Vladislav Kovtun[1], Andrew Okhrimenko[2]

Approaches for the Parallelization of Software Implementation of Integer Multiplication

In this paper there are considered several approaches for the increasing performance of software implementation of integer multiplication algorithm for the 32-bit & 64-bit platforms via parallelization. The main idea of algorithm parallelization consists in delayed carry mechanism using which authors have proposed earlier [11]. The delayed carry allows to get rid of connectivity in loop iterations for sums accumulation of products, which allows parallel execution of loops iterations in separate threads. Upon completion of sum accumulation threads, it is necessary to make corrections in final result via assimilation of carries. First approach consists in optimization of parallelization for the two execution threads and second approach is an evolution of the first approach and is oriented on three and more execution threads. Proposed approaches for parallelization allow increasing the total algorithm computational complexity, as for one execution thread, but decrease total execution time on multi-core CPU.

Keywords: multiplication, integers, parallelization, OpenMP, OpenCL, software implementation, cryptographic transformations, public key cryptosystem.

Introduction. The IT penetration of society life leads to increasing of information security role, which are unthinkable without cryptographic cryptosystems. Public key cryptosystems hold a special place among the cryptographic transformations.

Public key cryptosystems have a long history from first publication of Diffie & Hellman [1], which initiates basis to modern cryptosystems, for example algebraic curve cryptosystems. The increasing of software and hardware implementation of public key cryptosystems is the most important among topical tasks of public key cryptosystems developing.

Arithmetic operations in rings and fields of integers form the basis of public key cryptosystems the whole of germination time.

Integer multiplication operation is the main operation in ring or field arithmetic.

Thus, the increasing of performance of public key cryptosystems may be achieved via increasing performance of integer multiplication operation in ring or field.

It is well known, that software implementation of any algorithm depends on the architecture of hardware platform. Such microprocessors evolution goes to increasing a clock frequency. But upon reaching of physical limitation accent moves to increasing number of execution threads. Now, following CPU are available:

- AMD proposes CPU (Opteron 6200 Series) with 16 physical cores and 16 execution threads;
- Intel proposes CPU (Xeon 7000 Series) with 10 physical cores with Hyper-Threading and 20 execution threads.

There are two computational accelerators based on GP GPU available:

- NVIDIA proposes GP GPU (Tesla FermiM2090) with CUDA technology with 512 physical cores;
- AMD proposes GP GPU (FireStream 9370) with AMD APP technology with 20 SIMD PU and 1600 Streamed cores.

The task of efficiency using all these cores lies on software engineer which designs and programs algorithms.

In accordance with this, it is no doubt that adaptation of existing algorithms for execution on multi-core CPU (CPU with several execution threads) is an urgent task.

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

The task of algorithm parallelization of arithmetic operations for integers is not new [2, 3], in these papers there are considered Montgomery multiplication and integers arithmetic for implementation on NVIDIA GP GPU [2, 3]. Further evolution of this direction for other arithmetic algorithms allows to find more effective parallelization technique for different hardware platforms.

There are several well known parallelization techniques:

- OpenMP [4, 5] for general purpose CPU.
- OpenCL [6] for general purpose CPU and for GP GPU NVIDIA & AMD.
- Intel Threading Building Block [7] for the general purpose CPU.
- NVIDIA CUDA [8] for GP GPU NVIDIA.
- AMD Accelerated Parallel Processing (APP) [9] for GP GPU AMD.

Further we consider algorithms for integer multiplication and approaches for their parallelization with OpenMP technology. The OpenMP technology was chosen because it is supported by most modern C++ compilers for variety hardware platforms (like Intel C++ Compiler, GCC C++ Compiler & Microsoft C++ Compiler). OpenMP makes it easy to implement parallelism in existing C++ programs. Other mentioned technologies are more cumbersome and less obvious, but the main idea of proposed parallelization approach remains unchangeable.

Modified Comba. Earlier, in paper [11], authors proposed a modified algorithm Comba [10] – Modified Comba, with delayed carry technique. Usage of 64-bit variables for storing 32-bit variables allows to get rid of carry assimilation from high part of 32-bit variable after each arithmetic operation.

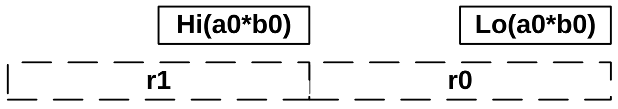

Fig. 1. Delayed carry mechanism idea

The carry accumulates in high part of 64-bit variable and may be assimilated if necessary, see Fig. 1. Modified Comba algorithm [11] is shown below.

