ECDLP on GPU

Lei Xu
State Key Laboratory of Information Security
Institute of Software,Chinese Academy of Sciences
Beijing, China
Email: xuleimath@gmail.com

Dongdai Lin
State Key Laboratory of Information Security
Institute of Software,Chinese Academy of Sciences
Beijing, China
Email: ddlin@is.iscas.ac.cn

Jin Zou
State Key Laboratory of Information Security
Institute of Software,Chinese Academy of Sciences
Beijing, China
Email: zoujing@is.iscas.ac.cn

***Abstract*—Elliptic curve discrete logarithm problem (ECDLP) is one of the most important hard problems that modern cryptography, especially public key cryptography, relies on. And many efforts are dedicate to solve this problem. In recent days, GPU technology develops very fast and GPU has become a powerful tool for massive computation. In this paper, we give an implementation of parallel Pollard ρ method, for ECDLP on GPU, and eliminate nearly all the conditional branches in procedures for big integer, elliptic curve and iteration function. The experimental result shows that with the help of GPU, we can gain a speedup of more than one hundred times. The branchless procedures are also useful for preventing side channel attacks.**

I. Introduction

Elliptic curve discrete logarithm problem (ECDLP) is one of the most important hard problems that modern cryptography, especially public key cryptography relies on. Informally, let P, Q be two points on an elliptic curve E and $Q = kP$, where k is some integer. To solve ECDLP means to find out the integer k. Many efforts are dedicated to the solve of ECDLP. For some special parameters, we have very efficient algorithm(such as for anomalous curve), and in some cases, we can deduce ECDLP to discrete logarithm problem on finite field ([1]), which is easier than ECDLP. But for general ECDLP, there is no sub-exponent algorithm and parallel Pollard ρ method([2]) is believed to be the most effective.

Recently, graphic processing units (GPU) are rising as an exciting new trend in high-performance computing. While multi-core CPUs generally exploit the task level parallelism, the GPU computing is based upon a data parallel programming model. When receiving a workload, a GPU would launch tens of thousands of fine-grained threads concurrently, with each thread executing the same program but on different chunk of data. NVIDIA's latest GPU GTX480, could deliver a peak double-precision arithmetic rate of 168 Gflops. Meanwhile, GPU programming has been made accessible to non-graphic programmers with NVIDIA's Compute Unified Device Architecture (CUDA) technology([3], [4]). [5] gave some description of solving `ECC2K-131` using GPU, but without detailed information.

In this paper, we give a method to implement the parallel Pollard ρ method on GPU. Detailed information on related algorithms such as integer and finite field is also provided. The preliminary experimental result is encouragement: the GPU is about 100 times faster than traditional CPU for this application.

The rest of this paper is organized as follows. Section II gives a brief introduction to elliptic curve over $GF(2^n)$ and parallel ρ method. Section III sketches NVIDIA's GPU technology and programming environment CUDA. Section IV gives detailed information about our implementation of parallel Pollard ρ method for `ECC2K-163` on GPU. Finally we give the experimental result and draw a conclusion in Section V.

II. Some Background

A. Elliptic Curve over $GF(2^n)$

For elliptic curve $E : y^2 + xy = x^3 + ax^2 + b$ defined on $GF(2^n)$, the addition laws for points are as follows:

1) Identity. For every $P \in E(GF(2^n))$, we have $P + \mathcal{O} = \mathcal{O} + P = P$;
2) Negative. If $P = (x, y) \in E(GF(2^n))$, then $(x, y) + (x, x + y) = \mathcal{O}$, and denote $(x, x + y)$ as $-P$, called the negative of P. Specifically, $\mathcal{O} = -\mathcal{O}$;
3) Addition. Let $P = (x_1, y_1), Q = (x_2, y_2) \in E(GF(2^n))$ and $P \neq \pm Q$, then $P + Q = (x_3, y_3)$, where
$$\begin{aligned} \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 + x_2) + x_3 + y_1; \end{aligned}$$
4) Doubling. Let $P = (x_1, y_1) \in E(GF(2^n))$ and $P \neq -P$, then $2P = (x_3, y_3)$, where
$$\begin{aligned} \lambda &= x_1 + y_1/x_1 \\ x_3 &= \lambda^2 + \lambda + a \\ y_3 &= x_1^2 + \lambda x_3 + x_3. \end{aligned}$$

B. Parallel ρ Method

ρ method is a general method to calculate discrete logarithms over Abelian groups. It was first proposed by Pollard in [6] and its time complexity is about $O(\sqrt{n})$, where n is the size of the group.

