Research Article

Techniques for Performance Improvement of Integer Multiplication in Cryptographic Applications

Robert Brumnik,[1] Vladislav Kovtun,[2] Andrew Okhrimenko,[2] and Sergii Kavun[3]

[1] *ICS-CIS, Center for Information Security, 1000 Ljubljana, Slovenia*

[2] *Department of Information Security, National Aviation University, Kiev 03058, Ukraine*

[3] *Department of Information Technologies, Kharkiv Institute of Banking of the University of Banking of the National Bank of Ukraine, Kharkiv 61174, Ukraine*

Correspondence should be addressed to Vladislav Kovtun; vladislav.kovtun@gmail.com

The problem of arithmetic operations performance in number fields is actively researched by many scientists, as evidenced by significant publications in this field. In this work, we offer some techniques to increase performance of software implementation of finite field multiplication algorithm, for both 32-bit and 64-bit platforms. The developed technique, called "delayed carry mechanism," allows to preventing necessity to consider a significant bit carry at each iteration of the sum accumulation loop. This mechanism enables reducing the total number of additions and applies the modern parallelization technologies effectively.

1. Introduction

The cryptographic transformations with public key are revolutionized from Diffie and Hellman consideration to modern algebraic curves cryptosystems [1]. However, transformations have stayed permanent—with operations in the number field $GF(p)$. The integer multiplication takes a special place in number field operations; see Figure 1. One of the urgent problems of public key cryptosystem improvements is an increase of software performance and hardware implementation. One of the approaches to increasing cryptosystems performance is the increasing of the performance of finite field arithmetic in multiplication operations.

The problem of the speedup of arithmetic operation in number fields is actively researched by many scientists, as evidenced by significant publications in this area [2–11]. Except the arithmetic operations algorithms, it is interesting to look at/study approaches to the architecture of software

libraries [12–21] with field operations, which allow decreasing overheads on field operations in whole.

Publications analysis [2–10] enabled extracting the most effective multiplication algorithms, Comba [2, 3] and Karatsuba [3, 8, 10]. However, the Comba algorithm shows better results in tests performance (benchmark) of software implementations on modern platforms [3–9]. Karatsuba-Comba described multiplication (KCM) algorithm for the RISC processors in the article [8]. The KCM algorithm is an interesting symbiosis of Comba and Karatsuba algorithms, where Karatsuba algorithm is specially used for machine word multiplication. As a result, the main goal of this paper is to provide a suggestion for the effective increasing of software implementation of finite field number $GF(p)$ multiplication (squaring) via well-known Comba algorithm [2, 3, 8]. Such researches were caused by the necessity of effective confirmation of software implementation of known algorithms for continuous development of modern 32-bit and

Cryptographic transformations		Encryption/decryption	Digital signature generation and verification	Key exchange
Arithmetic in elliptic curve point group		Scalar multiplication of elliptic curve point		
		Point addition		Point doubling
Arithmetic in finite field	Multiplication	Addition	Subtraction	Squaring	Inversion
CPU commands		mov, mul, shr, shl, add, sub . . .		

FIGURE 1: Operation hierarchy of elliptic curve cryptosystem.

a	1 0010010110100110101110011011011 10110101100010010011011011010000

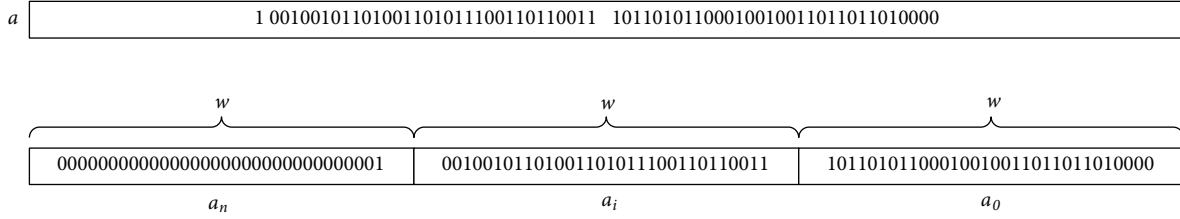| w	w	w
00000000000000000000000000000001	00100101101001101011100110110011	10110101100010010011011011010000
a_n	a_i	a_0

FIGURE 2: Representation of large integers.