Algorithm Modified Comba. Integer multiplication [11].

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

Let’s conduct a brief analysis of Modified Comba algorithm [11], and show main difference with prototype – Comba algorithm [10] as well as work out in details Modified Comba potentialities.

On fig. 2 and 3 graphical visualization of Modified Comba Algorithm for $n = 3$ is shown, where the addition of corresponding products is clearly traced in columns.

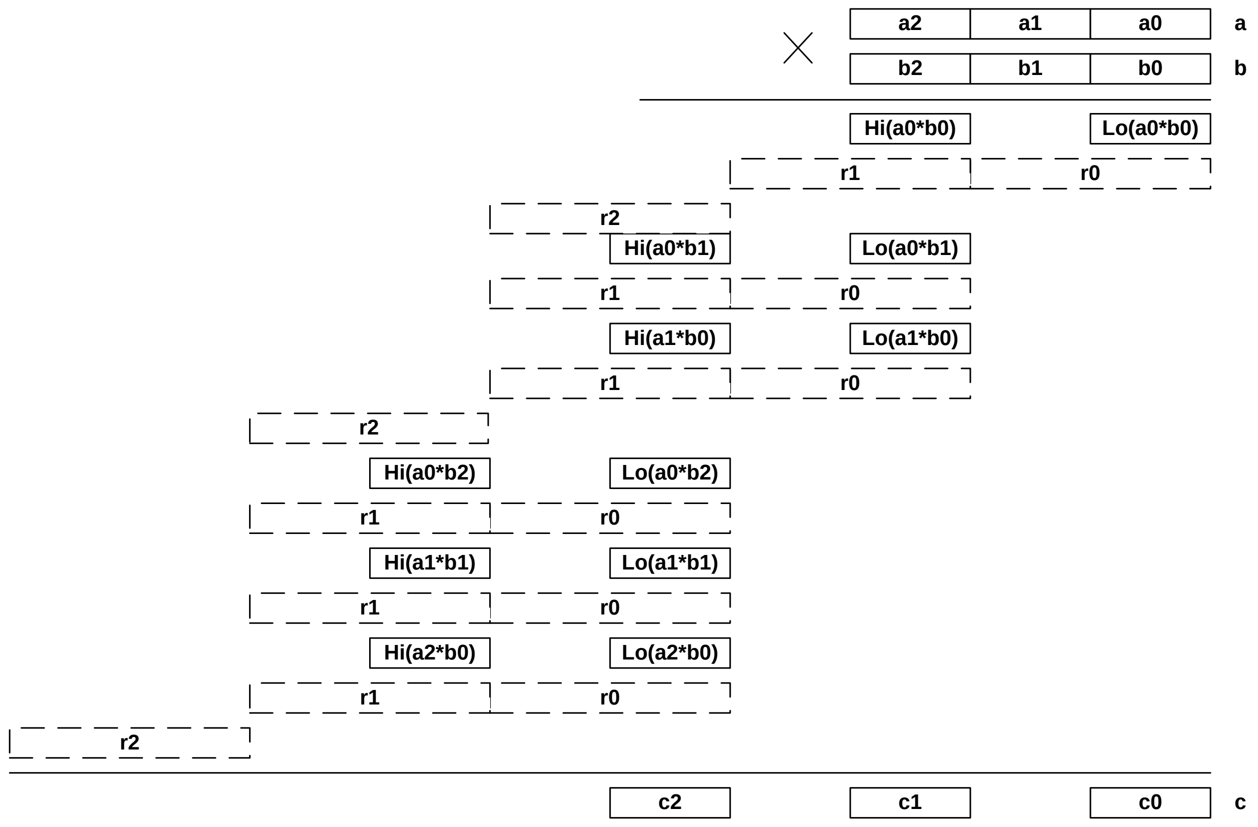

Fig. 2. Graphical visualization of loop 2 in Modified Comba Algorithm

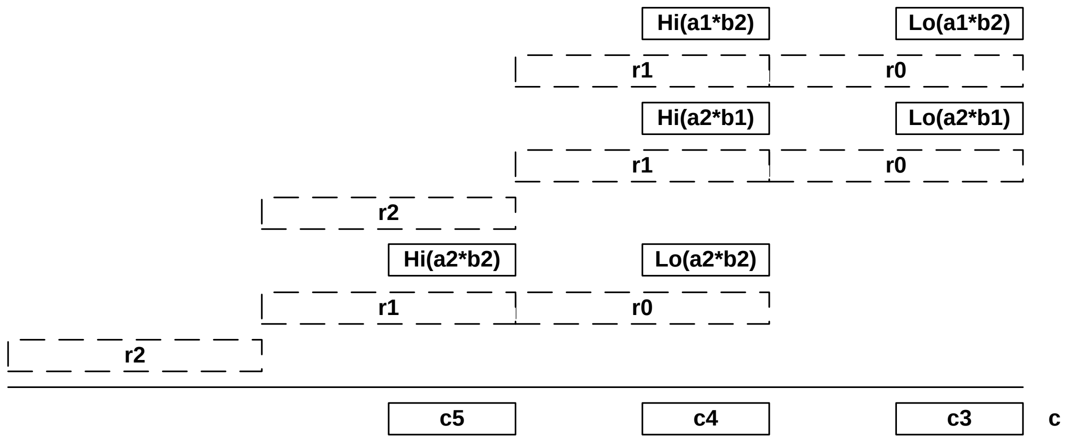

Fig. 3. Graphical visualization of loop 3 in Modified Comba Algorithm

The idea of delayed carry, previously described in Modified Comba Algorithm [11], has prompted authors on possibility of parallel addition values in columns $r_0 = \sum_{k=0}^{2n-1} \mathrm{Lo}\left(a_i \cdot b_j\right) | \, k = i + j, 0 \le i, \, j < n$ and $r_1 = \sum_{k=0}^{2n-1} \mathrm{Hi}\left(a_i \cdot b_j\right) | \, k = i + j, 0 \le i, \, j < n$. In the classical Comba algorithm, this approach is impossible, due to the fact that addition operations with carry have connectedness.

The fact of carry absence in additions integers in column (sum accumulation) for Modified Comba Algorithm allowing to say about isolatedness of sum accumulation operation, which allows to execute accumulation loop on step 2 and 3 in parallel independent threads.

Notice, after the completion of sum accumulation in all independent threads, still it is required to make an adjustment (to assimilate a carry) $r_1 = r_1 + \mathrm{Hi}(r_0)$, $r_2 = r_2 + \mathrm{Hi}(r_1)$ and compute result $c_i = \mathrm{Lo}(r_0)$.