Let G be cyclic Abelian group, $g, h \in G$, g is a generator of G and $h = g^x$. The basic principle of ρ method is to find integers a_1, b_1, a_2, b_2 s.t.

$$h^{a_1} g^{b_1} = h^{a_2} g^{b_2}$$

If $a_1 - a_2$ is relative prime to $|G|$, then we have

$$x \equiv (b_2 - b_1)(a_1 - a_2)^{-1} \ mod \ |G| \tag{1}$$

ρ method relies on a random function f. First initialize a beginning element $X_0 = h^{x_0} g^{y_0}$. Then we calculate $X_1 = f(X_0), X_2 = f(X_1)$ and so on. Until we get a collision s.t. $X_i = X_j$.

Paul C. van Oorschot and Michael J. Wiener proposed a parallel ρ method with linear speedup([2]). First we define some "distinguished points", which satisfy some criteria. Then we run an iteration function for each processor with different random initial point. If during an iteration, we find a distinguished point, the point is send to a central processor. The central processor is responsible for distinguished points storage and collision detection.

III. GPU AND CUDA

In this work, we use NVIDIA's CUDA platform. Here we briefly review the characters of CUDA hardware and software. See [3] and [4] for more detailed information.

The GPU we used in this work is NVIDIA Tesla C1060. It is not the latest flagship GPU chip of NVIDIA but has already offer great computation ability. The main computing resource is organized as an array of 30 streaming multiprocessors (SMs), and each SM has 8 streaming processors (SPs). NVIDIA's GPU offers various memories: global memory, texture memory, constant memory, local memory and register. The capacity and speed of these memory difference largely. Generally speaking, the larger the storage space, the slower the access speed.

A CUDA program is composed of codes running on both CPU and GPU. And the GPU code would be concurrently executed by GPU as coordinated by CPU. The function called by CPU but executed on GPU is named as a *kernel*. One CUDA program could have multiple kernels executed sequentially. According to the CUDA model, a GPU application could launch tens of thousands of threads, with each running the same program on different data sets. *Thread* is the minimum unit of parallel execution, and the code inside a thread runs sequentially. A number of threads are organized into a *block*, and a number of blocks are organized into a *grid*. Threads in the same block are reside on the same SM and these threads can exchange data through shared memory. CUDA also offers some methods for synchronization between GPU threads and between kernel and CPU program.

Performance optimization on GPU revolves around three basic strategies:

- Maximize parallel execution to achieve maximum utilization;
- Optimize memory usage to achieve maximum memory throughput;
- Optimize instruction usage to achieve maximum instruction throughput.

The essence of parallel Pollard ρ method is suitable for GPU, so the main concerns are to optimize memory and instruction usage, especially the control flow instructions. Any flow control instruction (`if`, `switch`, `do`, `for`, `while`) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge (i.e. to follow different execution paths). If this happens, the different executions paths have to be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path.

IV. OUR DESIGN AND IMPLEMENTATION

In this section, we give detailed information about our design and implementation of `ECC2K-163` on GPU with CUDA. However, our design is not specific to the parameters of `ECC2K-163`, it can be easily extended to other curves over $GF(2^n)$. Also note that in this work we only consider polynomial basis representation for elements of $GF(2^n)$.

In the following descriptions of algorithms, "&" stands for `AND` operation and "$\oplus$" stands for `XOR` operation.

