УДК 004.056.55
В.Ю.КОВТУН, А.А. ОХРИМЕНКО

УМНОЖЕНИЕ ЦЕЛЫХ ЧИСЕЛ С ИСПОЛЬЗОВАНИЕМ ОТЛОЖЕНОГО ПЕРЕНОСА ДЛЯ КРИПТОСИСТЕМ С ОТКРЫТЫМ КЛЮЧОМ

Развитие современных информационных технологий характеризуется постоянным повышением значения информации, что в свою очередь способствует проявлению к ней интереса злоумышленников. Повсеместное использование информационных систем требует использования современных средств и методов защиты информации и их непрерывного совершенствования.

С появлением криптографических преобразований с открытым ключом (КПОК) понятие о защите информации, а вместе с ним и функции криптографии, значительно расширились. Наиболее распространенные функции систем на основе КОК - шифрование и электронная цифровая подпись (ЭЦП) [9]. Роль ЭЦП в последнее время значительно возросла, относительно традиционного шифрования. Но из-за особенностей алгоритмов, которые имеют высокую вычислительную сложность, системы на основе КПОК значительно уступают в быстродействии симметричным системам [9], что накладывает ограничения на области их применения. Кроме того, основные идеи и алгоритмы КОК были ориентированы на последовательное выполнение операций, так как многопроцессорные и многоядерные системы не имели широкого распространения, а сегмент мобильных устройств еще не существовал. То есть, в современных условиях, КПОК не используют весь потенциал и архитектуры современных вычислительных систем.

Таким образом, повышение производительности КПОК является **актуальной задачей**. Для использования КПОК в мобильных системах, которые обладают ограниченными ресурсами, необходимо применение других алгоритмов, которые бы использовали меньше памяти, были более быстрыми и эффективнее использовали энергию. Что касается многопроцессорных и многоядерных систем, то в большинстве случаев в алгоритмах не учитываются потенциальные возможности по распараллеливанию и использовании w-разрядных машинных слов.

На протяжении всего периода развития КПОК, их основу составляют операции в кольцах и полях чисел, среди которых, умножение занимает ведущее место (рис. 1), являясь при этом достаточно трудоемкой операцией [10].

<table>
<tr><td colspan="2">Криптопреобразования</td><td colspan="2">Зашифровывание/ расшифровывание</td><td colspan="2">Формирование и проверка цифровой подписи</td><td>Обмен ключами</td></tr>
<tr><td colspan="2" rowspan="2">Арифметика в поле целых чисел</td><td colspan="5">Возведение в степень</td></tr>
<tr><td>Умножение</td><td>Сложение</td><td>Вычитание</td><td>Возведение в квадрат</td><td>Инвертирование</td></tr>
<tr><td colspan="2">Команды CPU</td><td colspan="5">mov, mul, shr, shl, add, sub ...</td></tr>
</table>

Рис. 1. Иерархия операций в криптосистемах с открытым ключом

Таким образом, **целью данной работы** является поиск путей увеличения скорости КПОК, путём увеличения быстродействия операций умножения в кольцах и полях чисел, используя метод отложенного переноса и подходы эффективного распараллеливания.

Известны следующие подходы увеличения быстродействия КПОК [7, 10]:

- уменьшение общего числа операций в криптографическом алгоритме;
- совершенствование самих алгоритмов;
- совершенствование структур данных;
- увеличение разрядности машинных слов;
- использование специализированых команд процессоров;
- распараллеливание.

В работе предлагается компиляция перечисленных подходов для повышения эффективности программной реализации операции умножения целых чисел.

Сегодня известны следующие алгоритмы умножения целых чисел:

- «в столбик» [5];
- рекурсивный алгоритм Карацубы-Офмана [5];
- алгоритм Шёнхаге – Штрассена (дискретное преобразование Фурье);
- алгоритм Фюрера (развитие Шёнхаге - Штрассена);
- алгоритм *Comba* [2, 4];
- и другие.

Анализ публикаций, проведенный в работе [10], позволил выделить алгоритм умножения *Comba* [2], показавший лучшие результаты тестов производительности программной реализации на современных платформах [3, 4, 6, 8] для длин чисел, применяемых в криптографии. Далее рассматривается обобщение алгоритма умножения целых чисел *Comba* для w-разрядных систем, используя предложенные подходы, повышающие производительность операции умножения для КПОК.

Описание алгоритма-прототипа умножения

Основу алгоритма ***Comba*** [2] составляет цикл п.2,3 и вложенный цикл 2.1, 3.1. На низшем уровне иерархии, в цикле п. 2.1, 3.1 выполняется умножения $(uv)^{(64)}$, результат является 64-х разрядным целым, которое затем разделяется на два 32-разрядных $u^{(32)}$ и $v^{(32)}$. Накопление суммы производится в 32-разрядных временных переменных r_0, r_1 и r_2, на каждой итерации п. 2.1.2, 3.1.2 и п. 2.1.3, 3.1.3. Присвоение конечного результата, а также изменение аккумуляторов суммы r_0, r_1 и r_2, происходит на каждой итерации п. 2.2, 3.2.

INPUT:	**OUTPUT:**
$a, b \in GF(p),\ n = \log_{2^w} a,\ nk = 2n-1$	$c = a \cdot b$

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).

Основные недостатки алгоритма ***Comba***:

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

В этом случае, выполняется 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.

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. 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, что не позволяет выполнять спаривание и распараллеливание таких операций, как следствие – неэффективное использование ресурсов процессора.
- Высокая внутренняя связность алгоритма, 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.

На рис. 2 проиллюстрирована, для $n = 3$, работа алгоритма ***Comba***.

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 Comba реализует подход известный со школы – «умножение в столбик», с небольшим отличием: умножается часть множителя $a_i \quad i = \overline{1, n}$ на все части другого множителя $b_j \quad j = \overline{1, n}$, 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 $b_j \quad j = \overline{1, n}$ (по строкам).

