Exploiting the Potential of GPUs for Modular Multiplication in ECC

Fangyu Zheng[1,2,3], Wuqiong Pan[1,2(✉)], Jingqiang Lin[1,2], Jiwu Jing[1,2], and Yuan Zhao[1,2,3]

[1] Data Assurance and Communication Security Research Center, CAS, Beijing, China

[2] State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China
wqpan@is.ac.cn

[3] University of Chinese Academy of Sciences, Beijing, China
{fyzheng,linjq,jing,zhaoyuan12}@is.ac.cn

Abstract. In traditional multiple precision large integer multiplication algorithm, the required number of additions approximates the number of multiplications needed. In some platforms, the great number of add instructions will occupy about half of computing latency in the overall implementation. In this paper, we propose a multiplication algorithm using separated multiply-add-with-carry instruction supported by NVIDIA GPUs. In the algorithm, we reorder the computational sequence, in which nearly all additions and carry flags handling can be combined with the multiplication instructions. The number of add instructions needed decreases from $O(n^2)$ in prevailing schoolbook algorithm to $O(n)$. Our resulting 256-bit modular multiplication and modular square over Mersenne prime respectively achieve 3.3837 billion and 5.9928 billion operations per second and reach 96 % of GPU hardware limitation. An elliptic curve point multiplication implementation using our algorithm achieves 43.6 % speedup compared to the existing fastest work.

Keywords: GPU · CUDA · Modular multiplication · ECC

1 Introduction

Asymmetrical cryptography is the core of modern Internet security. Widely used secure communication protocols in financial industry and e-commerce, rely on secure key exchange and digital signature algorithms such as the ECC [13,15] and RSA [20] algorithms. Unfortunately, great number of large integer modular multiplications seriously affects the performance of the algorithms, and becomes the bottleneck that restricts its wider application. To offload the costs, many researchers resort to graphics processing units (GPUs).

F. Zheng—This work was partially supported by the National 973 Program of China under award No. 2013CB338001 and the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDA06010702.

© Springer International Publishing Switzerland 2015
K.-H. Rhee and J.H. Yi (Eds.): WISA 2014, LNCS 8909, pp. 295–306, 2015.
DOI: 10.1007/978-3-319-15087-1_23

Many previous papers report performance benchmark results to demonstrate that the GPU architecture can already be used as an asymmetric cryptography workhorse, such as RSA [9,11,16,22], and ECC [1,4–6,8,22]. References [4,5,9,22] pioneered implementation of modular multiplication on CUDA. Bernstein et al. [5] implemented an 280-bit modular multiplication with single precision floating point (SPF) instructions, in which several threads compute an modular multiplication. In their follow-up work [4], they turned into integer instructions and employed single thread to handle an entire multiplication without the overhead of thread synchronization. Giorgi et al. [9] proposed a C++ library (PACE) to support modular arithmetic on an NVIDIA 9800GX2 GPU, in which the Montgomery representation of large integers is used to perform modular multiplication using the Finely Integrated Operand Scanning (FIOS) [14]. Ant*ão* et al. [1,2] used RNS (Reside Number System) to implement modular multiplication. In latest work [19], Pu et al. employed several threads to handle one multiplication with parallel RNS-based [3] computing model. The implementations above are based on generic modulus. In 2012 Bos et al. [6] accomplished modular multiplication over NIST P-224 modulus. His optimization focuses on elliptic curve point multiplication and only uses schoolbook modular multiplication algorithm [10], which is not suitable for GPUs. In this paper, we aim to fully exploit the potential of GPU for the implementation of the large integer modular multiplication over Mersenne prime [21].

Our contribution is to make full use of CUDA-featured separated multiply-add-with-carry instruction to avoid most of the add instructions and make the algorithm procedure more suitable for CUDA hardware framework. Using this instruction, we integrate nearly all additions in multiply-add instructions and very few carry flags need to be handled using extra instruction. The number of add instructions needed decreases from $O(n^2)$ in prevailing schoolbook algorithm to $O(n)$.

