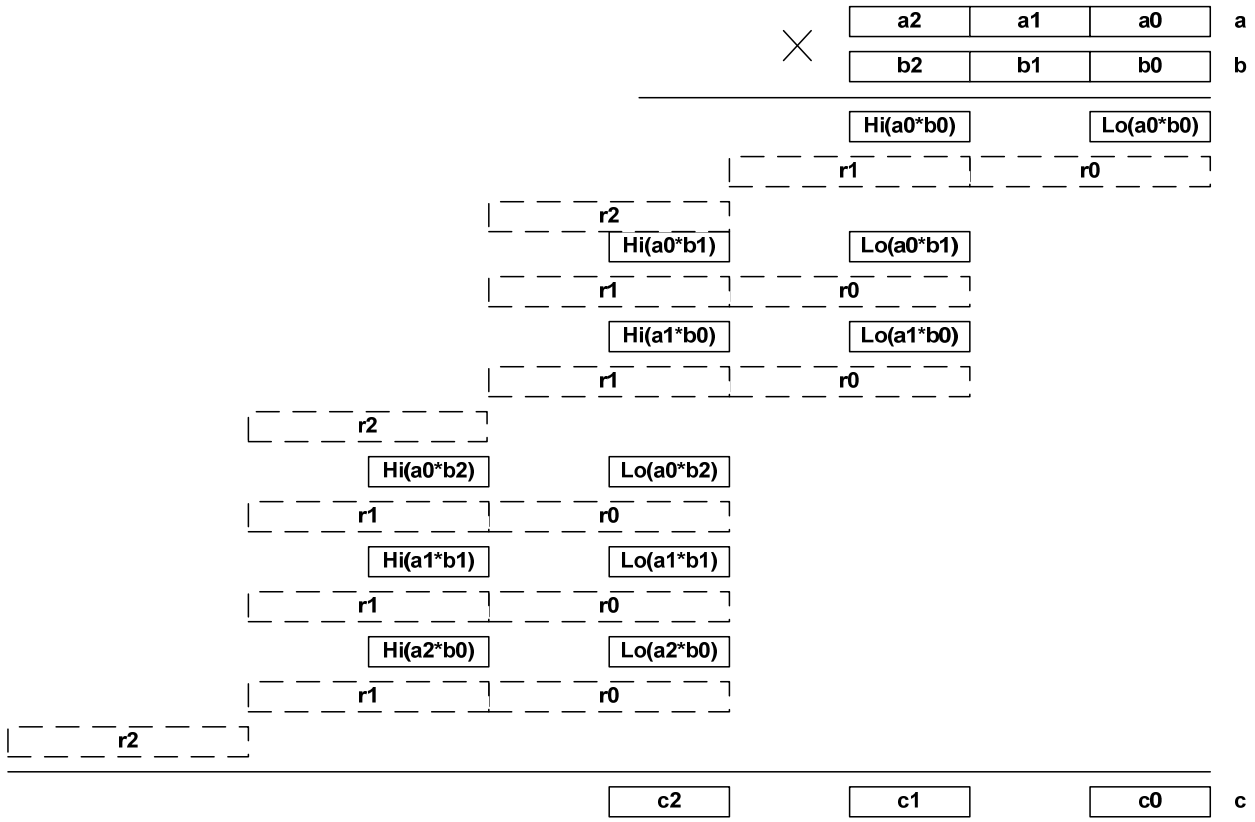

Рис. 3. Графическая интерпретация цикла 2, алгоритма 2

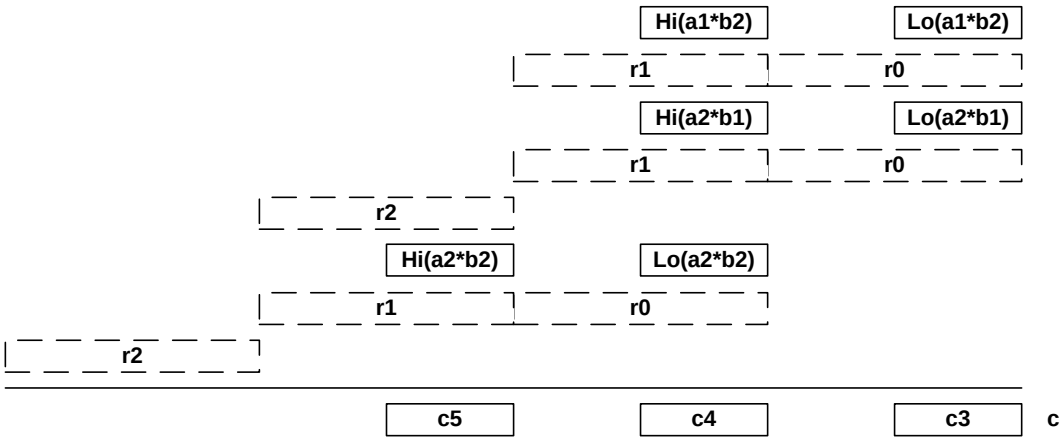

Рис. 4. Графическая интерпретация цикла 3, алгоритма 2

Сравнение с другими алгоритмами. Для проведения объективного сравнения полученных результатов, авторами был проведен обзор известных математических библиотек [14-23], по работе с целыми числами. По результатам анализа, в качестве эталона, была выбрана библиотека GMP [14]. Отметим, что в GMP используется алгоритм Карацубы для умножения целых чисел [2]. Сравнение программных реализаций будет проводиться путем сопоставления среднего времени выполнения программной реализации предложенного алгоритма и реализованного в GMP, для 1 млн. итераций.

Замерять производительность программной реализации предлагается для полей приведенных в работе [25], кроме $\mathbf{GF}(p82)$, и рекомендуемых к использованию для криптографических целей, для обеспечения различных уровней безопасности. В таблице 1 укажем краткое обозначение поля и простое число, по модулю которого образуется поле.

Поля для, которых производился замер производительности Таблица 1

Поле	Простое число
GF(p82)	5000000000000000008503491
GF(p164)	249999999999941304386009994022094639661975160756 99
GF(p192)	6277101735386680763835789423 176059013767194773182842284081
GF(p224)	269599466671506397946670150870196306735579162600263081435100662988 81
GF(p256)	115792089210356248762697446949407573530086143415290314195533631308867097853951
GF(p320)	427197407184182016479004215920066905783641406233172413793356519382596868657626708008708198483809 7
GF(p384)	394020061963944792122790401001436138050797392704654466679469052796276593991132635693989 5 630815229491355443365394 2643
GF(p521)	686479766013060971498190079908139321726943530014330540939446345918554318339765539424505774633321719753296399637136332111386476861244038034037280889270700544 9

Предложенный модифицированный алгоритм Comba и его прототип, алгоритм Comba, были реализованы на C++ и скомпилированы с помощью Microsoft Visual Studio 2010 в Release Win32 конфигурации с параметром Maximize Speed, с поддержкой SSE2.

Эталонная библиотека GMP версии 4.1.2, скомпилированная с помощью Microsoft Visual Studio .NET, тестовое приложение на C++ скомпилировано с помощью Microsoft Visual Studio 2010 в Release Win32 конфигурации с параметром Maximize Speed с поддержкой SSE2.