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

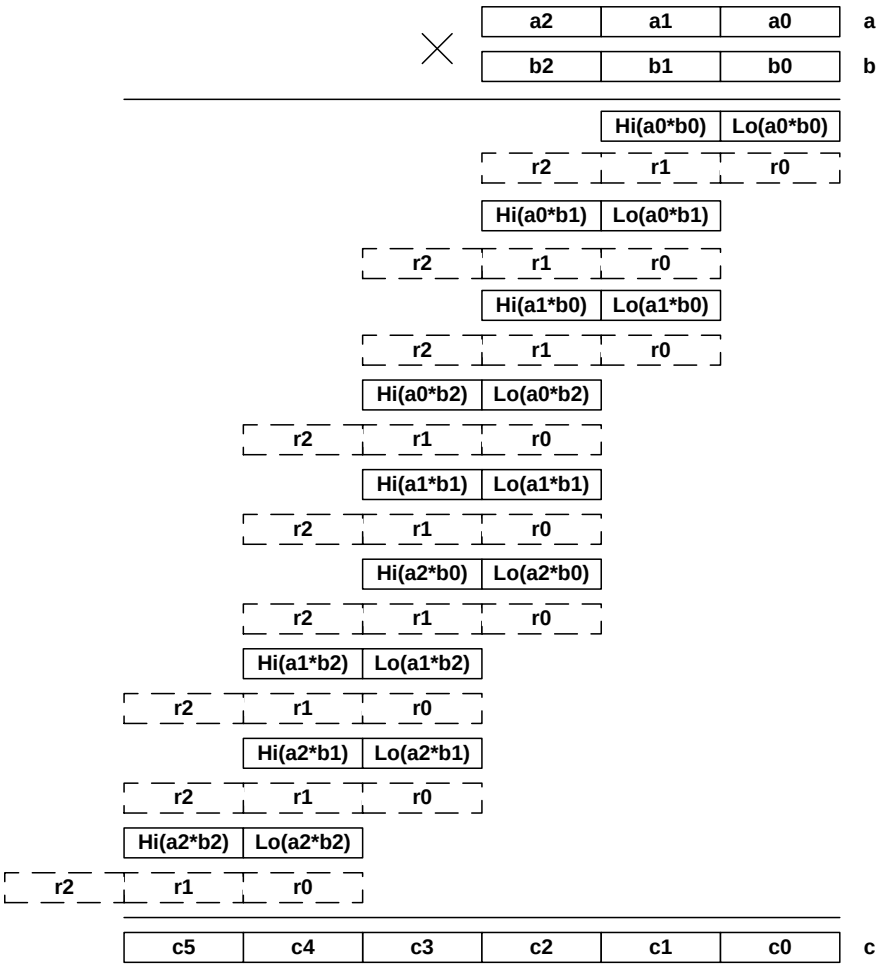

Рис. 2. Графическая интерпретация алгоритма *Comba*

В работе [10] авторы предложили подходы, 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 технологию *OpenMP*.

Modified Comba

В работе [10], предлагается модифицированный алгоритм ***Comba*** [2, 3] – ***Modified Comba (MC)***, в котором используется идея отложенного переноса.

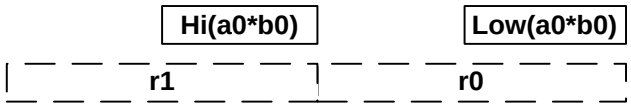

Рис. 3. Идея механизма отложенного переноса

Рассмотрим обобщение предложенного в [10] похода для *w*-разрядных машинных слов. Использование *2w*-разрядных переменных для хранения *w*-разрядных переменных, позволило избавиться от учета переноса из *w*-разрядной переменной, после каждой арифметической операции. Перенос накапливается в старшей части *2w*-разрядной переменной и может быть учтён при необходимости (рис. 3).

Обобщенный алгоритм ***MC*** [10] для *w*-разрядных систем приведен ниже:

INPUT:	**OUTPUT:**
$a, b \in GF(p),\ n = \log_{2^w} a,\ nk = 2n-1$	$c = a \cdot b$

1. $r_0^{(2w)} \leftarrow 0,\ r_1^{(2w)} \leftarrow 0,\ r_2^{(2w)} \leftarrow 0.$
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. $(uv)^{(2w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}.$
2.1.2. $r_0^{(2w)} \leftarrow r_0^{(2w)} + v^{(w)},\ r_1^{(2w)} \leftarrow r_1^{(2w)} + u^{(w)}$ ***//накопление отложенного переноса***
2.2. $r_1^{(2w)} \leftarrow r_1^{(2w)} + hi_{(w)}\left(r_0^{(2w)}\right),\ r_2^{(2w)} \leftarrow r_2^{(2w)} + hi_{(w)}\left(r_1^{(2w)}\right)$ ***//учёт отложенного переноса***
2.3. $c_k^{(w)} \leftarrow low_{(w)}\left(r_0^{(2w)}\right),\ r_0^{(2w)} \leftarrow low_{(w)}\left(r_1^{(2w)}\right),\ r_1^{(2w)} \leftarrow low_{(w)}\left(r_2^{(2w)}\right),\ r_2^{(2w)} \leftarrow 0.$
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. $(uv)^{(2w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}.$
3.1.2. $r_0^{(2w)} \leftarrow r_0^{(2w)} + v^{(w)},\ r_1^{(2w)} \leftarrow r_1^{(2w)} + u^{(w)}$ ***//накопление отложенного переноса***
3.2. $r_1^{(2w)} \leftarrow r_1^{(2w)} + hi_{(w)}\left(r_0^{(2w)}\right),\ r_2^{(2w)} \leftarrow r_2^{(2w)} + hi_{(2)}\left(r_1^{(2w)}\right)$ ***//учёт отложенного переноса***
3.3. $c_k^{(w)} \leftarrow low_{(w)}\left(r_0^{(2w)}\right),\ r_0^{(2w)} \leftarrow low_{(w)}\left(r_1^{(2w)}\right),\ r_1^{(2w)} \leftarrow low_{(w)}\left(r_2^{(2w)}\right),\ r_2^{(2w)} \leftarrow 0.$
4. $c_{nk}^{(w)} \leftarrow low_{(2)}\left(r_0^{(2w)}\right).$
5. Return (c).