Directed at NVIDIA GeForce GTX Titan, a 2688-CUDA-core GPU, our resulting modular multiplication and modular square respectively reach 3.3837 billion and 5.9928 billion operations per second, reaching nearly 96 % of the GPU's limitation. For better evaluation, we also implement an elliptic curve point multiplication implementation using our algorithm which reaches 391,595 operations per second.

The paper is organized as follows. Section 2 presents the overview of NVIDIA GPU and its multiplication instruction. Section 3 describes our proposed algorithm in detail. Section 4 analyses performance of proposed algorithm and compares it with previous work. Section 5 concludes the paper.

2 Background

2.1 CUDA GPUs and Its Multiplication Instruction

Our target platform GTX Titan is a GK-110 GPU, which contains 14 streaming multiprocessors (SM). 32 threads (grouped as a *warp*) can concurrently run in a clock. Following the SIMT (Single Instruction Multiple Threads) architecture,

each GPU thread runs one instance of the kernel function. A warp may be preempted when it is stalled due to memory access delay, and the scheduler may switch the runtime context to another available warp. Multiple warps of threads are usually assigned to one SM for better utilization of the pipeline of each SM. These warps are called one *block* [18].

Multiplication instruction of NVIDIA GPU has a unique feature: when calculating the 32-bit $\times$ 32-bit multiplication, the whole 64-bit product cannot be obtained using one instruction, but requires 2 independent instructions: one is for lower-32-bit half, the other for upper-32-bit half. Although the whole multiplication (`mul.wide`) is provided which is used in [12,17], it is a virtual instruction but not the native instruction, which will be broken into 2 instructions (`mul.lo.` and `mul.hi`) when running on the GPUs.

Ten work patterns of multiplication (or multiply-add) instruction are supported by NVIDIA GPUs. Among them, the separated multiply-add-with-carry instruction $s =$ `madc.cc{.lo,.hi}`(a, b, c) can multiply 32-bit integer a and b, extract lower or upper half of the 64-bit product, and add a third value c with CF (carry flag) bit in the condition code register (CC). The 32-bit result and the carry that it produces will be respectively written to s and the CF bit in CC [18].

3 Proposed Algorithm Description

3.1 Large Integer Multiplication Algorithm

Traditional large integer multiplication algorithm is introduced in *Guide to Elliptic Curve Cryptography* [10]. Bos et al. [6] used this algorithm to accomplish modular multiplication. However, this algorithm is not well supported in CUDA due to its separated multiplication instruction. Through experiment, we found that, when compiling, $2(m-1)(n-1)$ add instructions are needed, whose number approximates the number of multiply instructions.

Based on this observation, we make attempt to minimize the number of add instructions. As mentioned in Sect. 2.1, CUDA platform supports separated multiply-add-with-carry instruction. Taking advantage of this character, we propose a brand new algorithm. In the proposed algorithm, the computational sequence is carefully scheduled to utilize separated multiply-add-with-carry instructions and reduce extra handling of carry flags. We take 5-word $\times$ 5-word multiplication for example to demonstrate the overall computational sequence, which is shown in Fig. 1, where $A[0:4] = \sum_{i=0}^{4} a_i 2^{ri}$ and $B[0:4] = \sum_{i=0}^{4} b_i 2^{ri}$ are the multiplicands, $C[0:9] = \sum_{i=0}^{9} c_i 2^{ri}$ is the product, r stands for word length.

The upper part of Fig. 1 indicates the adjusted computational sequence. We accumulate each row into the product $C[0:9]$ from the top to the bottom. For convenience, we color each row by white or gray. The white rows contain only the lower-half instructions, while, the gray rows include only the upper-half.

