УДК 004.056.55

В. Ю. КОВТУН[1] *канд. техн. наук, А. А. ОХРИМЕНКО*[2]

ПОДХОДЫ К РАСПАРАЛЛЕЛИВАНИЮ ПРОГРАММНОЙ РЕАЛИЗАЦИИ ОПЕРАЦИИ УМНОЖЕНИЯ ЦЕЛЫХ ЧИСЕЛ

Введение. Информатизация общества ведет к увеличению роли систем защиты информации, которые немыслимы без криптографических преобразований, среди которых особое место занимают криптографические преобразования с открытым ключом.

Криптографические преобразования с открытым ключом имеют уже продолжительную историю: от публикации Диффи и Хеллмана [1], которая заложила фундамент, до современных криптосистем, например, на алгебраических кривых. Среди актуальных задач дальнейшего развития криптосистем с открытым ключом, выделяют повышение производительности программной и аппаратной реализации. На протяжении всего периода развития криптопреобразований с открытым ключом, их основу составляли операции в кольцах и полях чисел, среди которых, умножение занимает особое, ведущее, место. Таким образом, увеличения скорости преобразований с открытым ключом можно достигнуть, увеличив быстродействие операции умножения в кольцах и полях чисел.

Программная реализация любого алгоритма напрямую зависит от архитектуры аппаратной платформы. Развитие микропроцессорной техники до некоторого времени шло в сторону увеличения тактовой частоты процессоров, однако, после достижения физического предела частоты, акцент был сделан на увеличение числа потоков обработки команд. На сегодняшний день процессоры обладают от 2-х до 10-и физических ядер, однако эффективность использования всех вычислительных возможностей процессоров всецело ложиться на разработчика, который проектирует и программирует алгоритмы.

В связи с этим, актуальность задачи адаптации существующих алгоритмов на процессорах с несколькими потоками выполнения команд не вызывает сомнений.

Задача распараллеливания алгоритмов арифметических операций известна давно [2, 3]. В этих работах рассматриваются алгоритмы умножения Монтгомери и целочисленной арифметики для реализации на NVIDIA GPGPU [2, 3]. Дальнейшее развитие данного направления для других алгоритмов умножения, позволит найти наиболее эффективные техники распараллеливания для различных аппаратных платформ.

Известны следующие технологии распараллеливания:

- OpenMP [4, 5] для процессоров общего назначения.
- OpenCL [6] для процессоров общего назначения и графических процессорах NVIDIA и AMD.
- Intel Threading Building Block [7] для процессоров общего назначения.
- NVIDIA CUDA [8] для графических процессорах общего назначения NVIDIA.
- AMD Accelerated Parallel Processing (APP) [9] графических процессорах общего назначения AMD.

Далее рассмотрим алгоритмы умножения целых чисел, а также подходы к их распараллеливанию с помощью технологии OpenMP. Технология OpenMP была выбрана не случайно – она поддерживается большинством современных компиляторов языка C++ для различных аппаратных платформ, а также в связи с ее доступностью, простотой и наглядностью. Другие технологии являются более громоздкими и менее наглядными, при этом суть предложенного подхода к распараллеливанию остается неизменной.

Modified Comba. Ранее, в работе [11], авторами был предложен модифицированный алгоритм Comba [10] – Modified Comba, в котором используется подход отложенного переноса. Использование 64-х разрядных переменных для хранения 32-х разрядных

[1] National Aviation University of Ukraine. E-mail: vladislav.kovtun@gmail.com
[2] National Aviation University of Ukraine. E-mail: andrew.okhrimenko@gmail.com

переменных, позволило избавиться от учета переноса из 32-х разрядной переменной, после каждой арифметической операции.

Перенос накапливался в старшей части 64-х разрядной переменной и может быть учтен при необходимости (рис. 1). Алгоритм Modified Comba [11] приведен ниже.

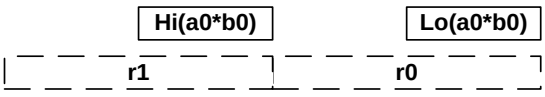

Рис. 1 Идея механизма отложенного переноса

Алгоритм Modified Comba. Умножение целых [11].

Input: *integers* $a, b \in \mathrm{GF}(p)$, $w = 32$, $n = \log_{2^w} a$, $nk = 2n - 1$.