Предложенная идея отложенного переноса позволяет независимо производить сложение результатов соответствующих произведений по столбцам, что дает возможность выполнять накопление суммы старших и младших разрядов в отдельных параллельных потоках. Однако, после

завершения накопления суммы в каждом отдельном потоке, все же необходимо выполнить корректировку (учесть перенос) $r_1 = r_1 + Hi(r_0)$, $r_2 = r_2 + Hi(r_1)$ и сформировать результат $c_i = Low(r_0)$. На рис. 4 и 5 приведена графическая интерпретация алгоритма Modified Comba для $n = 3$, где четко прослеживается сложение результатов соответствующих произведений по столбцам.

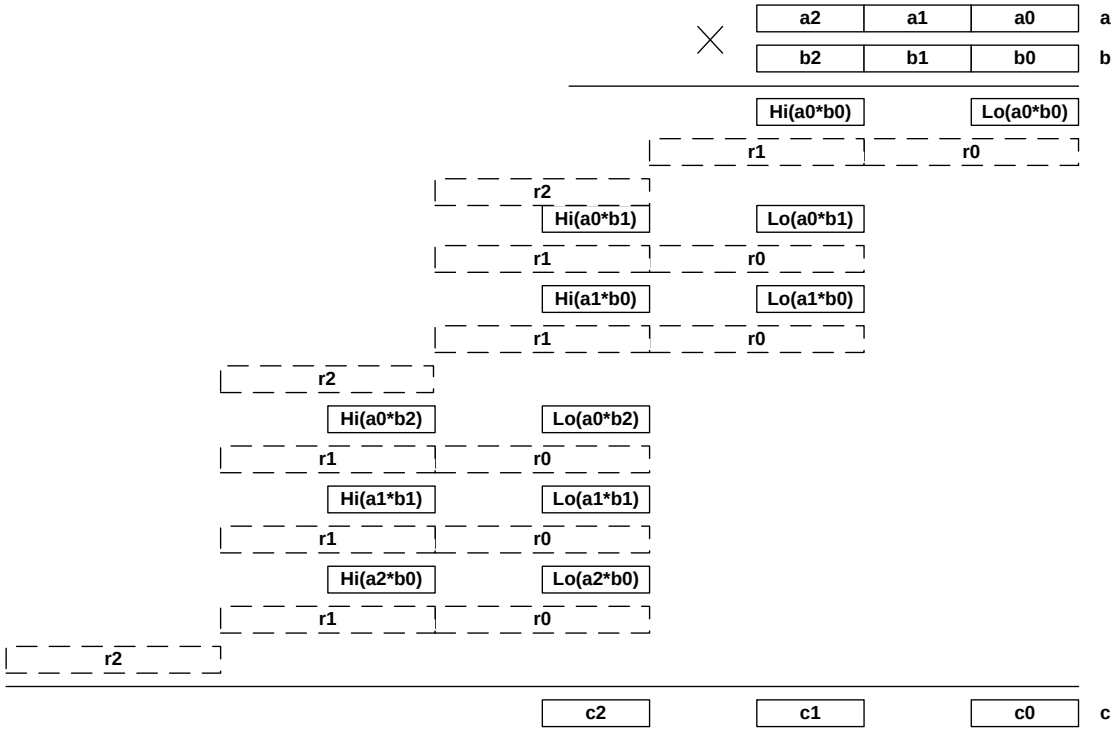

Рис. 4. Графическая интерпретация цикла 2, алгоритма Modified Comba

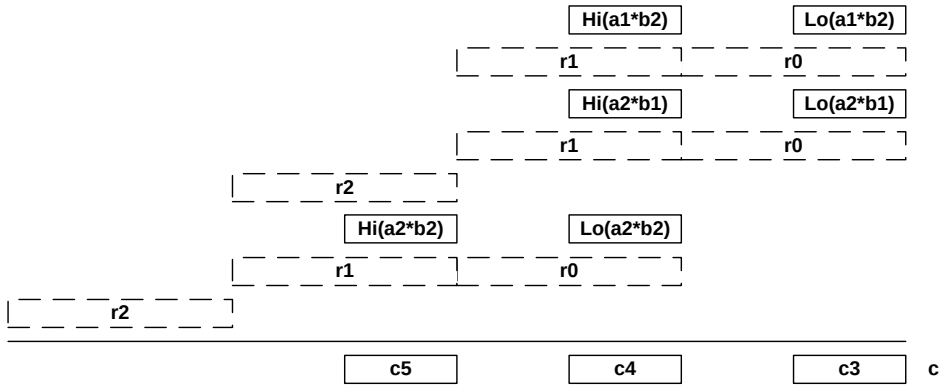

Рис. 5. Графическая интерпретация цикла 3, алгоритма Modified Comba

Механизм отложенного переноса позволяет сформулировать несколько подходов к распараллеливанию алгоритма умножения ***MC***:

- Параллельное выполнение циклов п. 2 и п. 3, с последующей коррекцией результатов, используя два параллельных потока.
- Параллельное выполнение итераций циклов п. 2 и п. 3, с последующим слиянием промежуточных результатов, с использованием множества параллельных потоков.