64-bit platforms. It is important to mention that last ten years have seen much development in the direction of the multicore CPU and multi-CPU systems [8, 9].

2. Multiplication Algorithm-Prototype Description and Its Modification

Let us begin by introducing some notation and basic definitions. Carry is a digit that is transferred from one column of digits to another column of more significant digits during a calculation algorithm; w is machine word size and n is the number of machine words required to store a large integer. We present large integers (multipliers) as a set of n machine words; see Figure 2. For example, if we have 65-bit integer, we need three 32-bit machine words to store it.

The Comba algorithm [2] is based on main loops p. 2 and p. 3 and nested loops p. 2.1 and p. 3.1 (Algorithm 1). In the low level of hierarchy, in loops p. 2.1 and p. 3.1 we will compute 64-bit integer product $(uv)^{(2w)}$ which consists of two 32-bit integers $u^{(w)}$ and $v^{(w)}$.

The sum accumulation occurs in 32-bit temporary variables r_0, r_1, and r_2, on each iteration p. 2.1.2 and p. 2.1.3.

The final result of the assignment is temporary variables r_0, r_1, and r_2 which are changing at each iteration on p. 2.2.

Comba's algorithm main drawbacks are as follows.

(i) In nested loops p. 2.1 and p. 3.1 there is a sum accumulation with carry in 32-bit temporary variables r_0, r_1, and r_2, p. 2.1.2, p. 2.1.3 and p. 3.1.2, p. 3.1.3:

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

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

In this case there are 3 additions of 32-bit integer (includes 2 additions with carry) and 3 assignments of 32-bit variables

r_0, r_1 and r_2. The sum accumulation with carry takes place in each iteration of loop p. 2.1.

(ii) In nested loops p. 2.1 and p. 3.1, for the sum accumulation, for 32-bit variables r_0, r_1, and r_2 the transfers are considered using the assembler code for the implementation of addition operation with carry. This does not allow pairing and parallelizing [22]; therefore we observe an ineffective processor resource usage.

(iii) Loops p. 2 and p. 3 cannot be effectively parallelized due to high internal linkage code because of carry consideration.

It is easy to obtain a computational complexity for Comba's algorithm:

$$
\begin{aligned}
I_{\text{mul}}^{\text{Comba}} &= 4I_{\text{assign}}^{w} + \left(\frac{n+1}{2}n + \frac{1+n-1}{2}(n-1) \right) \\
&\quad \times \left(1I_{\text{mul}}^{w} + 4I_{\text{add}}^{w} + 6I_{\text{assign}}^{w} \right) + 4(2n-1)I_{\text{assign}}^{w} \\
&= 4I_{\text{assign}}^{w} + n^2 \left(1I_{\text{mul}}^{w} + 4I_{\text{add}}^{w} + 6I_{\text{assign}}^{w} \right) \\
&\quad + 4(2n-1)I_{\text{assign}}^{w},
\end{aligned} \tag{1}
$$

where I_{assign}^{w} is an assignment operation of 32-bit integers, I_{add}^{w} is an addition operation of 32-bit integers, and I_{mul}^{w} is a multiplication operation of 32-bit integer.

Figure 2 illustrates the drawbacks of algorithm for $n = 3$ and its impact on computational complexity of algorithm.

Modern CPUs allow the use of 64-bit data types and operations to achieve better performance, but the algorithm is not adapted for their use.

In the upper part of the figure, there are two big coefficients a and b represented by three 32-bit integers $a = (a_2, a_1, a_0)$ and $b = (b_2, b_1, b_0)$, where a_i and b_i have a machine-word bit size. Algorithm iterations are presented