Output: integer $c = a \cdot b$.

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. $c_k^{(32)} \leftarrow \mathrm{low}_{(32)}\left(r_0^{(64)}\right),\ r_0^{(64)} \leftarrow \mathrm{low}_{(32)}\left(r_1^{(64)}\right),\ r_1^{(64)} \leftarrow \mathrm{low}_{(32)}\left(r_2^{(64)}\right),\ r_2^{(64)} \leftarrow 0.$
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. $c_k^{(32)} \leftarrow \mathrm{low}_{(32)}\left(r_0^{(64)}\right),\ r_0^{(64)} \leftarrow \mathrm{low}_{(32)}\left(r_1^{(64)}\right),\ r_1^{(64)} \leftarrow \mathrm{low}_{(32)}\left(r_2^{(64)}\right),\ r_2^{(64)} \leftarrow 0.$
4. $c_{nk}^{(32)} \leftarrow \mathrm{low}_{(32)}\left(r_0^{(64)}\right).$
5. Return (c).

Проведем краткий анализ алгоритма Modified Comba [11], а также укажем его основные отличия от прототипа – алгоритма Comba [10], также более детально остановимся на его потенциальных возможностях.

На рис. 2 и 3 приведена графическая интерпретация алгоритма Modified Comba для $n = 3$, где четко прослеживается сложение результатов соответствующих произведений по столбцам.

Изложенная в алгоритме Modified Comba [11] идея отложенного переноса, натолкнула авторов на возможность параллельно выполнить сложение значений в столбцах $r_0 = \sum_{k=0}^{2n-1} Lo\left(a_i \cdot b_j\right) | \, k = i + j, 0 \le i,\ j < n$ и $r_1 = \sum_{k=0}^{2n-1} Hi\left(a_i \cdot b_j\right) | \, k = i + j, 0 \le i,\ j < n$.

В классическом алгоритме Comba, этот подход неосуществим, по причине связанности операций сложения переносом из старшего разряда. Факт отсутствия переноса при сложении чисел в столбце (накоплении суммы) для Modified Comba, позволяет говорить об изолированности операции накопления суммы, что в свою очередь дает возможность выполнять цикл накопления п.2 и п.3 параллельно, в отдельных (независимых) потоках.

Заметим, что после завершения накопления суммы в 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 $c_i = Lo(r_0)$.