The delayed carry mechanism allows formulating several approaches to the Modified Comba Algorithm parallelization:

- Parallel execution (in two parallel threads) of loops in the step 2 and 3 with further final result correction. We will call it Modified Comba 2x.
- Parallel execution (number of parallel threads) of iterations in loops on steps 2 and 3 with further intermediate results (from parallel threads) merging. We will call it Modified Comba Mx.

Modified Comba 2x algorithm. The algorithm contains two loops on step 2 and 3, which read elements $a_i^{(32)}$ and $b_j^{(32)}$ of corresponding arrays with further writing results of multiplication $a_i^{(32)}$ and $b_j^{(32)}$ to elements $c_k^{(32)}$. Note, indexes k in loops on step 2 and 3 are not repeated while writing to elements $c_k^{(32)}$. This allows to say about data independence in these loops and possibility of parallel loops execution by the parallel technique. It is worth underlining that, both loops on step 2 and 3 use common temporary variables r_0, r_1 and r_2. Moreover, variables r_0 and r_1 keep values which use in loop in step 3 after the loop on step 2 is complete.

Thus, after the finishing loop on step 3 it will require making a correction of results of loop execution in step 3 – to take into account results of execution loop in step 2 (results keep in temporary variables r_0, r_1 and r_2). See, while loops on steps 3 and parallelization, each thread should work with its own private temporary variables rl_0, rl_1 and rl_2.

Global variables r_0 and r_1 will be used only for the possible carry from rl_0, rl_1 loop on step 2 to further correction of results accumulation in loop on step 3.

Let's consider Modified Comba algorithm with parallelization via OpenMP into two threads.