Modified Comba 2x с двумя потоками

В алгоритме ***MC*** присутствует два цикла п. 2 и п. 3, которые производят чтение элементов $a_i^{(w)}$ и $b_j^{(w)}$ из соответствующих массивов, а также запись результатов умножения в элементы $c_k^{(w)}$. Заметим, что индексы k в циклах п. 2 и п. 3 не повторяются при записи в $c_k^{(w)}$, это позволяет говорить о независимости данных в циклах и возможности применения техники распараллеливания для их параллельного выполнения. Следует обратить внимание, что оба цикла п. 2 и п. 3 используют общие временные переменные r_0, r_1 и r_2. Причем в r_0 и r_1 хранятся значения, которые используются в цикле п. 3 после окончания цикла п. 2. Таким образом, после окончания цикла п. 3 необходимо выполнить корректировку – учесть результаты работы цикла п. 2 в результатах цикла п. 3, которые хранятся во временных переменных r_0 и r_1. Заметим, что при распараллеливании, каждый поток работает с собственными временными переменными rl_0, rl_1 и rl_2. Глобальные переменные r_0 и r_1 необходимы лишь для передачи возможного переноса из цикла п. 2, для последующей корректировки результатов накопления в цикле п. 3.Обобщенный алгоритм ***Modified Comba 2x*** (***MC2x***) для *w*-разрядных систем:

INPUT:	**OUTPUT:**
$a, b \in GF(p),\ n = \log_{2^w} a,\ nk = 2n-1$	$c = a \cdot b$

1. #pragma omp parallel sections private($r_0^{(2w)}, r_1^{(2w)}$) begin
1.1. #pragma omp section begin
1.1.1. $rl_0^{(2w)} \leftarrow 0,\ rl_1^{(2w)} \leftarrow 0,\ rl_2^{(2w)} \leftarrow 0.$
1.1.2. For $k \leftarrow 0,\ k < n,\ k{+}{+}$ do
1.1.2.1. For $i \leftarrow 0,\ j \leftarrow k,\ i \le k,\ i{+}{+},\ j{-}{-}$ do
1.1.2.1.1. $(uv)^{(2w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}.$
1.1.2.1.2. $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + v^{(w)},\ rl_1^{(2w)} \leftarrow rl_1^{(2w)} + u^{(w)}$ ***//накопление отложенного переноса***
1.1.2.2. $rl_1^{(2w)} \leftarrow rl_1^{(2w)} + hi_{(w)}\left(rl_0^{(2w)}\right),\ rl_2^{(2w)} \leftarrow rl_2^{(2w)} + hi_{(w)}\left(rl_1^{(2w)}\right)$ ***//учёт переноса***
1.1.2.3. $c_k^{(w)} \leftarrow low_{(w)}\left(rl_0^{(2w)}\right),\ rl_0^{(2w)} \leftarrow low_{(w)}\left(rl_1^{(2w)}\right),\ rl_1^{(2w)} \leftarrow low_{(w)}\left(rl_2^{(2w)}\right),\ rl_2^{(2w)} \leftarrow 0.$
1.1.3. $r_0^{(2w)} \leftarrow rl_0^{(2w)}.$
#pragma omp section end
1.2. #pragma omp section begin
1.2.1. $rl_0^{(2w)} \leftarrow 0,\ rl_1^{(2w)} \leftarrow 0,\ rl_2^{(2w)} \leftarrow 0.$
1.2.2. For $k \leftarrow n,\ l \leftarrow 1, k < nk,\ k{+}{+},\ l{+}{+}$ do
1.2.2.1. For $i \leftarrow l,\ j \leftarrow n-1,\ i < n,\ i{+}{+},\ j{-}{-}$ do

1.2.2.1.1. $(uv)^{(2w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}$.

1.2.2.1.2. $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + v^{(w)}$, $rl_1^{(2w)} \leftarrow rl_1^{(2w)} + u^{(w)}$ ***//накопление отложенного переноса***

1.2.2.2. $rl_1^{(2w)} \leftarrow rl_1^{(2w)} + hi_{(w)}\left(rl_0^{(2w)}\right)$, $rl_2^{(2w)} \leftarrow rl_2^{(2w)} + hi_{(w)}\left(rl_1^{(2w)}\right)$ ***//учёт переноса***

1.2.2.3. $c_k^{(w)} \leftarrow low_{(w)}\left(rl_0^{(2w)}\right)$, $rl_0^{(2w)} \leftarrow low_{(w)}\left(rl_1^{(2w)}\right)$, $rl_1^{(2w)} \leftarrow low_{(w)}\left(rl_2^{(2w)}\right)$, $rl_2^{(2w)} \leftarrow 0$.

#pragma omp section end

#pragma omp parallel sections end

2. For $k \leftarrow n$, $k < nk$, $k{+}{+}$ do

2.1. $r_0^{(2w)} \leftarrow r_0^{(2w)} + c_k^{(w)}$. 2.2. $c_k^{(w)} \leftarrow low_{(w)}\left(r_0^{(2w)}\right)$.

2.3. $low_{(w)}\left(r_0^{(2w)}\right) \leftarrow hi_{(w)}\left(r_0^{(2w)}\right)$. 2.4. $hi_{(w)}\left(r_0^{(2w)}\right) \leftarrow 0$.

3. $c_{nk}^{(w)} \leftarrow c_{nk}^{(w)} + low_{(w)}\left(r_0^{(2w)}\right)$.

4. Return (c).

Вычислительная сложность алгоритма:

$$I_{mul}^{MC2x} = 4I_{assign}^{2w} + \max\left[n\left(\left\lceil \tfrac{n}{2} \right\rceil\left(I_{mul}^{w} + 2I_{add}^{2w+w}\right) + 2I_{add}^{2w+w} + 4I_{assign}^{2w}\right),\right.$$
$$\left. n\left(\left\lfloor \tfrac{n}{2} \right\rfloor\left(I_{mul}^{w} + 2I_{add}^{2w+w}\right) + 2I_{add}^{2w+w} + 4I_{assign}^{2w}\right)\right] + \tfrac{n}{2}\left[I_{add}^{2w+w} + 3I_{assign}^{w}\right] + I_{add}^{w}$$

На рис. 6 показано работу алгоритма с 2-мя потоками команд для $n = 3$.