Тестирование проводилось на распространенной мобильной системе с CPU Intel Core i3 350M и настольной системе CPU Intel Pentium Dual Core E5400.

Результаты замеров для различных программных реализаций и вычислительных систем приведем в таблице 2.

Результаты тестирования операции умножения без приведения по модулю Таблица 2

Поле	**Время, мкс**					
	Core i3			**Pentium Dual Core**		
	Mod. Comba	**Comba**	**GMP4.1**	**Mod Comba**	**Comba**	**GMP4.1**
GF(p82)	0,075	0,120	0,121	0,0687	0,119	0,125
GF(p164)	0,21	0,393	0,4	0,209	0,363	0,407
GF(p192)	0,276	0,393	0,41	0,289	0,363	0,414
GF(p224)	0,343	0,69	0,549	0,364	0,59	0,522
GF(p256)	0,422	0,875	0,638	0,456	0,744	0,648
GF(p320)	0,6973	1,278	0,97	0,686	1,053	0,969
GF(p384)	0,961	1,75	1,38	0,94	1,45	1,36
GF(p521)	1,63	2,8	2,663	1,486	2,41	2,643

Как видно из результатов замеров приведенных в таблице 2, предложенные модификации алгоритма Comba, позволили добиться преимущества в 1,5 раза над GMP. Классическая реализация алгоритма Comba оказалась наиболее медленной, что подтверждается теоретической оценкой (содержит большее число операций сложения и присвоения). Также следует обратить внимание, что предложенные реализации алгоритмов умножения оказались наиболее эффективны на процессоре Intel Pentium Dual Core, с более высокой частотой. Приведенные реализации алгоритмов не подразумевают распараллеливание, поэтому более производительный процессор Intel Core i3 с 4-мя потоками обработки команд не смог реализовать свой поетнциал.

Заключение. По результатам проведенных исследований можно сделать **следующие выводы:**

1. Предложенный в работе подход отложенного переноса, позволяет добиться увеличения производительности программной реализации алгоритма умножения целых чисел Comba, в 1,5-2 раза, а также превзойти производительность популярной математической библиотеки GMP 4.1.2, в среднем в 1,5 раз.
2. Модифицированный алгоритм умножения Comba, является предпочтительнее алгоритма Карацубы [2], используемого в GMP, т.к. программная реализация модифицированного алгоритма умножения Comba оказалась быстрее, реализации алгоритма Карацубы [2] в GMP, для современных аппаратных платформ (32-х и 64-х бит).
3. Механизм отложенного переноса позволяет применить различные техники распараллеливания к модифицированному алгоритму Comba, например OpenMP.