The lower part of Fig. 1 takes the last 2 rows to demonstrate the detailed step of the accumulation. In each row, from left to right, we accumulate the

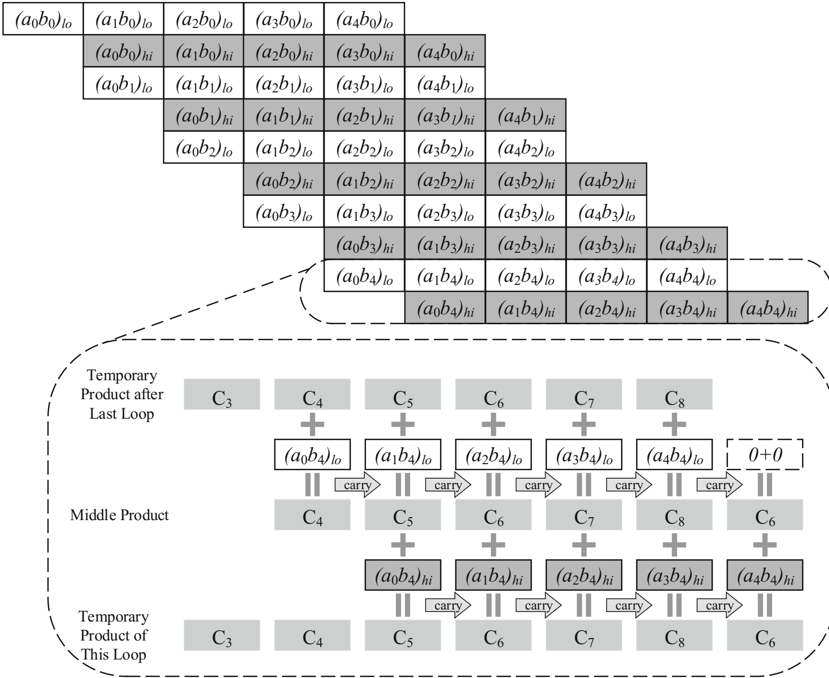

Fig. 1. $5 - word \times 5 - word$ multiplication structure using proposed algorithm

half product into $C[0:9]$. Firstly, we accumulate the white row. Using instruction `madc.cc`, each cell in the white row accumulates its lower half product into the corresponding word of $C[0:9]$, also accumulates the CF bit that previous instruction produces (except the first instruction), then writes back its carry to CF bit. The CF bit that the last instruction produces needs to be stored using add-with-carry instruction `addc`(0,0). The accumulation for the gray row is similar, but we do not need to handle the CF bit that the last instruction produces because it will not produce carry. Note that there are n white rows totally. Among them, the 0th white row need no temporary variable, because its last instruction adds zero thus will not produce carry. Therefore, $2mn$ multiplication and $(n-1)$ add instructions are needed in total. Detail is shown in Algorithm 1.

3.2 Large Integer Square Algorithm

Compared with multiplication, square operation has a natural advantage: some sub-products can be reused. Equation below shows this advantage in detail, which takes $(A[0:n-1])^2$ for example, where $A[0:n-1] = \sum_{i=0}^{n-1} a_i 2^{ri}$.

Algorithm 1. Proposed Multiplication Algorithm(r bits per word)

Input:

m-word-length Multiplicand, $A[0:m-1]=\sum_{i=0}^{m-1} a_i 2^{ri}$;
n-word-length Multiplier, $B[0:n-1]=\sum_{i=0}^{n-1} b_i 2^{ri}$;
$m > n$

Output:

(m+n)-word-length Product, $C[0:m+n-1] = A[0:m-1] \times B[0:n-1] = \sum_{i=0}^{m+n-1} c_i 2^{ri}$;

```
1: C[0 : m + n − 1] = 0
2: for i = 0 to n − 1 do
3:    for j = 0 to m − 1 do
4:       set CF = 0
5:       c_{i+j} = madc.cc.lo(a_j, b_i, c_{i+j})
6:    end for
7:    if i ≠ 0 then
8:       c_{i+n} = addc(0, 0)
9:    end if
10:   set CF = 0
11:   for j = 0 to m − 1 do
12:      c_{i+j+1} = madc.cc.hi(a_j, b_i, c_{i+j+1})
13:   end for
14: end for
15: return C[0 : m + n − 1];
```

$$
\begin{aligned}
(A[0:n-1])^2 = & \sum_{0\le i<j\le n-1} [2^{r(i+j)}(a_i \times a_j)_{lo} + 2^{r(i+j+1)}(a_i \times a_j)_{hi}] \\
& + \sum_{0\le i\le n-1} [2^{2ri}(a_i^2)_{lo} + 2^{r(2i+1)}(a_i^2)_{hi}] \\
& + \sum_{0\le j<i\le n-1} [2^{r(i+j)}(a_i \times a_j)_{lo} + 2^{r(i+j+1)}(a_i \times a_j)_{hi}] \\
= & 2 \sum_{0\le i<j\le n-1} [2^{r(i+j)}(a_i \times a_j)_{lo} + 2^{r(i+j+1)}(a_i \times a_j)_{hi}] \\
& + \sum_{0\le i\le n-1} [2^{2ri}(a_i^2)_{lo} + 2^{r(2i+1)}(a_i^2)_{hi}]
\end{aligned}
$$