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

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

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. $r_1^{(64)} \leftarrow r_1^{(64)} + \mathrm{hi}_{(32)}\left(rl_0^{(64)}\right) + c_{n+1}^{(32)}$.
4. $t^{(64)} \leftarrow \mathrm{hi}_{(32)}\left(rl_1^{(64)}\right)$.
5. For $k \leftarrow n + 2$, $k < nk$, $k{+}{+}$ do
5.1. $t^{(64)} \leftarrow t^{(64)} + c_k^{(32)}$.
5.2. $c_k^{(32)} \leftarrow \mathrm{low}_{(32)}\left(t^{(64)}\right)$.
5.3. $\mathrm{low}_{(32)}\left(t^{(64)}\right) \leftarrow \mathrm{hi}_{(32)}\left(t^{(64)}\right)$.
5.4. $\mathrm{hi}_{(32)}\left(t^{(64)}\right) \leftarrow 0$.
6. $c_{nk}^{(32)} \leftarrow \mathrm{low}_{(32)}\left(r_0^{(64)}\right)$.
7. Return $(c)$.
```

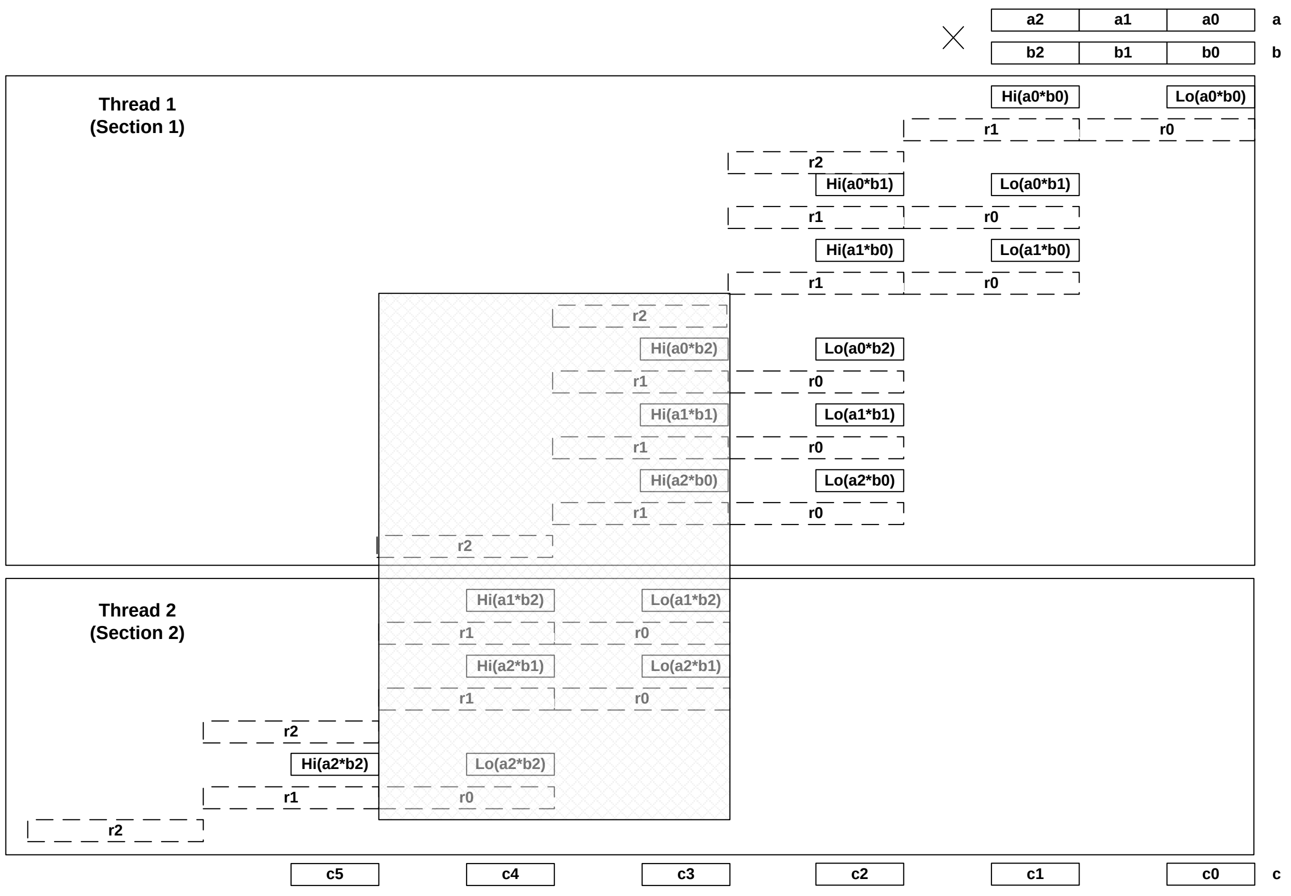

Fig. 4. Graphical interpretation of loop 2 in Modified Comba 2x Algorithm

After the finishing execution of the parallel threads in step 1.1 and 1.2, it is required to correct in steps 2-6 the result of thread in step 1.2 via carry transfer from result in other thread in step 1.1.

The algorithm Modified Comba 2x for 2 threads and $n=3$ shown on fig. 4.

An algorithm with parallelization on multiply threads deserves special attention. It is described below.

Modified Comba Mx algorithm. In detailed consideration of Modified Comba algorithm, it is easy to note, that iterations in loops in steps 2 and 3 do not depend on one another.
The exclusions are addition accumulation results and carry from current iteration to only that addition and carry results after the current iteration to further iteration processing in steps 2.2 and 2.3. By entering individual local variables for the sum accumulation parallel sum accumulation in iteration on steps 2 and 3 may be correctly performed.

For these purposes in algorithm Modified Comba Mx two arrays $r0_i^{(64)}$ и $r1_i^{(64)}$, $i=\overline{0,2n-1}$. Are declared. Easy to see, that this approach allows to variate number of parallel threads without algorithm modification in whole.

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