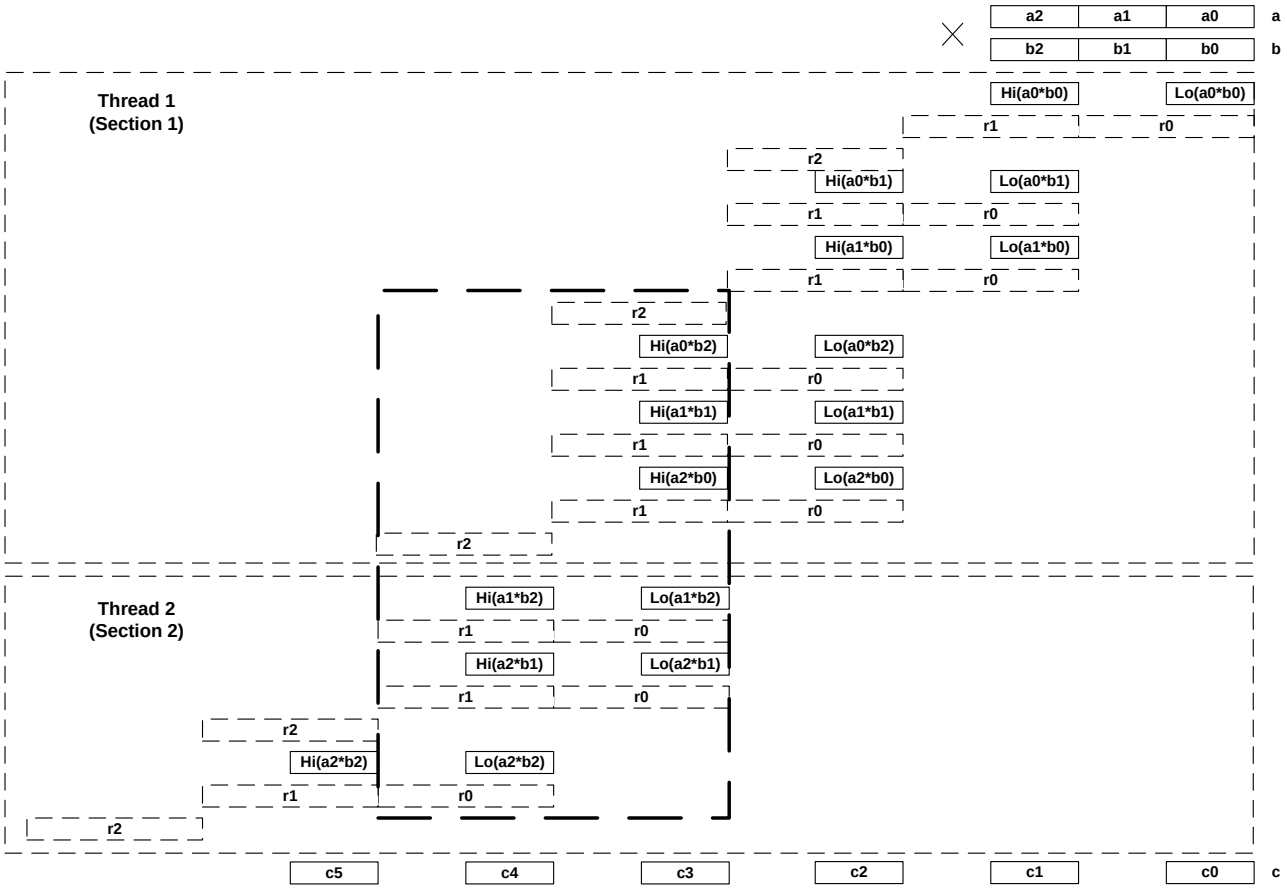

Рис. 6. Графическая интерпретация цикла 2, алгоритма Modified Comba 2x

Особого внимания заслуживает алгоритм с распараллеливанием на нескольких потоках, который изложен ниже.

Modified Comba Mx с множеством потоков

При более детальном рассмотрении алгоритма ***MC***, легко заметить, что итерации циклов п. 2 и п. 3 не зависят друг от друга. Исключение составляют результаты накопления сложения и переноса после текущей итерации и переходе к следующей итерации в п. 2.2 и п. 3.2. Посредством введения индивидуальных локальных переменных накопления суммы в рамках итерации, можно корректно произвести накопление суммы в итерациях цикла п.2 и п.3 – параллельно. Для этих целей в алгоритме ***Modified Comba Mx*** (***MCMx***) создано два массива $r0_i^{(2w)}$ и $r1_i^{(2w)}$, $i = \overline{0, 2n-1}$. Такой подход позволяет варьировать числом параллельных потоков, без модификации алгоритма в целом.

Обобщенный алгоритм ***MCMx*** для *w*-разрядных систем приведен ниже.

INPUT:	**OUTPUT:**
$a, b \in GF(p),\ n = \log_{2^w} a,\ nk = 2n-1$	$c = a \cdot b$

1. $l \leftarrow 1$.
2. Arrays $r0_i^{(w)}$ and $r1_i^{(w)}$, $i = \overline{0, nk}$.
3. #pragma omp parallel private ($r0_i^{(w)}, r1_i^{(w)}$) reduction (+ : l) begin
4. #pragma omp for nowait begin
4.1. For $k \leftarrow 0,\ k < n,\ k{+}{+}$ do
4.1.1. $rl_0^{(w)} \leftarrow 0,\ rl_1^{(w)} \leftarrow 0$.
4.1.2. For $i \leftarrow 0,\ j \leftarrow k,\ i \le k,\ i{+}{+},\ j{-}{-}$ do
4.1.2.1. $(uv)^{(2w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}$.
4.1.2.2. $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + v^{(w)},\ rl_1^{(2w)} \leftarrow rl_1^{(2w)} + u^{(w)}$ ***//накопление отложенного переноса***
4.1.3. $r0_k^{(2w)} \leftarrow rl_0^{(2w)},\ r1_k^{(2w)} \leftarrow rl_1^{(2w)}$ ***//сохранение переносов***
#pragma omp for end
5. #pragma omp for nowait begin
5.1. For $k \leftarrow n,\ k < nk,\ k{+}{+}$ do
5.1.1. $rl_0^{(2w)} \leftarrow 0,\ rl_1^{(2w)} \leftarrow 0$.
5.1.2. For $i \leftarrow l,\ j \leftarrow n-1,\ i < n,\ i{+}{+},\ j{-}{-}$ do
5.1.2.1. $(uv)^{(2w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}$.
5.1.2.2. $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + v^{(w)},\ rl_1^{(2w)} \leftarrow rl_1^{(2w)} + u^{(w)}$ ***//накопление отложенного переноса***
5.1.3. $r0_k^{(2w)} \leftarrow rl_0^{(2w)},\ r1_k^{(2w)} \leftarrow rl_1^{(2w)}$ ***//сохранение переносов***
5.1.4. $l{+}{+}$.
#pragma omp for end
#pragma omp parallel end
6. $r_0^{(2w)} \leftarrow 0$.
7. For $k \leftarrow 0,\ k < nk,\ k{+}{+}$ do ***//цикл учёта сохраненных переносов***