In this way, only $(\frac{n(n-1)}{2}+n)\times 2 = n^2+n$ multiplications are needed.

For convenience, we take 5-word square operation for example. Firstly, we calculate $\sum_{0\le i<j\le n-1}^{n}[2^{r(i+j)}(a_i \times a_j)_{lo} + 2^{r(i+j+1)}(a_i \times a_j)_{hi}]$. In upper part of Fig. 2, the 2 white pieces are corresponding to it. We choose the upper separated piece to calculate. Before calculation, we carry out a transformation for it. As demonstrated in the middle part of Fig. 2, we "flatten" it into the new structure. In this structure, we can accumulate each row into the product from the top to the bottom, which is similar with the operation in previous section. And in each row, we accumulate the product from left to right.

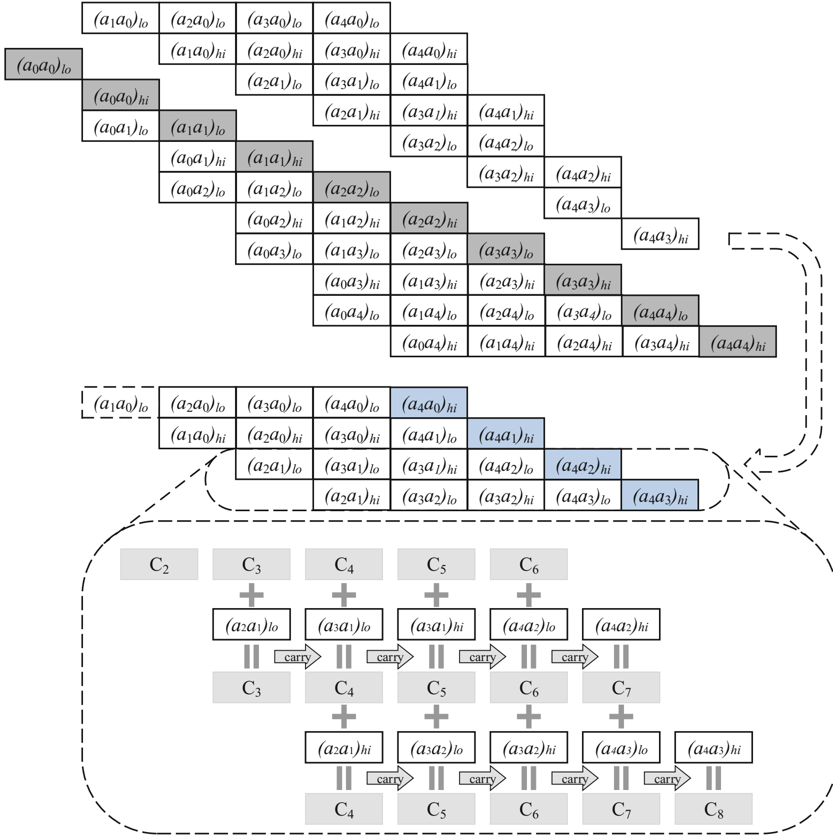

Fig. 2. $5 - word \times 5 - word$ square structure using proposed algorithm

The lower part of Fig. 2 takes the last 2 rows to demonstrate the detailed step of the accumulation. Using instruction `madc.cc`, each cell in the row accumulates its lower or upper half product into the corresponding word of $C[0:9]$, also accumulates the CF bit that previous instruction produces (except the first instruction), then writes back its carry to CF bit. CF bit that the last instruction produces needs no extra handling, because that if the last instruction of each row is upper-word multiply-add instruction, it will not produce carry. It can be proved as below.