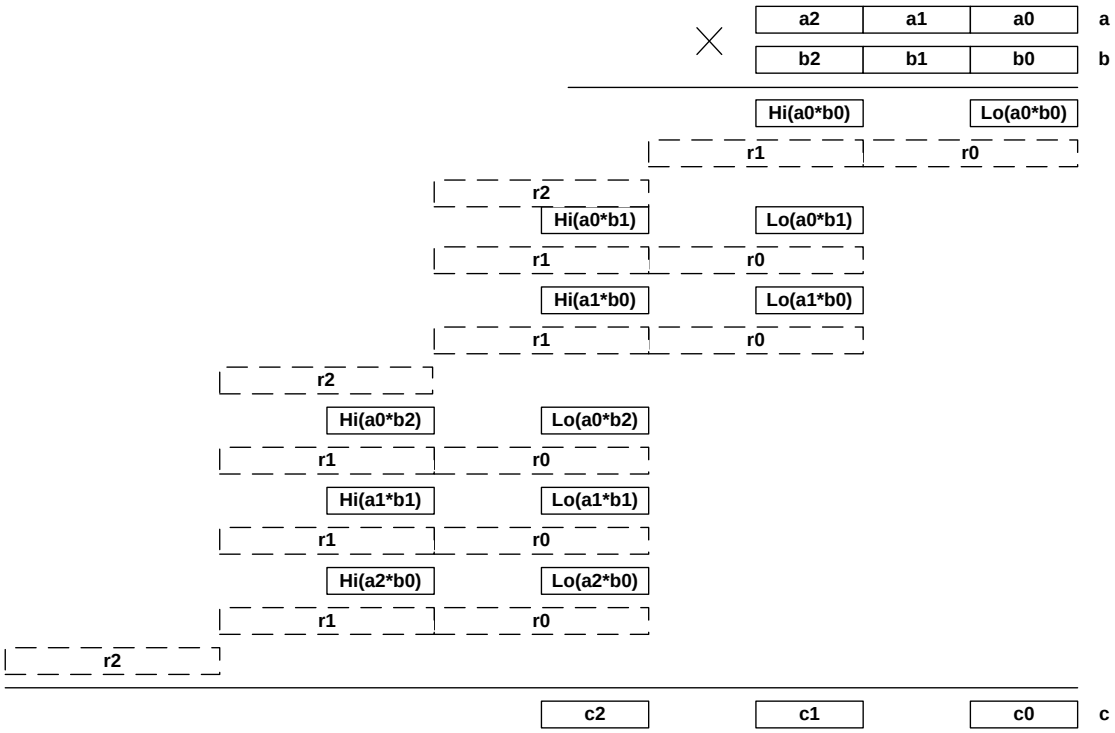

Рис. 2 Графическая интерпретация цикла 2, алгоритма Modified Comba

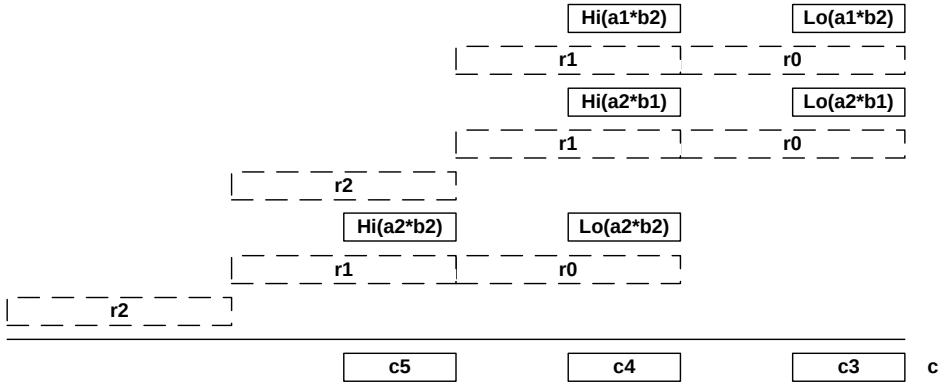

Рис. 3 Графическая интерпретация цикла 3, алгоритма Modified Comba

Описанный механизм отложенного переноса позволяет сформулировать несколько подходов к распараллеливанию алгоритма умножения Modified Comba:

- Параллельное выполнение циклов п. 2 и п. 3, с последующей коррекцией результатов (два параллельных потока). В дальнейшем будем называть его Modified Comba 2x.
- Параллельное выполнение итераций циклов п. 2 и п. 3, с последующим слиянием промежуточных результатов (множество параллельных потоков). В дальнейшем будем называть его Modified Comba Mx.

Алгоритм Modified Comba 2x. В алгоритме присутствует два цикла п. 2 и п. 3, которые производят чтение элементов $a_i^{(32)}$ и $b_j^{(32)}$ соответствующих массивов, а также запись результатов умножения в элементы $c_k^{(32)}$. Заметим, что индексы k в циклах п. 2 и п. 3 не повторяются при записи в $c_k^{(32)}$, 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 потоку 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.

Рассмотрим алгоритм Modified Comba с распараллеливанием с помощью OpenMP на два потока.

Algorithm Modified Comba 2x. Integer multiplication with OpenMP supports two threads.

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

Output: integer $c = a \cdot b$.

1. #pragma omp parallel sections private($r_0^{(64)}, r_1^{(64)}$) begin

1.1. #pragma omp section begin

1.1.1. $rl_0^{(64)} \leftarrow 0$, $rl_1^{(64)} \leftarrow 0$, $rl_2^{(64)} \leftarrow 0$.

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

1.1.2.1.2. $rl_0^{(64)} \leftarrow rl_0^{(64)} + v^{(32)}$, $rl_1^{(64)} \leftarrow rl_1^{(64)} + u^{(32)}$.

1.1.2.2. $rl_1^{(64)} \leftarrow rl_1^{(64)} + \mathrm{hi}_{(32)}\left(rl_0^{(64)}\right)$, $rl_2^{(64)} \leftarrow rl_2^{(64)} + \mathrm{hi}_{(32)}\left(rl_1^{(64)}\right)$.