```
Input: integers $a, b \in \mathbf{GF}(p)$, $w=32$, $n=\log_{2^w} a$, $nk=2n-1$.
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. $c_0^{(32)} \leftarrow \mathrm{low}_{(32)}\left(rl_0^{(64)}\right)$.
7. $rl_1^{(64)} \leftarrow rl_1^{(64)} + \mathrm{low}_{(32)}\left(rl_0^{(64)}\right)$.
8. $rl_2^{(64)} \leftarrow \mathrm{hi}_{(32)}\left(rl_1^{(64)}\right)$.
```

```
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. $rl_0^{(64)} \leftarrow rl_0^{(64)} + \mathrm{low}_{(32)}\left(rll_0^{(64)}\right)$.
12.4. $rl_1^{(64)} \leftarrow rl_0^{(64)} + \mathrm{hi}_{(32)}\left(rl_0^{(64)}\right) + \mathrm{hi}_{(32)}\left(rll_0^{(64)}\right) + \mathrm{low}_{(32)}\left(rll_1^{(64)}\right)$.
12.5. $rl_2^{(64)} \leftarrow rl_2^{(64)} + \mathrm{hi}_{(32)}\left(rl_1^{(64)}\right) + \mathrm{hi}_{(32)}\left(rll_1^{(64)}\right)$.
12.6. $c_k^{(32)} \leftarrow \mathrm{low}_{(32)}\left(rl_0^{(64)}\right)$.
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. $c_{nk}^{(32)} \leftarrow \mathrm{low}_{(32)}\left(rl_0^{(64)}\right)$.
14. Return $(c)$.
```

Comparison with other algorithms. Parallelization efficiency may be evaluated by the comparison of average time execution of software implementations for proposed parallel algorithms with single thread Modfied Comba algorithm [11], for one million iterations for different integers bit-length.

Performance measurement of software implementation is performed for the arrays with 32-bit machine words, which allows to estimate performance of implementations as a whole.

The Modified Comba 2x, Mx and single thread Modified Comba are implemented in C++ and compiled with Intel C++ Compiler XE 2011 in Microsoft Visual Studio 2005 in Release Win32 target with Maximize Speed option and SSE2 supports.

While testing several hardware platforms are used:

- Entry-level old mobile CPU – Intel Dual Core T2130 on Microsoft Windows 7 x86.
- Middle-level old mobile CPU – Intel Core2 Duo T7200 on Microsoft Windows XP x86.
- High-level modern mobile CPU – AMD A8-3510 MX on Microsoft Windows 7 x86-64.
- Middle-level old desktop CPU – Intel Core2 Duo E6400 on Microsoft Windows 7 x86.
- High-level modern desktop CPU – Intel Core i7 2600 on Microsoft Windows 7 x86.

All CPU have two cores with two execution threads without Hyper-Threading only that:

- AMD A8-3510 MX has 4 cores and 4 execution threads (without Hyper-Threading).
- Intel Core i7 2600 has 4 cores and 8 execution threads (with Hyper-Threading).

In table 1 there are shown the performance measurement results for different software implementations for one million multiplications and different CPU for specified arrays with 32-bit words length.

Table 1. Performance of different implementations of integer multiplication algorithms in ms

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

Let's try to analyze the results of experiments shown in table 1. The low-end CPU Dual Core T2130 showed superiority two thread implementation Cmb* 2x over single thread implementation Cmb* on 32 word long integers and Cmb* Mx over single thread implementation Cmb* on 48 word Long integers. Note, Cmb* Mx showed much worse performance than Cmb* 2x.

CPU Core2 Duo T7200 has better performance then CPU Dual Core T2130, in accordance with this Cmb* 2x leaded Cmb* on integers with length of 32 words and Cmb* Mx leaded Cmb* on 48 word long integers. Note, as in previous case, Cmb* Mx showed worst performance then Cmb* 2x. This behavior may be clearly explained via higher core performance of Core2 Duo T7200 in comparison with Dual Core T2130.

Special attention should be paid to the mobile CPU AMD A83510 MX, which shows the superiority Cmb* 2x on 32 word long integers and Cmb* Mx on 48 word long integers. In addition, the implementation of Cmb* Mx excels the Cmb* 2x on 96 words long integers. These results allow us to safely say, what Cmb* Mx may be efficiently used in computers with multiprocessors and multi-core CPUs.