Proof. Assuming a, b is 32-bit integer, their upper half product is at most $2^{32}-2$, because

$$(a \times b)_{hi} \leq \lfloor \frac{(2^{32}-1) \times (2^{32}-1)}{2^{32}} \rfloor = 2^{32} - 2 \tag{1}$$

We take a row as a unit to employ mathematical induction.

(1) Obviously, the last instruction of row 0 will not produce carry, because it adds with zero.

(2) Assume the last instruction of row $k-1$ does not produce carry. Thus it affects only the $(n+k-1)$-th or lower word of the product. In this way, the $(n+k)$-th or higher word of temporary product remains zero. As to row k, the second-last instruction may produce carry to the $(n+k)$-th. According to equation (1), after executing its last instruction, the $(n+k)$-th word of temporary product is less than or equal to $(2^{32}-2)+1 = 2^{32}-1$. It implies that it will not produce carry, either.

In this way, we prove that when every last instruction of the row is the upper-half multiply-add instruction, it will not produce carry. As shown in Fig. 2, every last instruction of the row meets this requirement. Thus, we do not need to handle the CF bit that the last instruction produces.

Secondly, we need to double the temporary product. Since temporary product is $(2n-2)$ words (from 1th to $(2n-2)$-th word), we need $(2n-1)$ add instructions to double it (extra one add instruction is used for storing CF the last add instruction produces).

Thirdly, we add the gray piece, which is corresponding to $\sum_{0\le i\le n-1}^{n}[2^{2ri}(a_i^2)_{lo}+2^{r(2i+1)}(a_i^2)_{hi}]$. Like what we did before, gray piece can also be flattened into one row similarly. Using the instruction `madc.cc`, no add instruction is needed.

Therefore, in total, (n^2+n) multiply and $(2n-1)$ add instructions are needed. There is a slight difference between situations when n is odd or even. Detail is shown in Algorithm 2.

4 Implementation and Performance Evaluation

4.1 Comparison between Traditional and Proposed Algorithm

In theory, our proposed algorithm needs only $O(n)$ add instructions, while the traditional one needs $O(n^2)$. However, in practice, we should take other factors into consideration.

In CUDA GPU, integer add and multiplication instruction use different computing units, which allows the 2 kinds of instructions to execute parallelly. Due to multi-thread feature of GPU, when some threads occupy the multiplier, others can use the adder. Thus, in some situations, the decrease of add instruction may not lead to significant performance promotion.

To measure how the number of add instructions affects the overall performance, we conduct an experiment. A function which contains 128 multiply-add instructions (that 256-bit multiplication needs) with several add instructions appending is constructed for evaluation. To ignore other overheads, we run 10,000 loops of the test function, in which uses result in the loop as a new operand for the next loop. In this experiment, 14×1024 threads are used to fully occupy the computing resource. Varying the number of add instructions, we record its latency. The experimental data shows in Fig. 3.

We can see from Fig. 3, when the number of add instructions is less than 128, the performance changes slightly. After reaching the threshold 128, the latency starts to correlate positively with the number of add instructions. From this

Algorithm 2. Proposed Square Algorithm(r bits per word)