1.1.2.3. $c_k^{(32)} \leftarrow \mathrm{low}_{(32)}\left(rl_0^{(64)}\right)$, $rl_0^{(64)} \leftarrow \mathrm{low}_{(32)}\left(rl_1^{(64)}\right)$, $rl_1^{(64)} \leftarrow \mathrm{low}_{(32)}\left(rl_2^{(64)}\right)$, $rl_2^{(64)} \leftarrow 0$.

1.1.3. $r_0^{(64)} \leftarrow rl_1^{(64)}$.

1.1.4. $r_1^{(64)} \leftarrow rl_2^{(64)}$.

#pragma omp section end

1.2. #pragma omp section begin

1.2.1. $rl_0^{(64)} \leftarrow 0$, $rl_1^{(64)} \leftarrow 0$, $rl_2^{(64)} \leftarrow 0$.

1.2.2. For $k \leftarrow n$, $l \leftarrow 1, k < nk$, $k{+}{+}$, $l{+}{+}$ do

1.2.2.1. For $i \leftarrow l$, $j \leftarrow k - l$, $i < n$, $i{+}{+}$, $j{-}{-}$ do

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

1.2.2.1.2. $rl_0^{(64)} \leftarrow rl_0^{(64)} + v^{(32)}$, $rl_1^{(64)} \leftarrow rl_1^{(64)} + u^{(32)}$.

1.2.2.2. $rl_1^{(64)} \leftarrow rl_1^{(64)} + \mathrm{hi}_{(32)}\left(rl_0^{(64)}\right)$, $rl_2^{(64)} \leftarrow rl_2^{(64)} + \mathrm{hi}_{(32)}\left(rl_1^{(64)}\right)$.

1.2.2.3. $c_k^{(32)} \leftarrow \mathrm{low}_{(32)}\left(rl_0^{(64)}\right)$, $rl_0^{(64)} \leftarrow \mathrm{low}_{(32)}\left(rl_1^{(64)}\right)$, $rl_1^{(64)} \leftarrow \mathrm{low}_{(32)}\left(rl_2^{(64)}\right)$, $rl_2^{(64)} \leftarrow 0$.

#pragma omp section end

#pragma omp parallel sections end

2. $r_0^{(64)} \leftarrow r_0^{(64)} + c_n^{(32)}$.

3. $r_1^{(64)} \leftarrow r_1^{(64)} + hi_{(32)}\left(rl_0^{(64)}\right) + c_{n+1}^{(32)}$.

4. $t^{(64)} \leftarrow hi_{(32)}\left(rl_1^{(64)}\right)$.

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

5.1. $t^{(64)} \leftarrow t^{(64)} + c_k^{(32)}$.

5.2. $c_k^{(32)} \leftarrow low_{(32)}\left(t^{(64)}\right)$.

5.3. $low_{(32)}\left(t^{(64)}\right) \leftarrow hi_{(32)}\left(t^{(64)}\right)$.

5.4. $hi_{(32)}\left(t^{(64)}\right) \leftarrow 0$.

6. $c_{nk}^{(32)} \leftarrow \text{low}_{(32)}\left(r_0^{(64)}\right)$.

7. Return (c).

После выполнения работы двумя параллельными потоками п. 1.1 и п. 1.2, необходимо произвести корректировку пп. 2-6 результатов работы потока п. 1.2 посредством переноса, полученного в другом потоке п. 1.1.

На рис. 4 продемонстрируем работу алгоритма с распараллеливанием на 2 потока команд для $n = 3$.