Input: integers $a, b \in GF(p)$, $w = 32$ bit, $n = \log_{2^w} a$.
Output: $c = a \cdot b$
(1) $r_0^{(w)} \leftarrow 0, r_1^{(w)} \leftarrow 0, r_2^{(w)} \leftarrow 0$.
(2) For $k \leftarrow 0, k < n, k++$ do
(2.1) For $i \leftarrow 0, j \leftarrow k, i \leq k, i++, j--$ do
(2.1.1) $(uv)^{(w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}$.
(2.1.2) $r_0^{(w)} \leftarrow r_0^{(w)} + v^{(w)}, r_1^{(w)} \leftarrow r_1^{(w)} + u^{(w)} + \text{carry}, \text{carry} \leftarrow 0$.
(2.1.3) $r_2^{(w)} \leftarrow r_2^{(w)} + \text{carry}, \text{carry} \leftarrow 0$.
(2.2) $c_k^{(w)} \leftarrow r_0^{(w)}, r_0^{(w)} \leftarrow r_1^{(w)}, r_1^{(w)} \leftarrow r_2^{(w)}, r_2^{(w)} \leftarrow 0$.
(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) $(uv)^{(w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}$.
(3.1.2) $r_0^{(w)} \leftarrow r_0^{(w)} + v^{(w)}, r_1^{(w)} \leftarrow r_1^{(w)} + u^{(w)} + \text{carry}, \text{carry} \leftarrow 0$.
(3.1.3) $r_2^{(w)} \leftarrow r_2^{(w)} + \text{carry}, \text{carry} \leftarrow 0$.
(3.2) $c_k^{(w)} \leftarrow r_0^{(w)}, r_0^{(w)} \leftarrow r_1^{(w)}, r_1^{(w)} \leftarrow r_2^{(w)}, r_2^{(w)} \leftarrow 0$.
(4) $c_{nk}^{(w)} \leftarrow r_0^{(w)}$.
(5) Return (c).

ALGORITHM 1: Comba's integer multiplication.

Input: integers $a, b \in GF(p)$, $w = 32$ bit, $n = \log_{2^w} a$, $nk = 2n - 1$.
Output: $c = a \cdot b$
(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 $i \leftarrow 0, j \leftarrow k, i \leq k, i++, j--$ do
(2.1.1) $(uv)^{(2w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}$.
(2.1.2) $r_0^{(2w)} \leftarrow r_0^{(2w)} + v^{(w)}, r_1^{(2w)} \leftarrow r_1^{(2w)} + u^{(w)}$
(2.2) $r_1^{(2w)} \leftarrow r_1^{(2w)} + \text{hi}_{(w)}\left(r_0^{(2w)}\right), r_2^{(2w)} \leftarrow r_2^{(2w)} + \text{hi}_{(w)}\left(r_1^{(2w)}\right)$
(2.3) $c_k^{(w)} \leftarrow \text{low}_{(w)}\left(r_0^{(2w)}\right), r_0^{(2w)} \leftarrow \text{low}_{(w)}\left(r_1^{(2w)}\right), r_1^{(2w)} \leftarrow \text{low}_{(w)}\left(r_2^{(2w)}\right), r_2^{(2w)} \leftarrow 0$.
(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) $r_0^{(2w)} \leftarrow r_0^{(2w)} + v^{(w)}, r_1^{(2w)} \leftarrow r_1^{(2w)} + u^{(w)}$
(3.2) $r_1^{(2w)} \leftarrow r_1^{(2w)} + \text{hi}_{(w)}\left(r_0^{(2w)}\right), r_2^{(2w)} \leftarrow r_2^{(2w)} + \text{hi}_{(2)}\left(r_1^{(2w)}\right)$
(3.3) $c_k^{(w)} \leftarrow \text{low}_{(w)}\left(r_0^{(2w)}\right), r_0^{(2w)} \leftarrow \text{low}_{(w)}\left(r_1^{(2w)}\right), r_1^{(2w)} \leftarrow \text{low}_{(w)}\left(r_2^{(2w)}\right), r_2^{(2w)} \leftarrow 0$.
(4) $c_{nk}^{(w)} \leftarrow \text{low}_{(w)}\left(r_0^{(2w)}\right)$.
(5) Return (c).

ALGORITHM 2: Modified Comba's integer multiplication.

under the solidus. It should be noted that Comba's algorithm implements well-known long multiplication technique, with a small difference where the multiplier part a_i $i = \overline{1, n}$ multiplies all parts of other multipliers b_j $j = \overline{1, n}$, in case of fulfillment condition $(i + j = k)$ (in columns).