7.1. $rll_0^{(2w)} \leftarrow r0_k^{(2w)}$, $rll_1^{(2w)} \leftarrow r1_k^{(2w)}$.
7.2. $rll_0^{(2w)} \leftarrow rll_0^{(2w)} + low_{(w)}\left(r^{(2w)}\right)$.
7.3. $c_k^{(w)} \leftarrow low_{(w)}\left(rll_0^{(2w)}\right)$.
7.4. $rll_1^{(2w)} \leftarrow rll_1^{(2w)} + low_{(w)}\left(rll_0^{(2w)}\right)$.
7.5. $r^{(2w)} \leftarrow rll_1^{(2w)}$.
8. $c_{nk}^{(w)} \leftarrow low_{(w)}\left(r^{(2w)}\right)$.
9. Return (c).

Вычислительная сложность предложеного алгоритма:

$$I_{mul}^{MCMx} = \frac{n}{Z}\left[4I_{assign}^{w} + \left\lceil \frac{n}{2} \right\rceil \left(I_{mul}^{w} + 2I_{add}^{2w+w}\right) + \frac{n}{Z}\left[4I_{assign}^{w} + \left\lfloor \frac{n}{2} \right\rfloor \left(I_{mul}^{w} + 2I_{add}^{2w+w}\right)\right] + \right.$$
$$\left. + I_{assign}^{2w} + I_{assign}^{w} + (2n-1)\left[3I_{assign}^{2w} + 3I_{add}^{2w+w}\right],\right.$$

где Z – число потоков.

Экспериментальное исследование предложенных алгоритмов

Предложенный модифицированный алгоритм ***MC***, а также распараллеленные его версии ***MC2x*** и ***MCMx***, были реализованы на *C++* и скомпилированы с помощью *Intel C++ Compiler XE 13*. Для проверки предложенных алгоритмов использовались две сборки, в которых использовались машинные слова размером 32 и 64 бита соответственно. Так как результатом умножения двух 64-битных целых чисел будет 128-битное число, в виду отсутствия переменных и операции сложения для переменных такого размера, при умножении использовалась встроенная в компилятор *intrinsic* функция *_umul128,* которая позволяет обойти это ограничение (128-битная переменная представлена в виде массива из 64-битных слов).

На основе проведенного анализа [10] известных математических библиотек по работе с целыми числами [1], применяемые в криптографии, в качестве эталона для сравнения полученных результатов, была выбрана библиотека *GMP* (используется алгоритм Карацубы для умножения целых чисел [1]). Сравнение проводилось путем сопоставления среднего времени выполнения операции умножения в программной реализации предложенных алгоритмов и реализованного в *GMP*, для 1 млн. итераций.

Эталонная библиотека *GMP* версии *4.1.2*, скомпилирована с помощью *Microsoft Visual Studio .NET*, тестовое приложение на *C++* – с помощью *Intel C++ Compiler XE 13*.

Замеры производительности проводились для двоичных чисел различной длинны.

Эксперимент проводился на компьютере под управлением операционной системы *Microsoft Windows 7 Ultimate x64 SP1* и процессором *Intel Core i5-3570 (6M Cache, 3.40 GHz)* с четырьмя физическими ядрами.

Результаты эксперимента реализованных алгоритмов с использованием 32-битных машинных слов представлены в таблице 1.

Таблица 1. Результаты экспериментов для *w=32 bit*

Size	**MC, мс**	**MC2x, мс**	**MCMx, мс**	**GMP, мс**	**Size**	**MC, мс**	**MC2x, мс**	**MCMx, мс**	**GMP, мс**
128	63	764	764	93	**6144**	77 735	45 755	5 569	67 751
256	203	764	749	312	**8192**	137 265	79 591	8 767	117 265
512	640	1 014	764	1 186	**12288**	305 605	180 633	15 740	216 482
1024	2 340	1 919	889	3 697	**16384**	540 916	310 035	24 975	344 823
2048	8 923	5 881	1 591	11 794	**24576**	1 213 120	691 226	51 215	662 361
3072	19 905	12 543	2 543	21 871	**32768**	2 172 853	1 237 771	90 082	957 296
4096	34 991	20 561	3 401	37 050					

Для графического представления результатов эксперимента, предлагается нормализовать данные, путем отношения результатов эталонной библиотеки ***GMP*** к результатам ***MC, MC2x и MCMx*** (таблица 2 и рис.7).