Input:
n-word-length Multiplicand, $A[0:n-1] = \sum_{i=0}^{n-1} a_i 2^{ri}$;
Output:
$2n$-word-length $C[0:2n-1] = A[0:n-1]^2 = \sum_{i=0}^{2n-1} c_i 2^{ri}$;
1: $C[0:2n-1] = 0$,set CF=0
2: **for** $i = 0$ *to* $\lceil \frac{n}{2} - 2 \rceil$ **do**
3: **for** $j = i+1$ *to* $n-i-2$ **do**
4: $c_{i+j} = \texttt{madc.cc.lo}(a_j, a_i, c_{i+j})$
5: **end for**
6: **for** $k = 0$ *to* i **do**
7: $c_{n+2k-1} = \texttt{madc.cc.lo}(a_{n-1-i+k}, a_{i+k}, c_{n+2k-1})$
$c_{n+2k} = \texttt{madc.cc.hi}(a_{n-1-i+k}, a_{i+k}, c_{n+2k})$
8: **end for**
9: set CF=0
10: **for** $j = i+1$ *to* $n-i-2$ **do**
11: $c_{i+j+1} = \texttt{madc.cc.hi}(a_j, a_i, c_{i+j+1})$
12: **end for**
13: **for** $k = 0$ *to* i **do**
14: $c_{n+2k} = \texttt{madc.cc.lo}(a_{n-1-i+k}, a_{i+k+1}, c_{n+2k})$
$c_{n+2k+1} = \texttt{madc.cc.hi}(a_{n-1-i+k}, a_{i+k+1}, c_{n+2k+1})$
15: **end for**
16: **end for**
17: **if** n is even **then**
18: set CF = 0
19: **for** $k = 0$ *to* $\frac{n}{2} - 1$ **do**
20: $c_{n+2k-1} = \texttt{madc.cc.lo}(a_{n/2+k}, a_{n/2+k-1}, c_{n+2k-1})$
$c_{n+2k} = \texttt{madc.cc.hi}(a_{n/2+k}, a_{n/2+k-1,n+2k})$
21: **end for**
22: **end if**
23: set CF=0
24: **for** $i = 1$ *to* $2n-1$ **do**
25: $c_i = \texttt{addc}(c_i, c_i)$
26: **end for**
27: set CF=0
28: **for** $i = 0$ *to* $n-1$ **do**
29: $c_{2i} = \texttt{madc.cc.lo}(a_i, a_i, c_{2i})$
$c_{2i+1} = \texttt{madc.cc.hi}(a_i, a_i, c_{2i+1})$
30: **end for**
31: **return** $C[0:2n-1]$;

phenomenon, we can speculate that the traditional and proposed multiplication algorithms will perform the same. Because the number of add instructions does not reach the threshold in both algorithms.

However, our optimization is not useless. We choose $P = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$ as modulus, which is used in *Chinese Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves* [7]. As a Mersenne prime, fast reduction

method can be applied to conduct modular reduction based on the work by [21]. Through optimizing, a 256-bit modular reduction over P consumes about 110 add instructions in total. Due to the great decrease of add instructions, when cooperating with modular reduction, our proposed modular multiplication algorithm is still on the "flat" line, while, traditional modular multiplication algorithm has surpassed the threshold, whose latency increases by nearly 20 %.

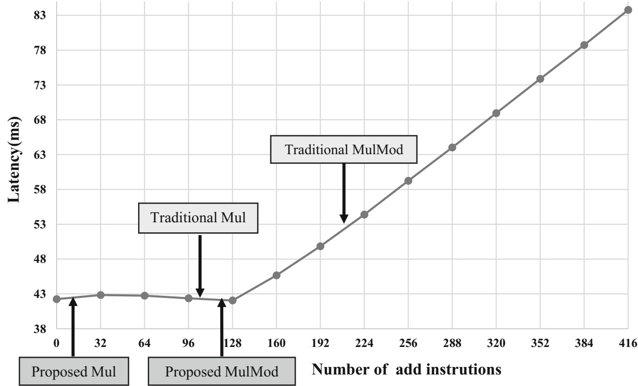

Fig. 3. Latency of 10,000 loops of test function (128 multiplication instructions with varying number of add instructions appending)

Our performance evaluation comparison proves the speculation well, which is shown in Table 1.

Table 1. Comparison between the traditional algorithm and the proposed algorithm (GeForce GTX Titan, 14 × 1024 threads)

		Traditional	Proposed
256-bit multiplication	Throughput (10^9/s)	3.3939	3.4121
	Latency (μs)	4.2241	4.2015
256-bit modular multiplication	Throughput (10^9/s)	2.8853	3.3837
	Latency (μs)	4.9687	4.2368
256-bit square	Throughput (10^9/s)	5.6436	6.0011
	Latency (μs)	2.5402	2.3889
256-bit modular square	Throughput (10^9/s)	5.1661	5.9928
	Latency (μs)	2.7750	2.3922

4.2 Related Work Comparison

In this section, we compare our resulting implementation with previous work [2,4,6,9,19]. For fair comparison, we firstly evaluate the CUDA platform of each work.