Such approach leads not to strings addition (multiplication of intermediate results) as long multiplication but to columns addition. That allows finding a part of resulting product c_i (under the solidus). Each multiplication is accompanied by the sum accumulation, as shown in Figure 3.

The computational complexity for $n = 3$ will be

$$I_{\text{mul}}^{\text{Comba}} = 4I_{\text{assign}}^w + 9\left(1I_{\text{mul}}^w + 4I_{\text{add}}^w + 6I_{\text{assign}}^w\right) \\ + 20I_{\text{assign}}^w = 78I_{\text{assign}}^w + 9I_{\text{mul}}^w + 36I_{\text{add}}^w. \tag{2}$$

In the following steps of calculation procedure we eliminate the drawbacks.

(i) The modern 32-bit CPUs effectively implement the addition operations of 32-bit and 64-bit integers, using 32-bit CPUs commands. That allows implementing a carry accumulation by the addition of 32-bit variables in 64-bit variable-accumulator that obviate the need for carry accounting and correction requirements after the addition of variables r_0, r_1, and r_2. An accumulated carry will be accounted in the final iterations in the loops in p. 2 and p. 3.

(ii) Modern CPUs have multicore architecture that allows them to execute several instruction flows at the
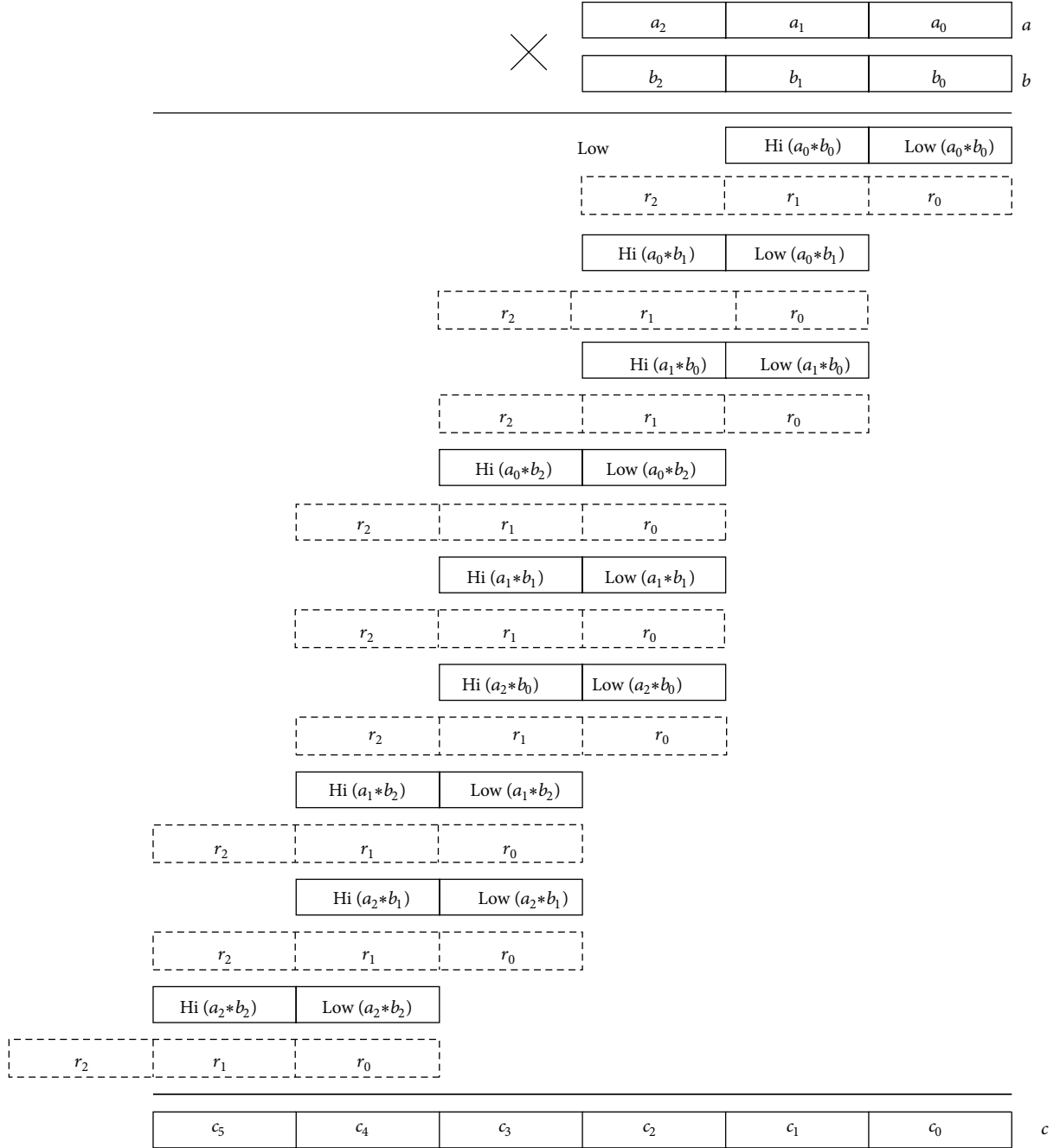
FIGURE 3: Graphic interpretation of Comba's algorithm.