A. Parameters of *ECC2K-163*

The parameters of `ECC2K-163` are as follows:

- The elliptic curve:

$$E : y^2 + xy = x^3 + x^2 + 1$$

- The field polynomial:

$$p(x) = x^{163} + x^8 + x^2 + 1$$

- The base point order:

$$n = \texttt{00000004 00000000 00000000 00020108 A2E0CC0D 99F8A5EF}$$

- The base point $P = (P_x, P_y)$

$$\begin{aligned} P_x &= \texttt{00000002 091945E4 2080CD9C BCF14A71 07D8BC55 CDD65EA9} \\ P_y &= \texttt{00000006 33156938 33774294 A39CF6F8 C175D02B 8E6A5587} \end{aligned} \tag{2}$$

- The public key point $Q = (Q_x, Q_y)$

$$\begin{aligned} Q_x &= \texttt{00000000 7530EE86 4EDCF4A3 1C85AA17 C197FFF5 CAFECAE1} \\ Q_y &= \texttt{00000007 5DB1E80D 7C4A92C7} \\ &= \texttt{BBB79EAE 3EC545F8 A31CFA6B} \end{aligned}$$

B. Overall Roadmap

The overall roadmap is as follow:

1) GPU threads start from different random points on the elliptic curve and iterate independently;
2) A cache is set in global memory to save distinguished points and an variable `counter` is maintained for the next position to save distinguished point;
3) If a thread find a distinguished point, use the CUDA atomic instruction atomicAdd() to increase `counter`, and the point is saved in the cache at position stored in previous `counter`;
4) When the cache is full, all the data is copied to host memory or disk.

C. Judgement of Distinguished Points

The criteria for distinguished point is in fact a trade-off between time and space: If the criteria is easy to achieve, then we will gain more distinguished points and get the desired discrete logarithm more quickly, but at the same time the space consumption will increase, and vise versa.

Here we use the hamming weight of the x-coordinate of a point as the criteria. CUDA offers a function to get the hamming weight of a 32 bits integer, so the hamming weight of the x-coordinate of a point can easily been obtained.

D. Iteration Function Implementation

We just select the simple iteration function firstly given by Pollard in [6]. The functions used in the iteration are given in the following sections.

Let $T_i = a_iP + b_iQ$, the iteration function is $T_{i+1} = f(T_i) and$:

$$f(T_i) = \begin{cases} T_i + P, & a_i \leftarrow a_i + 1, hw(T_i.x) \ mod \ 3 = 0 \\ T_i + T_i, & a_i \leftarrow a_i + a_i, hw(T_i.x) \ mod \ 3 = 1 \\ & b_i \leftarrow b_i + b_i \\ T_i + Q, & b_i \leftarrow b_i + 1, hw(T_i.x) \ mod \ 3 = 2 \end{cases} \tag{3}$$

In order to keep the iteration function branch free, let

$$\begin{aligned} T_i &= T_i + k_1P + k_2Q + k_3T_i, \\ a_i &= a_i + l_1a_i + l_2, \\ b_i &= b_i + m_1b_i + m_2, \end{aligned}$$

where $k_1, k_2, k_3, l_1, l_2, m_1, m_2 \in \{0, 1\}$.

According to the iteration function we choose, these values can be set as in Table I.

E. Integer Representation and Related Algorithms

In this work we concern the `Certicom ECC2K-163` challenge, so only integers less than n are considered.

TABLE I
THE PARAMETERS SETTING FOR ITERATION FUNCTION

	$HW(T_i) \ mod \ 3$		
	0	1	2
k	$k_1 = 1$	$k_1 = 0$	$k_1 = 0$
	$k_2 = 0$	$k_2 = 0$	$k_2 = 1$
	$k_3 = 0$	$k_3 = 1$	$k_3 = 0$
l	$l_1 = 0$	$l_1 = 1$	$l_1 = 0$
	$l_2 = 1$	$l_2 = 0$	$l_2 = 0$
m	$m_1 = 0$	$m_1 = 1$	$m_1 = 0$
	$m_2 = 0$	$m_2 = 0$	$m_2 = 1$

1) Data Structure for Big Integer: Suppose the word size is 32 bits, we use 6 words to represent an integer. In order to keep the implementation branch free, only the low 28 bits of each word are used except the highest word. So there are totally 172 bits for an integer.

Specifically, an integer a is represented by $A[5]A[4]\dots A[0]$, s.t.

$$a = A[0] + A[1]2^{28} + A[2]2^{56} + \cdots + A[5]2^{140},$$

the low 28 bits of $A[0], A[1], A[2], A[3], A[4]$ are used and the whole word $A[5]$ can be used for the representation.

2) Addition of Big Integers: The addition of two integers is carried out in two stage:

1) Add the corresponding words of the two integers;
2) Process the carries.