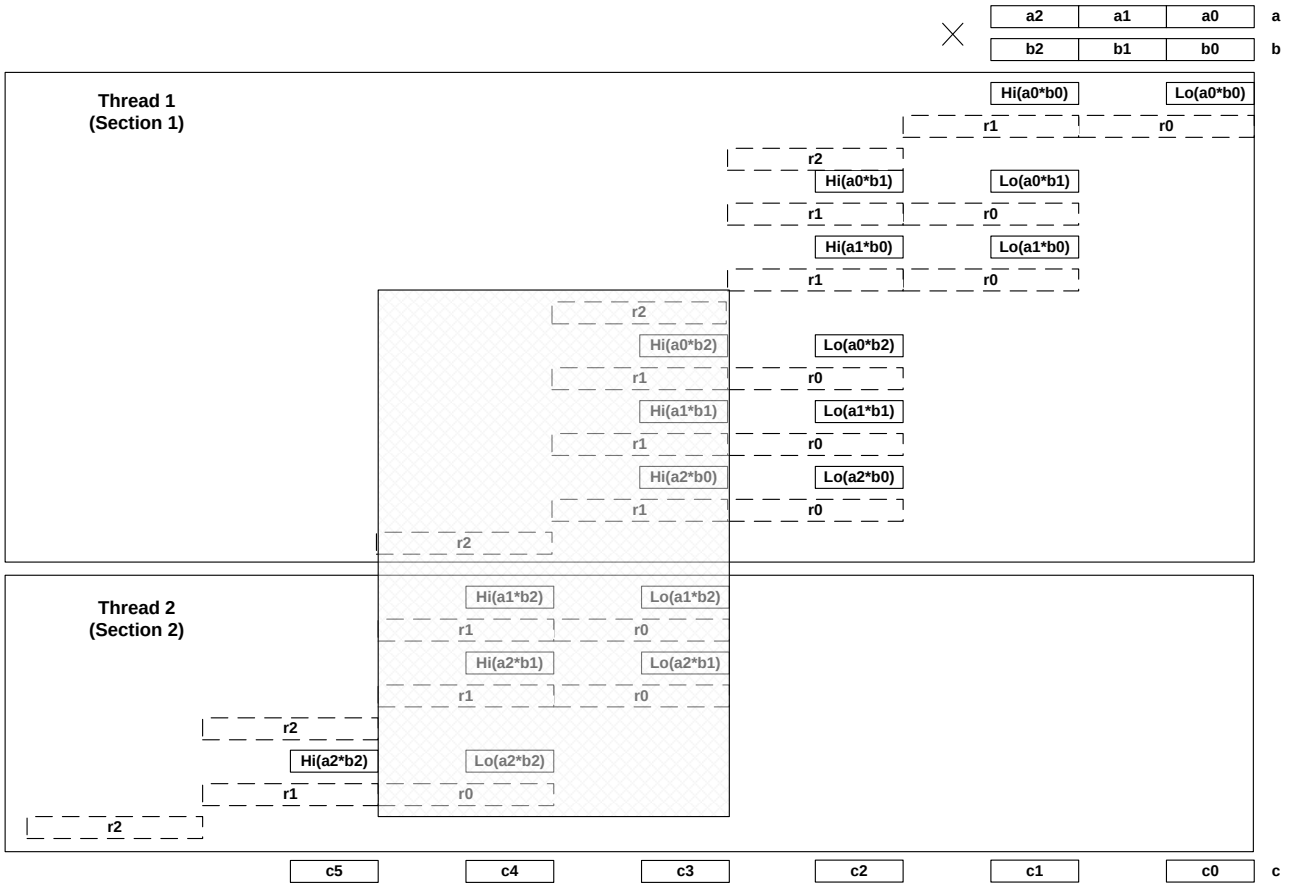

Рис. 4 Графическая интерпретация цикла 2, алгоритма Modified Comba 2x.

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

Алгоритм Modified Comba Mx. При более детальном рассмотрении алгоритма Modified Comba, 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. 2.3. 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 создано два массива $r0_i^{(64)}$ и $r1_i^{(64)}$, $i = \overline{0, 2n-1}$. Легко заметить, что такой подход позволяет легко варьировать числом параллельных потоков, без модификации алгоритма в целом.

Algorithm Modified Comba Mx. Integer multiplication with OpenMP supports multiply threads

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

Output: integers $c = a \cdot b$.

0. $l \leftarrow 1$.

1. #pragma omp parallel private ($r_0^{(64)}, r_1^{(64)}$) reduction (+ : l) begin

2. #pragma omp for nowait begin

2.1. For $k \leftarrow 0$, $k < n$, $k{+}{+}$ do

2.1.1. $rl_0^{(64)} \leftarrow 0$, $rl_1^{(64)} \leftarrow 0$.

2.1.2. For $i \leftarrow 0$, $j \leftarrow k$, $i \le k$, $i{+}{+}$, $j{-}{-}$ do

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

2.1.2.2. $rl_0^{(64)} \leftarrow rl_0^{(64)} + v^{(32)}$, $rl_1^{(64)} \leftarrow rl_1^{(64)} + u^{(32)}$.

2.1.3. $r0_k^{(64)} \leftarrow rl_0^{(64)}$, $r1_k^{(64)} \leftarrow rl_1^{(64)}$.