same time. This property brings to parallel iterations execution in loops p. 2 and p. 3 by the OpenMP library [22–24].

The following notations are introduced in Algorithm 2.

(i) Variable $t^{(2w)}$ is used to denote 64-bit variables, $t^{(w)}$ is used to denote 32-bit variables;

(ii) Operation $\mathrm{hi}_{(w)}(t^{(2w)})$ is used to extract 32 the most significant bits in 64-bit variable, and operation

$\mathrm{low}_{(w)}(t^{(2w)})$ is used to extract 32 the least significant bits in 64-bit variable.

It is not difficult to get a computational complexity of modified Comba's algorithm:

$$
\begin{aligned}
I_{\mathrm{mul}}^{\mathrm{Mod.Comba}} = {} & 4I_{\mathrm{assign}}^{2w} + \left(\frac{n+1}{2}n + \frac{1+n-1}{2}(n-1) \right) \\
& \times \left(1I_{\mathrm{mul}}^{w} + 2I_{\mathrm{add}}^{2w|w} + 2I_{\mathrm{assign}}^{2w} \right) \\
& + (2n-1)\left(2I_{\mathrm{add}}^{2w|w} + 1I_{\mathrm{assign}}^{2w} + 1I_{\mathrm{assign}}^{w} \right)
\end{aligned}
$$
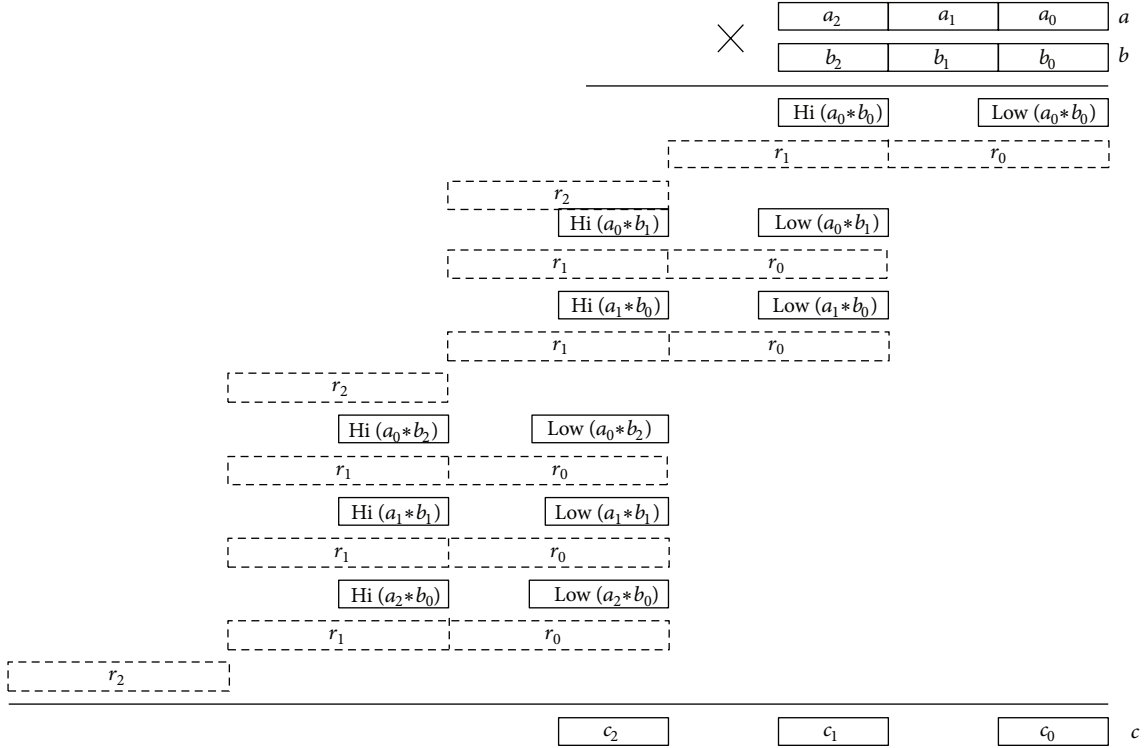
FIGURE 4: Graphic interpretation of loop 2 in modified Comba's algorithm.
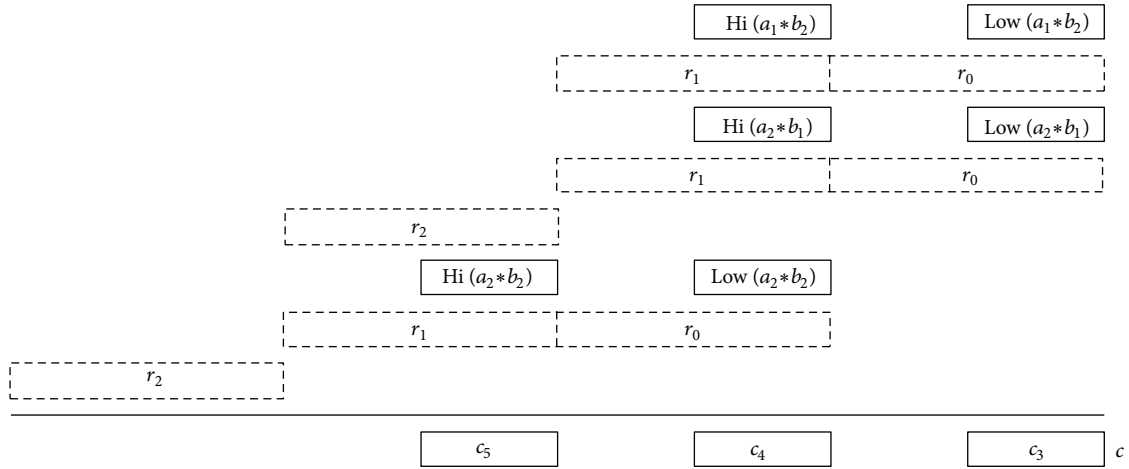
FIGURE 5: Graphic interpretation of loop 3 in modified Comba's algorithm.

$$
\begin{aligned}