Note that this two-stage strategy is applicable because we do not use the whole word for big integers(except for the highest one), so addition of words will not cause overflow.

Algorithm 1 describes the addition procedure, the first loop (line 1 to 3 of Algorithm 1) is for word additions, and the second loop (line 4 to 7 of Algorithm 1) is for carries treatment.

Algorithm 1: Addition of Big Integers

Input: Big integer a, b
Output: $c = a + b$

```
1 for i from 0 to 5 do
2 |  C[i] ← A[i] + B[i]
3 end
4 for i from 0 to 4 do
5 |  C[i+1] ← C[i+1] + (C[i] >> 28)
6 |  C[i] ← C[i] & 0x0FFFFFFF
7 end
8 return c
```

3) Modulus of Big Integers n: The same representation method is used for the modular number n(the order of point P), that is to say

$$n = N[0] + N[1]2^{28} + N[2]2^{56} + \cdots + N[5]2^{140},$$

where $N[5] = \texttt{0x0400000}, N[4] = \texttt{0x0000000}, N[3] = \texttt{0x0000000}, N[2] = \texttt{0x020108A2}, N[1] = \texttt{0x0E0CC0D9}, N[0] = \texttt{0x09F8A5EF}$.

We do not do modular operation as soon as an addition is completed, only in the case that the integer a is much bigger than the moduli n (that is, the highest 4 bits of $A[5]$ are used), a modular operation is taken place.

The modular operation also works in two stages:

1) Borrows in advance and subtract n. Because the highest 4 bits of $A[5]$ are used, $A[5]$ is strictly larger than $N[5]$, so after subtract $N[5]$ from $A[5]$, $A[4]$ can borrow a digit from $A[5]$. This ensures that $A[4] > N[4]$. Repeat the same procedure and at last we subtract $N[0]$ from $A[0]$.
2) Process possible carries. In the first stage, unnecessary borrows may be done, so we treat these problems in this stage.

Algorithm 2 describes the procedure of modulus.

Algorithm 2: Modulus of Big Integers n

Input: Big integer a and moduli n
Output: $c = a \ mod \ n$

$T \leftarrow ((A[5] \ \& \ \texttt{0xF0000000}) == 0) \cdot \texttt{0xFFFFFFFF}$
$C[5] \leftarrow C[5] - (T \ \& \ N[5])$, $C[5] \leftarrow C[5] - 1$
$C[4] \leftarrow C[4] \mid \texttt{0x10000000}$, $C[4] \leftarrow C[4] - 1$
$C[3] \leftarrow C[3] \mid \texttt{0x10000000}$, $C[3] \leftarrow C[3] - 1$
$C[2] \leftarrow C[2] \mid \texttt{0x10000000}$,
  $C[2] \leftarrow C[2] - (T \ \& \ N[2])$, $C[2] \leftarrow C[2] - 1$
$C[1] \leftarrow C[1] \mid \texttt{0x10000000}$,
  $C[1] \leftarrow C[1] - (T \ \& \ N[1])$, $C[1] \leftarrow C[1] - 1$
$C[0] \leftarrow C[0] \mid \texttt{0x10000000}$,
  $C[0] \leftarrow C[0] - (T \ \& \ N[0])$
for i *from* 0 *to* 4 **do**
  $C[i+1] \leftarrow C[i+1] + (C[i] >> 28)$
  $C[i] \leftarrow C[i] \ \& \ \texttt{0x0FFFFFFF}$
end
return c

F. Finite Field Element Representation and Related Algorithms

1) Data Structure for Finite Field Element: Using polynomial basis, element of $GF(2^n)$ is represented by an array naturally. Specifically, for $e \in GF(2^n)$,

$$e = E[0]E[1] \dots E[5],$$

where the lowest bit of $E[0]$ is the constant coefficient of the corresponding polynomial.

2) Addition of Finite Field Elements: The addition of two elements is simple. We can just `XOR` the corresponding words of two finite field elements and get the result.

3) Multiplication of Finite Field Elements: Multiplication of finite field elements is done by multiplication of two polynomials and then reduce the result with $p(x)$.

We adopt the multiplication algorithm of NTL([7]), which extend the original Karatsuba-Ofman algorithm([8], [9]).