Таблица 2. Преимущество над эталонной библиотекой для *w=32 bit*

Size	**GMP / MC**	**GMP / MC2x**	**GMP / MCMx**	**Size**	**GMP / MC**	**GMP / MC2x**	**GMP / MCMx**
128	1,476	0,122	0,122	**6144**	0,872	1,481	12,166
256	1,537	0,408	0,417	**8192**	0,854	1,473	13,376
512	1,853	1,170	1,552	**12288**	0,708	1,198	13,754
1024	1,580	1,927	4,159	**16384**	0,637	1,112	13,807
2048	1,322	2,005	7,413	**24576**	0,546	0,958	12,933
3072	1,099	1,744	8,600	**32768**	0,441	0,773	10,627
4096	1,059	1,802	10,894				

Из таблицы 2 видно, что предложенный алгоритм ***MC*** показывает лучшие результаты на числах размером 128–4096 бит, которые сейчас широко используются в криптографии; алгоритм ***MC2x*** эффективнее на числах с 512 по 16384 бит. В то же время, самые лучшие результаты показывает алгоритм ***MCMx***, который на некоторых числах, имеет почти 14-кратное преимущество по сравнению с ***GMP***. Алгоритмы ***MC2x*** и ***MCMx*** показывают хуже результаты на числах малых размеров (128 и 256 бит), по сравнению с ***MC.*** Это объясняется затратами времени на инициализацию потоков обработки данных. Для чисел длинной более 16384 бит, преимущество сокращается, из-за большей, чем у алгоритма Карацубы [10] вычислительной сложности.

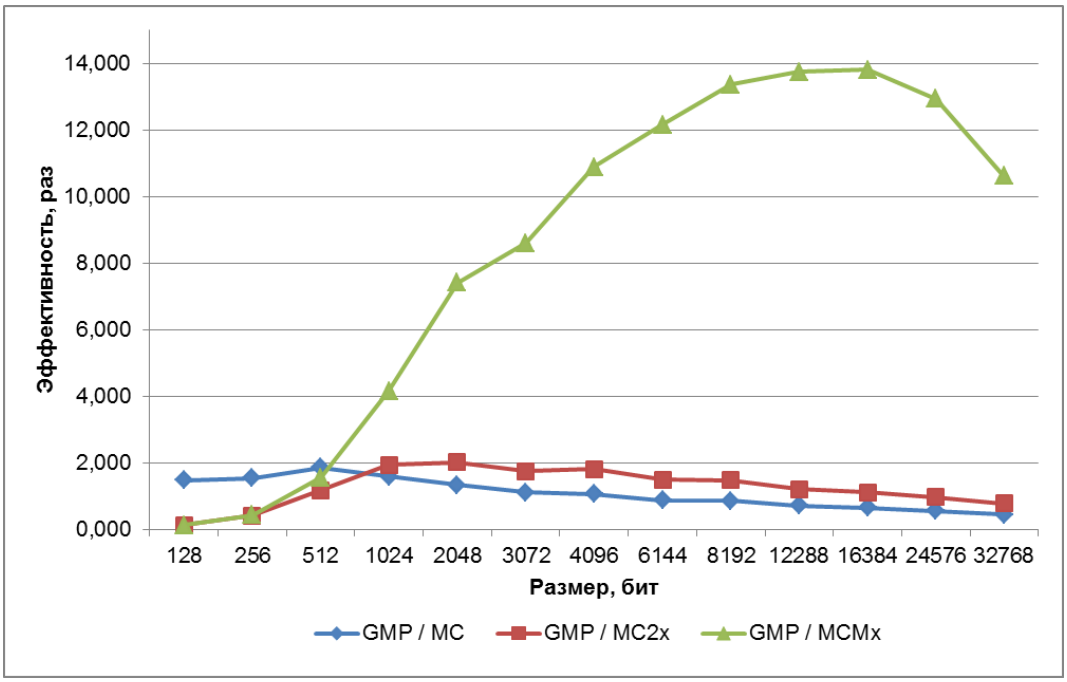

Рис.7. Преимущество предложенных алгоритмов перед GMP

Результаты экспериментов программной реализации алгоритмов с 64-битными машинными словами, представлены в таблице 3.

Таблица 3. Результаты экспериментов для *w=64 bit*

Size	MC, мс	MC2x, мс	MCMx, мс	Size	MC, мс	MC2x, мс	MCMx, мс
128	21	725	847	**6144**	19 063	11 529	10 858
256	50	647	771	**8192**	33 571	19 875	18 049
512	153	780	873	**12288**	74 786	44 055	38 953
1024	558	1 141	1 102	**16384**	132 538	77 438	68 515
2048	2 153	1 779	1 840	**24576**	296 369	171 288	151 867
3072	4 867	3 385	3 260	**32768**	524 645	302 453	267 759
4096	8 580	5 507	5 226				

Для анализа эффективности программных реализаций предложенных алгоритмов с использованием разной разрядности, предлагается провести сравнение, путём отношения полученных результатов друг к другу (реализации x86 к x64). Результаты сравнения представлены в таблице 4 и на рис.8.

Таблица 2. Преимущество реализации для *w=64 bit* над *w=32 bit*

Size	MC x86 / MC x64	MC2x x86 / MC2x x64	MCMx x86 / MCMx x64	Size	MC x86 / MC x64	MC2x x86 / MC2x x64	MCMx x86 / MCMx x64
128	3,000	1,054	0,902	**6144**	4,078	3,969	0,513
256	4,060	1,181	0,971	**8192**	4,089	4,005	0,486
512	4,183	1,300	0,875	**12288**	4,086	4,100	0,404
1024	4,194	1,682	0,807	**16384**	4,081	4,004	0,365
2048	4,144	3,306	0,865	**24576**	4,093	4,035	0,337
3072	4,090	3,705	0,780	**32768**	4,142	4,092	0,336
4096	4,078	3,734	0,651				

Проанализировав полученные результаты, следует, что *x64* реализация для алгоритма ***MC*** примерно в 4 раза эффективнее (кроме чисел размером 128 бит – там преимущество в 3 раза), а ***MC2x*** до четырех раз лучше, чем *x86*. В то же время, реализация ***MCMx*** *x64* оказалась хуже, что можно объяснить несовершенством используемых типов данных, а также тем, что пока не существует эффективной реализации 128-битного сложения на аппаратном уровне. ***MCMx*** *x64* показывает лучшие результаты по сравнению с ***MC*** и ***MC2x***.