#pragma omp for end

3. #pragma omp for nowait begin

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

3.1.1. $rl_0^{(64)} \leftarrow 0$, $rl_1^{(64)} \leftarrow 0$, $rl_2^{(64)} \leftarrow 0$.

3.1.2. For $i \leftarrow l$, $j \leftarrow k - l$, $i < n$, $i{+}{+}$, $j{-}{-}$ do

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

3.1.2.2. $rl_0^{(64)} \leftarrow rl_0^{(64)} + v^{(32)}$, $rl_1^{(64)} \leftarrow rl_1^{(64)} + u^{(32)}$.

3.1.3. $r0_k^{(64)} \leftarrow rl_0^{(64)}$, $r1_k^{(64)} \leftarrow rl_1^{(64)}$.

3.1.4. $l{+}{+}$.

#pragma omp for end

#pragma omp parallel end

4. $rl_0^{(64)} \leftarrow r0_0^{(64)}$.

5. $rl_1^{(64)} \leftarrow r1_0^{(64)}$.

6. $c_0^{(32)} \leftarrow low_{(32)}\left(rl_0^{(64)}\right)$.

7. $rl_1^{(64)} \leftarrow rl_1^{(64)} + low_{(32)}\left(rl_0^{(64)}\right)$.

8. $rl_2^{(64)} \leftarrow hi_{(32)}\left(rl_1^{(64)}\right)$.

9. $rl_0^{(64)} \leftarrow rl_1^{(64)}$.

10. $rl_1^{(64)} \leftarrow rl_2^{(64)}$.

11. $rl_2^{(64)} \leftarrow 0$.

12. For $k \leftarrow 1$, $k < nk$, $k{+}{+}$ do

12.1. $rll_0^{(64)} \leftarrow r0_k^{(64)}$.

12.2. $rll_1^{(64)} \leftarrow r1_k^{(64)}$.

12.3. $rl_0^{(64)} \leftarrow rl_0^{(64)} + low_{(32)}\left(rll_0^{(64)}\right)$.

12.4. $rl_1^{(64)} \leftarrow rl_0^{(64)} + hi_{(32)}\left(rl_0^{(64)}\right) + hi_{(32)}\left(rll_0^{(64)}\right) + low_{(32)}\left(rll_1^{(64)}\right)$.

12.5. $rl_2^{(64)} \leftarrow rl_2^{(64)} + hi_{(32)}\left(rl_1^{(64)}\right) + hi_{(32)}\left(rll_1^{(64)}\right)$.

12.6. $c_k^{(32)} \leftarrow low_{(32)}\left(rl_0^{(64)}\right)$.

12.7. $rl_0^{(64)} \leftarrow rl_1^{(64)}$.

12.8. $rl_1^{(64)} \leftarrow rl_2^{(64)}$.

12.9. $rl_2^{(64)} \leftarrow 0$.

13. $c_{nk}^{(32)} \leftarrow \text{low}_{(32)}\left(rl_0^{(64)}\right)$.

14. Return (c).

Сравнение с другими алгоритмами. Эффективность распараллеливания оценивается сопоставлением среднего времени выполнения программной реализации предложенных параллельных алгоритмов с однопоточной версией алгоритма Modified Combo [11], для 1 млн. итераций. Замер производительности программной реализации производится для массивов 32-х разрядных слов, что позволяет оценить производительность реализаций в целом. Предложенные модификации Modified Comba 2x, Mx и сам алгоритм Modified Comba, были реализованы на C++ и скомпилированы Intel C++ Compiler XE 2011 с помощью MS Visual Studio 2005 в Release Win32 конфигурации с параметром Maximize Speed, с поддержкой SSE2. Тестирование проводилось на следующих аппаратных платформах:

- мобильных системах с CPU Intel DualCore T2130 (Microsoft Windows 7), Intel Core2 Duo T7200 (Microsoft Windows XP) и AMD A8-3510 MX (Microsoft Windows 7)
- на настольной системе среднего уровня с CPU Intel Core2 Duo E6400 (Microsoft Windows 7);
- высокопроизводительной настольной системе с CPU Intel Core i7 2600 (Microsoft Windows 7).