В последнее время, развитие микропроцессоров идет в сторону увеличения потоков обработки команд, позволяет говорить о необходимости разработки полноценных алгоритмов пригодных для эффективной программной реализации на перспективных микропроцессорах.

Компания NVIDIA, уже сейчас предлагает графические процессоры с числом ядер более 256, а также удобную среду разработки CUDA [26], которая позволяет создавать полноценные приложения с поддержкой многопоточности. Данному направлению уже активно уделяется внимание, о чем свидетельствует работа [9]. Дальнейшее направление исследований будет направлено на изучение и эффективное распараллеливание алгоритмов арифметических операций с целыми числами.

Литература

1. Diffie W., Hellman M. E., “New directions in cryptography,” IEEE Transactions on Information Theory, vol. IT-22, pp. 644–654, 1976.
2. Comba P. G. Exponentiation cryptosystems on the IBM PC // IBM Systems Journal. –Vol. 29(4). -1990. -pp. 526–538.
3. Brown M., Hankerson D., Lopez J., Menezes A. Software implementation of the NIST elliptic curves over prime fields // Research Report CORR 2000–55. Department of Combinatorics and Optimization, University of Waterloo. –Canada: Waterloo, Ontario, 2000. –21p.
4. Hong S-M., Oh S-Y., Yoon H. New Modular Multiplication algorithms for fast modular exponeniation // Advances in Cryptology-Proceedings of Eurocrypt ’96. –Springer-Verlag. -1996. –pp.166-177.
5. Avanzi R. M. Aspects of hyperelliptic curves over large prime fields in software implementations // Cryptology ePrint Archive. –Report 2003/253. –2003. –23p. Available at: http://eprint.iacr.org
6. Paar C. Implementation options for finite filed arithmetic for elliptic curve cryptosystems // Worchester Polytechnic Institute. –ECC’99. –1999. –31p. Available at: http://www.ece.wpi.edu/research/crypto.html
7. Gaubatz G. Versatile Montgomery multiplier architectures. Master thesis: electrical and computer engineering. –2002. –Worcester polytechnic institute. –101p.
8. Johann Großschadl, Roberto M. Avanzi, Erkay Sava, Stefan Tillich. Energy-Efficient Software Implementation of Long Integer Modular Arithmetic // Advances in Cryptology-Prociding in CHES’2005. –Springer-Verlag. -2005. -LNCS 3659. -pp.75-90.

9. Giorgi P. Izard T, Tisserand A. Comparison of Modular Arithmetic Algorithms on GPUs. URL: http://hal-lirmm.ccsd.cnrs.fr/lirmm-00424288/fr/
10. Weimerskirch A., Paar C. Generalizations of the Karatsuba Algorithm for Efficient Implementations. // Cryptology ePrint Archive. –Report 2006/224. –2006. –17p. Available at: http://eprint.iacr.org
11. Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-025. Available at: http://intel.com
12. The OpenMP API Specification for Parallel Programming. Available at: http://openmp.org
13. OpenMP in Visual C++. Available at: http:// http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx
14. The GNU Multiply Precision Library (GMP). URL: http://gmplib.org/
15. LiDIA. URL: https://www.cdc.informatik.tu-darmstadt.de/en/cdc
16. Multiprecision Unsigned Number Template Library (MUNTL). URL: http://mktmk.narod.ru/eng/muntl/muntl.htm
17. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. URL: http://discovery.csc.ncsu.edu/software/TinyECC/
18. Galois Field Arithmetic Library. URL: http://www.partow.net/projects/galois/
19. MPFQ: Fast Finite Fields Library. URL: http://mpfq.gforge.inria.fr/
20. BBNUM. URL: http://www.iw-net.org/index.php?title=Bbnum_library
21. FLINT: Fast Library for Number Theory. URL: http://www.flintlib.org
22. Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL). URL: http://indigo.ie/~mscott
23. LibTom Projects: LibTomMath, TomsFastMath. URL: http://libtom.org
24. Giorgi P., Imbert L., Izard T. Multipartite Modular Multiplication. Preprint. URL: http://hal.archives-ouvertes.fr/lirmm-00618437/fr/
25. National Institute of Standards and Technology, Recommended Elliptic Curves for Federal Government Use, Appendix to FIPS 186-2, 2000. –43p.
26. NVIDIA. NVIDIA CUDA Programming Guide 2.0. 2008.