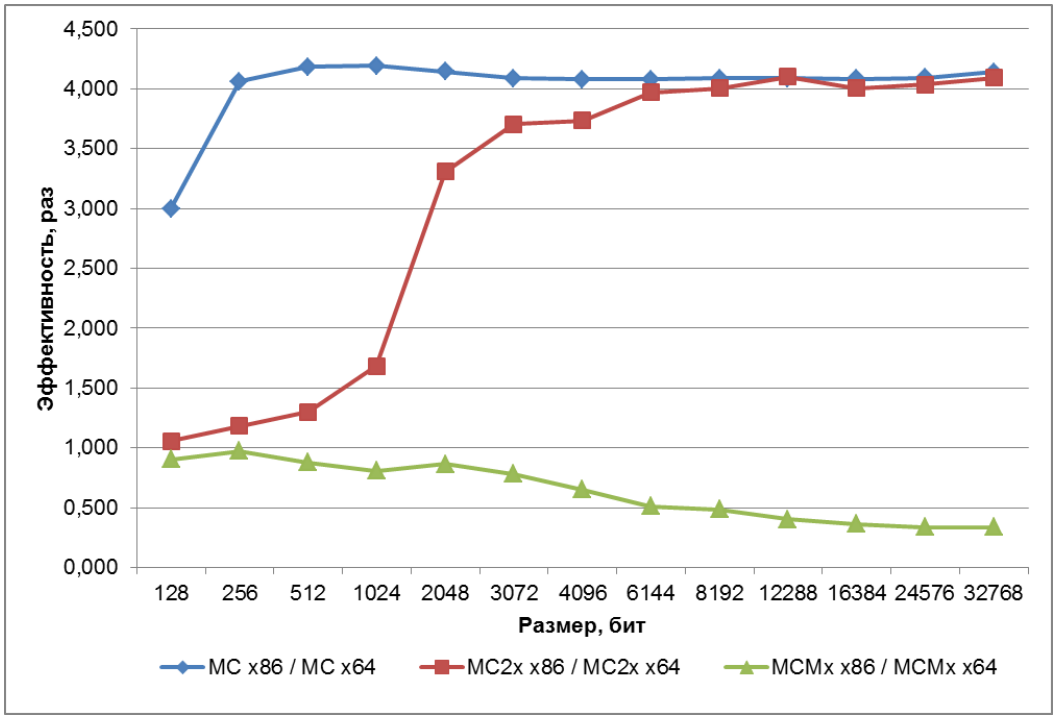

Рис.8. Преимущество x64 реализации предложенных алгоритмов перед *x86*

Выводы

На основании теоретических исследований и полученных экспериментальных данных, можно сказать, что предложенный комбинированный подход к вопросу о повышении быстродействия операции умножения в криптосистемах с открытым ключом, полностью себя оправдал.

1. Использование отложенного переноса позволяет добиться преимущества над существующими алгоритмами. В сочетании с использованием специальных команд и типов данных, а также технологий распараллеливания, удалось добиться 14-кратного преимущества (***MCMx x86***) над *GMP*.

2. Предложенные обобщенные алгоритмы, показали свою эффективность при реализации для *w*-разрядных систем. Это было продемонстрировано при реализации для современных систем с 32-х и 64-битными машинными словами, а в будущем, по мере развития вычислительной техники, и 128-битных.

3. Наиболее перспективным является алгоритм ***MCMx***, который показывает значительно лучшие результаты по сравнению с другими представленными

алгоритмами, по этому, дальнейшие исследования будут направлены на его развитие с использованием специализированых программных и аппаратных средств (например, технологии *NVIDIA CUDA*).

Литература

1. Abusharekh A. Comparative Analysis of Software Libraries for Public Key Cryptography / A.Abusharekh, K.Gaj // Software Performance Enhancement for Encryption and Decryption, SPEED'2007. June 11-12, 2007.
2. Comba P.G. Exponentiation cryptosystems on the IBM PC // IBM Systems Journal. –Vol. 29(4), 1990. – P. 526–538.
3. Giorgi P. Comparison of Modular Arithmetic Algorithms on GPUs. / P.Giorgi, T.Izard, A.Tisserand. – Mode of access: World Wide Web. – URL: http://hal-lirmm.ccsd.cnrs.fr/lirmm-00424288/fr. – Description based on screen.
4. Großschadl J. Energy-Efficient Software Implementation of Long Integer Modular Arithmetic / J.Großschadl, R.Avanzi, E.Sava, S.Tillich // Advances in Cryptology Prociding in CHES'2005. – Springer-Verlag. 2005. LNCS 3659. P.75-90.
5. Hankerson Darrel. Guide to Elliptic Curve Cryptography. / Hankerson Darrel, Menezes Alfred J., Vanstone Scott – Springer-Verlag Professional Computing Series. – 2004. – 311 p.
6. Hong S-M. New Modular Multiplication algorithms for fast modular exponeniation / S-M.Hong, S-Y.Oh, H.Yoon // Advances in Cryptology Proceedings of Eurocrypt '96. – Springer-Verlag, 1996. – P.166-177.
7. Kovtun V. Approaches for the Parallelization of Software Implementation of Integer Multiplication [Electronic resource] / V.Kovtun, A.Okhrimenko // Cryptology ePrint Archive. – Report 2012/482. – 2012. – Mode of access: World Wide Web. – URL: http://eprint.iacr.org/2012/482.pdf. — Description based on screen.
8. Paar C. Implementation options for finite filed arithmetic for elliptic curve cryptosystems [Electronic resource]. – Worchester Polytechnic Institute. – ECC'99, 1999. – Mode of access: World Wide Web. – URL: http://www.ece.wpi.edu/research/crypto.html. – Description based on screen.
9. Schneier B. Applied cryptography: protocols, algorithms, and source code in C, Wiley-India, 2007. – 784 p.
10. Ковтун В.Ю. Подходы к повышению производительности программной реализации операции умножения в поле целых чисел / Ковтун В.Ю., Охрименко А.А., Нечипорук В.В. // – Защита информации. − №1 (54). – 2012. − С. 68-75.