&= 4I_{\text{assign}}^{2w} + n^2 \left(1I_{\text{mul}}^{w} + 2I_{\text{add}}^{2w|w} + 2I_{\text{assign}}^{2w} \right) \\
&\quad + (2n - 1) \left(2I_{\text{add}}^{2w|w} + 1I_{\text{assign}}^{2w} + 1I_{\text{assign}}^{w} \right),
\end{aligned}
\tag{3}
$$

where I_{assign}^{w} is an assignment operation of 32-bit integers, I_{assign}^{2w} is an assignment operations of 64-bit integers, I_{add}^{w} is an addition operation of 32-bit integers, $I_{\text{add}}^{2w|w}$ is an addition operation of 32-bit and 64-bit integers, and I_{mul}^{w} is a multiplication of 32-bit integers.

Figures 4 and 5 illustrate Algorithm 2 for $n = 3$; computational complexity in this case will be

$$
I_{\text{mul}}^{\text{Mod.Comba}} = 27I_{\text{assign}}^{2w} + 9I_{\text{mul}}^{w} + 28I_{\text{add}}^{2w|w} + 5I_{\text{assign}}^{w}.
\tag{4}
$$

3. Comparison with Other Algorithms

In order to provide an objective comparison of given results, the authors have made the review of well-known software math libraries [12–21] for public key cryptography. According to results review [25, 26], the software library GMP was an etalon [12]. GMP uses Karatsuba's integer multiplication

TABLE 1: Performance measurements of fields.