Все из перечисленных процессоров являются двуядерными с двумя потоками выполнения команд (без поддержки Hyper Threading), в отличие от AMD A8-3510 MX (MS Windows 7), который является 4-х ядерным с поддержкой 4 параллельных потоков команд и Intel Core i7 2600 (MS Windows 7), который является 4-х ядерным с поддержкой 8 параллельных потоков выполнения команд. Результаты замеров производительности различных программных реализаций для 1 млн. умножений и вычислительных систем, для заданной длины массива 32-х разрядных слов, приведем в таблице 1.

Таблица 1

CPU	Algorithm	Count of 32-bit words / ms											
		3	4	8	16	32	48	64	96	128	192	256	384
Intel Dual Core T2130	Cmb*	16	16	46	187	702	1544	2652	5786	10246	22988	40206	90002
	Cmb* 2x	202	202	219	328	546	966	1591	3758	5381	13194	20508	49173
	Cmb* Mx	281	297	343	464	889	1544	2417	5332	8234	17935	33203	72519
Intel Core2 Duo T7200	Cmb*	16	16	46	156	484	1015	1734	3766	6719	14703	26063	57735
	Cmb* 2x	234	172	187	235	422	703	1125	2157	3641	7812	13531	29859
	Cmb* Mx	266	281	297	406	719	1235	1844	3593	6015	12891	22500	49625
Intel Core2 Duo E6400	Cmb*	0	16	31	94	328	688	1172	2515	4500	9843	17438	38610
	Cmb* 2x	78	93	93	125	266	453	719	1672	2438	5265	9547	20281
	Cmb* Mx	141	141	172	250	453	781	1218	2735	4047	8688	16000	34578
AMD A83510 MX	Cmb*	16	16	31	156	452	921	1576	3682	6537	14212	24133	51480
	Cmb* 2x	187	187	209	250	421	687	1061	2075	3416	7472	13072	28205
	Cmb* Mx	359	375	390	437	593	826	1185	2075	3276	6755	11404	24804
Intel Core i7-2600	Cmb*	0	16	16	78	249	546	936	2044	3525	5554	13884	30872
	Cmb* 2x	124	109	109	156	266	484	764	1622	2605	3635	9734	21902
	Cmb* Mx	172	156	203	234	312	421	609	1139	1794	7831	6334	13089

Попытаемся проанализировать результаты экспериментов, приведенные в таблице 1. Так, на платформе с процессором Dual Core T2130, над однопоточной реализацией Cmb*, свое превосходство показал Cmb* 2x на числах длиной в 32 слова, и Cmb* Mx на числах длиной в 48 слов. Отметим, что Cmb* Mx показал значительно хуже производительность, чем Cmb* 2x. Процессор Core2 Duo T7200 оказался значительно производительнее Dual Core T2130, в связи с этим, однопоточную реализацию Cmb*, смогли превзойти Cmb* 2x на числах длиной в 32 слова, а Cmb* Mx на числах длиной в 96 слов. Как и в предыдущем случае, Cmb* Mx показал значительно хуже производительность, чем Cmb* 2x. Такое поведение легко объясняется более высокой производительностью ядер Core2 Duo T7200, в сравнении с Dual Core T2130.

Особого внимания заслуживает мобильный процессор AMD A83510 MX, который показал, превосходство Cmb* 2x на числах длиной в 32 слова, а Cmb* Mx на числах длиной в 48 слов, кроме того, реализация Cmb* Mx превзошла Cmb* 2x на числах длиной в 96 слов. Данные результаты позволяют смело заявить, что Cmb* Mx может эффективно применяться на многопроцессорных/ многоядерных вычислительных системах.

Перейдем к рассмотрению результатов экспериментов на настольных системах.

Система Core2 Duo E6400, показала схожее поведение с Core2 Duo T7200: над однопоточной реализацией Cmb*, свое превосходство показал Cmb* 2x на числах длиной в 32 слова, и Cmb* Mx на числах длиной в 128 слов. Такое поведение также объясняется более высокой производительностью ядер Core2 Duo E6400, в сравнении с Core2 Duo T7200.

Наилучше показатели производительности оказались у системы с процессором Core i7-2600, так благодаря высокой производительности ядер, превосходство Cmb* 2x и Cmb* Mx над Cmb*, проявилось лишь на числах длиной в 48 слов. При этом эффект от выполнения Cmb* Mx на 4-х ядрах в 8-ем потоков проявился в явном превосходстве не только над Cmb*, но и над Cmb* 2x. Дальнейшее увеличение длинны чисел показало значительное превосходство Cmb* Mx над Cmb* 2x.