It is difficult to evaluate performance of each CUDA platform since they are constructed in different architectures. Note that [2,4,6,9,19] and ours are all based on integer arithmetic. Therefore, we can measure their performance depending mainly on integer processing capability as shown in the former part of Table 2. The parameters in Table 2 origin from [23], but the integer processing power is not given directly, we calculate them by SM Number, processing power of each SM and Shader Clock. Note that 9800GX2, GTX 285 and GTX 295 support only 24-bit multiply instruction, while, the other platforms support 32-bit multiply instruction. Hence, we adjust their integer multiply processing capability by a correction parameter $(\frac{24}{32})^2$. The overall integer processing capability *Int Capability* is graded by how many operations including a multiplication (or multiply-add) and an add instruction can be accomplished in a second.

Since the results of modular multiplication are not given in some papers, we also implement a simple elliptic curve point multiplication using our algorithm for evaluation. In overall structure, we use single thread to execute one elliptical curve point multiplication. Except for the modular multiplication, we do not take special optimization for it.

For better comparison, we scale the modular multiplication performance of each implementation using the Eq. 2:

$$MulMod(scaled) = MulMod \times \frac{370.7}{Int\ Capability} \times (\frac{Bits}{256})^2 \times (1 + isModulusGeneric) \quad (2)$$

The scaled point multiplication performance *Curve(scaled)* is defined similarly. The latter part of Table 2 shows the corresponding numerical results.

From Table 2, we can see that our algorithm makes great improvement compared with previous implementations. We gain over 3 times performance of the next fastest implementation [4]. Our point multiplication implementation also outperforms others by a considerable margin. We achieve 43.6 % performance promotion of the existing fastest implementation [6].

Additionally, we measure the hardware limiting performance for 256-bit modular multiplication. In modular multiplication, the number of multiplication instructions is the bottleneck of the performance, which cannot be reduced. From Table 2, we can see that the throughput of the 32-bit multiplication instruction reaches 448.6×10^9/s in GTX Titan. 256-bit modular multiplication costs $(\frac{256}{32})^2 \times 2 = 128$ instructions, thus the upper bound of modular multiplication performance is $\frac{448.6}{128} = 3.5047 \times 10^9$/s. Similarly evaluated, the upper bound of the modular square performance is $\frac{448.6}{72} = 6.2306 \times 10^9$/s. Respectively, our modular multiplication and square algorithm reach $\frac{3.3837}{3.5047} = 96.5\,\%$ and $\frac{5.9928}{6.2306} = 96.2\,\%$ of the GPU hardware limitation.

Table 2. Throughput and latency of operations per second

	Giorgi et al.[9]	Antão et al.[2]	Bernstein et al.[4]	Bos et al.[6]	Pu et al.[19]	Ours
Bits	256-bit	224-bit	210-bit	224-bit	224-bit	256-bit
isModulusGeneric	Yes	Yes	Yes	No	Yes	No
CUDA platform	9800GX2	GTX 285	GTX 295	GTX 580	GTX 680	GTX Titan
SM Number	32	30	60	16	8	14
Shader Clock(GHz)	1.500	1.476	1.242	1.544	1.006	0.993
Int Mul/SM(/Clock)	8	8	8	16	32	32
Int Add/SM(/Clock)	10	10	10	32	160	160
Int Mul(G/s)	216	199	335	395	257	448.6
Int Add(G/s)	480	443	791	745	1288	2224
Int Capability(G/s)	149.0	137.3	235.3	258.1	226.4	370.7
MulMod (10^6/s)	12.34	-	481	-	219	3383.7
MulMod(scaled) (10^6/s)	61.40	-	1019	-	549	3383.7
SqrMod (10^6/s)	-	-	-	-	-	5992.8
Curve(/s)	1,620	9,827	-	290,535	47,000	391,595
Curve(scaled)(/s)	8,061	40,627	-	272,681	117,839	391,595

5 Summary

In this paper, we propose and implement a new modular multiplication algorithm for ECC implementation in CUDA, aiming to minimize the number of add instructions and develop the full potential of GPU. In our proposed multiplication algorithm, the number of add instructions needed decreases from $O(n^2)$ in prevailing schoolbook algorithm to $O(n)$. Our resulting modular multiplication and modular square respectively reach 3.3837 billion and 5.9928 billion operations per second, reaching over 96 % of GPU hardware limitation. And a simple elliptic curve point multiplication using our algorithm is implemented at speed of 391,595 per second, which achieves 43.6 % speedup compared to the existing fastest work.