Field	Prime module
$GF(p82)$	50000000000000000008503491
$GF(p164)$	24999999999994130438600999402209463966197516075699
$GF(p192)$	6277101735386680763835789423 1760590137671947731828422840081
$GF(p224)$	26959946667150639794667015087019630673557916260026308143510066298881
$GF(p256)$	115792089210356248762697446949407573530086143415290314195533631308867097853951
$GF(p320)$	42719740718418201647900421592006690578364140623317241379335651938259686865762670800870819848380097
$GF(p384)$	39402006196394479212279040100143613805079739270465446667946905279627659399113263569398956308152294913554433653942643
$GF(p521)$	686479766013060971498190079908139321726943530014330540939446345918554318333976553942450577463332171975329639963713633211138647686124403803403728088892707005449

TABLE 2: Experimental results of software implementation of multiplication algorithms.

Field	Time, μs					
	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

algorithm [12]. The comparison of software implementations will be done by comparing the execution average time of software implementation of Comba and modified Comba's algorithms and implemented in GMP library for one million iterations.

To measure the algorithm performance of software implementation we can use protocols in fields of Table 1 from [27], except $GF(p82)$. These fields are recommended [3, 27] for usage in cryptographic application for different security levels. Table 1 gives a brief definition of fields and prime modules.

The proposed modified algorithm Comba and its prototype-algorithm Comba were implemented in C++, compiled with Microsoft Visual Studio 2010 in Release Win32 configuration with Maximize Speed parameter and SSE2 instruction support.

We will use the etalon library GMP v4.1.2 compiled with Microsoft Visual Studio. NET and instrumental application compiled with Microsoft Visual Studio 2010 in Win32 release configuration with Maximize Speed parameter and SSE2 instruction support.

In testing mainstream mobile platform with Intel Core i3 350 M CPU and desktop platform with Intel Pentium Dual Core E5400 were used.

Performance measurement timings for different algorithms, implementations, and CPU are shown in Table 2.

As we can see from the timing in Table 2, the proposed modification of the algorithm Comba has 1.5 times better time characteristic compared with GMP. Classic implementation of Comba's algorithm is the slowest, which is confirmed by the theoretical estimation (as it contains a larger number of additions and assignment operations). In addition, proposed software implementation of multiplication algorithms is more efficient on Dual Pentium CPU with higher frequency than on Core i3 CPU with several instruction streams. This implementation of multiplication algorithms does not support parallelization; thus, a more powerful multicore CPU Core i3 with 4 instructions processing flows does not realize their full potential.

4. Conclusions

The research resulted in the following conclusions.

(1) We ensure an increase in performance of software implementation of Comba's integer multiplication algorithm for 1.5–2 times and surpass of performance of the popular math library GMP v4.1.2, an average for 1.5 times.

(2) Modified Comba's multiplication algorithm is preferred to Karatsuba's algorithm [2] which is used in GMP library, because implementation of modified Comba's algorithm is faster than Karastuba [2] implementation in GMP for modern hardware platform (32- and 64-bit).

(3) Delayed carry mechanism allows applying different parallelization techniques to modified Comba's algorithm, for example, OpenMP [23], Intel Threading Building Blocks [28], NVIDIA CUDA [29], and OpenCL [30].

Recently, the microprocessors development increases the number of instruction processing flows. Thus, we should perform the necessity of suitable algorithms development for efficient parallelization.

NVIDIA has already proposed GPU with more than 256 cores and suitable CUDA toolkit [29] which allows creating valid multithread applications. This area is already under close monitoring and is demonstrated in publication [9, 31]. A further line of our research will focus on investigation and effective parallelization algorithms for arithmetic operations with integers.

Conflict of Interests

The authors declare that they have no conflict of interests.

References

[1] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[2] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, 1990.

[3] M. Brown, D. Hankerson, J. Lopez, and A. Menezes, "Software implementation of the NIST elliptic curves over prime fields," Research Report CORR 2000-55, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Canada, 2000.