Let us now consider the results of experiments in desktops. The desktop with Core2 Duo E6400 shows the comparatively results with Core2 Duo T7200: Cmb* 2x shows the superiority over single thread implementation of Cmb* on integers with length in 32 words and Cmb* Mx shows the superiority over single thread implementation of Cmb* on integers with length in 128 words. This behavior may be explained by the higher core performance of Core2 Duo E6400 in comparison with Core2 Duo T7200.

The desktop with Core i7-2600 CPU shows the best results in comparison with other: so due to high performance of cores the superiority of Cmb* 2x and Cmb* Mx over single thread Cmb* shows on integers with length in 48 words.

The effect of execution Cmb* Mx on CPU with 4 cores and 8 execution threads appears in explicit superiority not only on Cmb* but also on Cmb* 2x. A further increasing the integers length showed a significant superiority Cmb* Mx over Cmb* 2x.

Table 1 clearly shows that the effect of parallelism begins to appear during multiplication of integers with length more than 32 words. This is connected with significant expenses in new work thread creation in one multiplication operation. These expenses are commensurable with expenses) on multiplication by itself. This undesirable effect may be avoided via preliminary working thread creation, before it carries out all arithmetic operations on library initialization stage.

As well, it is worth to observe that: the effect from parallelization appears on the greater bit-length of integer if faster CPU is used. It is evidenced by the results of measurements on CPU: Intel Core i7-2600 and AMD A83510 MX.

Conclusion. Results of experiments and theoretical investigations, which made in this paper, allow saying about next **conclusions**:

1. Modified Comba algorithm proposed by authors may be efficiently parallelized. Modified Comba 2x algorithm in 1.5 times and Modified Comba Mx algorithm in 2 times is in excess of single thread algorithm.
2. The parallelization benefits appear on 1024 bit (on 32 word with 32-bit), this allow to say about large expenses on new parallel thread creation. These expenses may be compensated in the arithmetic library on initialization stage.
3. The OpenMP implementation (support) in GNU gcc C++ Compiler on Debian Linux 6.0 x86-64 and Microsoft C++ Compiler in Visual Studio 2005, 2008 and 2010 on Windows XP x86 and Windows 7 x86-64 appear to be much worse than Intel C++ Compiler XE 2011 on Windows 7 because C++ programs have much worse performance (they are not described in this work).

In software implementation, large time on new thread creation and large time on delay before thread destruction are mainly responsible for the worse performance (GNU gcc C++ Compiler on Debian Linux 6.0 x86-64 and Microsoft C++ Compiler in Visual Studio 2005, 2008 and 2010).

Further, authors see application of parallelization technique to other arithmetic operation algorithms in rings and fields such as reduction and inversion for the enhancing performance of public key cryptosystems, like cryptosystems on algebraic curves.

The necessity of these researches speaks results obtained by the authors of [12] via using CUDA technology in the implementation of elliptic curve cryptosystem.

Bibliography

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/
8. NVIDIA CUDA. URL: http://www.nvidia.ru/object/cuda_home_new_ru.html
9. AMD Accelerated Parallel Processing (APP). URL: http://developer.amd.com/sdks/AMDAPPSDK/samples/showcase/Pages/default.aspx
10. Comba P. G. Exponentiation cryptosystems on the IBM PC // IBM Systems Journal. –Vol. 29(4). -1990. -pp. 526–538.
11. 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.
12. Giorgi P. Izard T, Tisserand A. Comparison of Modular Arithmetic Algorithms on GPUs. URL: http://hal-lirmm.ccsd.cnrs.fr/lirmm-00424288/fr/