References

1. Antão, S., Bajard, J.C., Sousa, L.: Elliptic curve point multiplication on GPUs. In: IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP), pp. 192–199 (2010)
2. Antão, S., Bajard, J.C., Sousa, L.: RNS-Based elliptic curve point multiplication for massive parallel architectures. Comput. J. **55**(5), 629–647 (2012)
3. Bajard, J.C., Didier, L.S., Kornerup, P.: Modular multiplication and base extensions in residue number systems. In: Proceedings of the 15th IEEE Symposium on Computer Arithmetic, pp. 59–65 (2001)

4. Bernstein, D.J., Chen, H.C., Chen, M.S., Cheng, C.M., Hsiao, C.H., Lange, T., Lin, Z.C., Yang, B.Y.: The billion-mulmod-per-second PC. In: Workshop Record of SHARCS, vol. 9, pp. 131–144 (2009)
5. Bernstein, D.J., Chen, T.-R., Cheng, C.-M., Lange, T., Yang, B.-Y.: ECM on graphics cards. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 483–501. Springer, Heidelberg (2009)
6. Bos, J.W.: Low-latency elliptic curve scalar multiplication. Int. J Parallel Prog. **40**(5), 532–550 (2012)
7. Chinese Commercial Cryptography Administration Office: public key cryptographic algorithm SM2 based on elliptic curves (in Chinese) (2013). http://www.oscca.gov.cn/UpFile/2010122214822692.pdf
8. Cohen, A.E., Parhi, K.K.: GPU accelerated elliptic curve cryptography in GF (2^m). In: IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 57–60 (2010)
9. Giorgi, P., Izard, T., Tisserand, A., et al.: Comparison of modular arithmetic algorithms on GPUs. In: ParCo'09: International Conference on Parallel Computing (2009)
10. Hankerson, D., Vanstone, S., Menezes, A.J.: Guide to Elliptic Curve Cryptography. Springer, New York (2004)
11. Harrison, O., Waldron, J.: Efficient acceleration of asymmetric cryptography on graphics hardware. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 350–367. Springer, Heidelberg (2009)
12. Henry, R., Goldberg, I.: Solving discrete logarithms in smooth-order groups with CUDA. In: Workshop Record of SHARCS, pp. 101–118. Citeseer (2012)
13. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**(177), 203–209 (1987)
14. Koç, Ç.K., Acar, T., Kaliski Jr, B.S.: Analyzing and comparing montgomery multiplication algorithms. Micro IEEE **16**(3), 26–33 (1996)
15. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
16. Moss, A., Page, D., Smart, N.P.: Toward acceleration of RSA using 3D graphics hardware. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 364–383. Springer, Heidelberg (2007)
17. Neves, S., Araujo, F.: On the performance of GPU public-key cryptography. In: IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 133–140 (2011)
18. NVIDIA: CUDA Toolkit Documentation v6.0 (2014). http://docs.nvidia.com/cuda/index.html#axzz39iNG9lqx
19. Pu, S., Liu, J.-C.: EAGL: an elliptic curve arithmetic GPU-based library for bilinear pairing. In: Cao, Z., Zhang, F. (eds.) Pairing 2013. LNCS, vol. 8365, pp. 1–19. Springer, Heidelberg (2014)
20. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
21. Solinas, J.A.: Generalized mersenne numbers. Citeseer, Bielefeld (1999)
22. Szerwinski, R., Güneysu, T.: Exploiting the power of GPUs for asymmetric cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)
23. Wikipedia: Wikipedia: list of NVIDIA graphics processing units (2014). http://en.wikipedia.org/wiki/Comparison_of_NVIDIA_Graphics_Processing_Units