[4] S.-M. Hong, S.-Y. Oh, and H. Yoon, "New Modular Multiplication algorithms for fast modular exponentiation," in *Advances in Cryptology—EUROCRYPT '96*, vol. 1070 of *Lecture Notes in Computer Science*, pp. 166–177, Springer, New York, NY, USA, 1996.

[5] R. M. Avanzi, "Aspects of hyperelliptic curves over large prime fields in software implementations," Report 2003/253, Cryptology ePrint Archive, 2003, http://eprint.iacr.org/2003/253.

[6] C. Paar, "Implementation options for finite field arithmetic for elliptic curve cryptosystems," in *Proceedings of the Elliptic Curve Cryptography (ECC '99)*, Worchester Polytechnic Institute, 1999.

[7] G. Gaubatz, *Versatile montgomery multiplier architectures [M.S. thesis]*, Electrical and Computer Engineering, Worcester Polytechnic Institute, 2002.

[8] J. Großschadl, R. M. Avanzi, E. Savaş, and S. Tillich, "Energy-efficient software implementation of long integer modular arithmetic," in *Proceedings of the 7th International Conference on Cryptographic Hardware and Embedded Systems (CHES '05)*, pp. 75–90, Springer, 2005.

[9] P. Giorgi, T. Izard, and A. Tisserand, "Comparison of modular arithmetic algorithms on GPUs," in *Proceedings of the International Conference on Parallel Computing (ParCo '09)*, Lyon, France, 2009.

[10] A. Weimerskirch and C. Paar, "Generalizations of the Karatsuba algorithm for efficient implementations," Report 2006/224, Cryptology ePrint Archive, 2006, http://eprint.iacr.org/2006/224.

[11] L. Judge, S. Mane, and P. Schaumont, "A hardware-accelerated ECDLP with high-performance modular multiplication," *International Journal of Reconfigurable Computing*, vol. 2012, Article ID 439021, 14 pages, 2012.

[12] The GNU Multiply Precision Library (GMP), http://gmplib.org.

[13] LiDIA, https://www.cdc.informatik.tu-darmstadt.de/en/cdc.

[14] Multiprecision Unsigned Number Template Library (MUNTL), http://mktmk.narod.ru/eng/muntl/muntl.htm.

[15] "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks," http://discovery.csc.ncsu.edu/software/TinyECC.

[16] Galois Field Arithmetic Library, http://www.partow.net/projects/galois/.

[17] "MPFQ: Fast Finite Fields Library," http://mpfq.gforge.inria.fr/.

[18] BBNUM, http://www.iw-net.org/index.php?title=Bbnum_library.

[19] "FLINT: Fast Library for Number Theory," http://www.flintlib.org.

[20] Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL), http://indigo.ie/~mscott.

[21] "LibTom Projects: LibTomMath," TomsFastMath, http://libtom.org.

[22] "Intel 64 and IA-32 Architectures Optimization Reference Manual," Order Number: 248966-025, http://www.cs.princeton.edu/courses/archive/fall13/cos217/reading/ia32opt.pdf.

[23] The OpenMP API Specification for Parallel Programming, http://openmp.org/wp/openmp-specifications/.

[24] OpenMP in Visual C++, http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx.

[25] A. Abusharekh and K. Gaj, "Comparative analysis of software libraries for public key cryptography," in *Proceedings of the Software Performance Enhancement for Encryption and Decryption (SPEED '2007)*, June 2007.

[26] P. Giorgi, L. Imbert, and T. Izard, "Multipartite modular multiplication," http://hal.archives-ouvertes.fr/lirmm-00618437/fr/.

[27] National Institute of Standards and Technology, "Recommended elliptic curves for federal government use," Appendix to FIPS 186-2, 2000.

[28] Intel Threading Blocks, http://software.intel.com/en-us/articles/intel-tbb.

[29] NVIDIA, "NVIDIA CUDA Programming Guide 2.0," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[30] OpenCL, "The open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl.

[31] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Cryptographic Hardware and Embedded Systems—CHES 2008*, E. Oswald and P. Rohatgi, Eds., vol. 5154 of *Lecture Notes in Computer Science*, pp. 62–78, Springer, Berlin, Germany, 2008.