Из таблицы 1 видно, что эффект от распараллеливания начинает проявляться на числах длиной более 32 слов. Это связано с достаточно большими накладными расходами, которые вызваны созданием нового потока в рамках одной операции умножения, учитывая тот факт, что эти накладные расходы соизмеримы с расходами на само умножение. Избавиться от такого нежелательного эффекта можно, если параллельные потоки создавать заранее, перед выполнением всех арифметических операций, на этапе инициализации всей библиотеки.

Также, следует обратить внимание на другой эффект: чем выше производительность одного ядра, тем на большей длине числа проявляется эффект от распараллеливания, об этом свидетельствуют результаты замеров на процессорах Intel Core i7-2600 и AMD A83510 MX.

Заключение. Полученные результаты экспериментов и теоретических исследований, выполненных в рамках данной работы, позволяют сделать **следующие выводы**:

- Предложенный авторами алгоритм Modified Comba может быть эффективно распараллелен, так на процессоре Intel Core i7-2600 удалось добиться превосходства в 1,5 раза для Modified Comba 2x и в более чем 2 раза для Modified Comba Mx, в сравнении с однопоточной реализацией.
- Эффект от распараллеливания начинает проявляться при длине в 32 слова по 32 бита (1024 бита), что говорит о значительных накладных расходах на создание параллельного потока, компенсацию которых следует закладывать непосредственно в саму библиотеку арифметических преобразований на этапе инициализации.
- Проведенные исследования показали, что компиляторы GNU gcc C++ (Linux Debian 6.0 x64) и MS Visual Studio C++ (Windows XP, Windows 7) показали себя значительно хуже, чем Intel C++ Compiler XE 2011 (Windows 7), т.к. скомпилированные ими программные реализации, показали значительно худшую производительность, поэтому не приводятся в работе. Худшая производительность обусловлена большим временем, которое тратиться

программной реализацией на создание параллельного потока и временем задержки перед уничтожением потока после его отработки.

- В качестве дальнейшего направления работы видится применение подхода распараллеливания к другим алгоритмам арифметических операций в кольцах и полях для повышения производительности криптографических операций с отрытым ключом, например на алгебраических кривых. О необходимости проведения данных исследований говорят результаты полученные авторами работы [12] по использованию технологии CUDA при реализации криптосистемы на эллиптических кривых.

Список литературы: 1. *Diffie W., Hellman M. E.*, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, 1976. 2. *Selçuk Baktir and Erkay Sava*. Highly-Parallel Montgomery Multiplication for Multi-core General-Purpose Microprocessors. // *Cryptology ePrint Archive*. –Report 2012/140. – 2012. –16 p. Available at: http://eprint.iacr.org. 3. *Pascal Giorgi, Thomas Izard, Arnaud Tisserand*. Comparison of Modular Arithmetic Algorithms on GPUs // In Proc. International Conference on Parallel Computing (ParCo 2009). -Vol19. –Lyon, France. -2009. –pp. 315-322. 4. The OpenMP API Specification for Parallel Programming. Available at: http://openmp.org. 5. OpenMP in Visual C++. Available at: http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx. 6. Khronos OpenCL API Registry. URL: http://www.khronos.org/registry/cl/. 7. Intel Threading building blocks for open source. URL: http://threadingbuildingblocks.org/. 7. NVIDIA CUDA. URL: http://www.nvidia.ru/object/cuda_home_new_ru.html. 8. AMD Accelerated Parallel Processing (APP). URL: http://developer.amd.com/sdks/ AMDAPPSDK/samples/showcase/Pages/default.aspx. 9. *Comba P. G*. Exponentiation cryptosystems on the IBM PC // *IBM Systems Journal*. –Vol. 29(4). -1990. -pp. 526–538. 10. *Kovtun V., Okhrimenko A.* Approaches for the performance increasing of software implementation of integer multiplication in prime fields // *Cryptology ePrint Archive*. –Report 2012/170. –2012. –9 p. Available at: http://eprint.iacr.org. 11. *Giorgi P. Izard T, Tisserand A.* Comparison of Modular Arithmetic Algorithms on GPUs. URL: http://hal-lirmm.ccsd.cnrs.fr/lirmm-00424288/fr/

Национальный авиационный университет. Поступила в редколлегию