1) For multiplication of 1 word polynomials, use school book shift-add algorithm, and denote this operation by `MUL_1`;
2) For multiplication of 3 words polynomials, use 3-way Karatsuba-Ofman multiplication, denoted by `MUL_3` (Algorithm 3);
3) For multiplication of 6 words polynomials, use 2-way Karatsuba-Ofman multiplication, denoted by `MUL_6` (Algorithm 4).

Note that for the 1 word polynomials multiplication, NTL ([7]) use some pre-computation, but we just use the naive shift-add algorithm, because the space cost of pre-computation.

Algorithm 3: Multiplication of 3 Words Polynomials(`MUL_3`)

Input: Polynomial a and b, where a, b can be represent by 3 words
Output: $c = a \cdot b$

Allocate temporary storage
  $d0[2], d1[2], d2[2], d01[2], d02[2], d12[2]$
`MUL_1`$(d0, A[0], B[0])$
`MUL_1`$(d1, A[1], B[1])$
`MUL_1`$(d2, A[2], B[2])$
`MUL_1`$(d01, A[0] \oplus A[1], B[0] \oplus B[1])$
`MUL_1`$(d02, A[0] \oplus A[2], B[0] \oplus B[2])$
`MUL_1`$(d12, A[1] \oplus A[2], B[1] \oplus B[2])$
$C[0] \leftarrow d0[0]$
$C[1] \leftarrow d0[1] \oplus d01[0] \oplus d1[0] \oplus d0[0]$
$C[2] \leftarrow d01[1] \oplus d1[1] \oplus d0[1] \oplus d02[0] \oplus d2[0] \oplus d0[0] \oplus d1[0]$
$C[3] \leftarrow d02[1] \oplus d2[1] \oplus d0[1] \oplus d1[1] \oplus d12[0] \oplus d1[0] \oplus d2[0]$
$C[4] \leftarrow d12[1] \oplus d1[1] \oplus d2[1] \oplus d2[0]$
$C[5] \leftarrow d2[1]$
return c

It can be easily seen that the multiplication algorithm is branch free.

Reduction with $p(x) = x^{163} + x^8 + x^2 + 1$ is similar to the reduction algorithms for the NIST recommended irreducible polynomials, see [10] for referee.

We describe the reduction algorithm in Algorithm 5.

4) Inversion of Finite Field Element: We use Fermat's Little Theorem for inversion computation. This algorithm has two advantages:

- This algorithm is simple and branch free;
- This algorithm can applied to ZERO element and the inversion result is also ZERO, which is an useful property in point addition.

5) Multiplication of Finite Field Element with a Bit: In the points addition procedure, we have to do some multiplications of finite field element and a bit. This operation can be done using ordinary multiplication but the cost is expensive. Here we give a much simpler algorithm for this operation (Algorithm 6).

G. Representation of Points on Elliptic Curve and Related Algorithms

For detailed information about elliptic curve and related theory, we refer the readers to [10] and [11]. Here we focus

TABLE II
PARAMETERS CONFIGURATION FOR ELLIPTIC CURVE POINTS ADDITION

	P_1 or $P_2 = \mathcal{O}$	$P_1 = -P_2$	$P_1 \neq P_2 \neq \mathcal{O}$	$P_1 = P_2 \neq \mathcal{O}$
λ	0	0	N/A	N/A
ℓ	0	0	1	1
m	irrelevant with m_1	irrelevant with m_1	$m_1 = 1$	$m_1 = 0$
	$m_2 = m_5 = 0$	$m_2 = m_5 = 0$	$m_2 = 0, m_5 = 1$	$m_2 = m_5 = 1$
	$m_3 = m_4 = 1$	$m_3 = m_4 = 0$	$m_3 = 1, m_4 = 0$	$m_3 = m_4 = 0$
k	irrelevant with k_1, k_2	irrelevant with k_1, k_2	$k_1 = 1, k_2 = 0$	$k_1 = 0, k_2 = 1$
	$k_3 = 1$	$k_3 = 1$	$k_3 = 1$	$k_3 = 0$
t	0	irrelevant with t	1	1

Algorithm 4: Multiplication of 6 Words Polynomials

Input: Polynomial a and b, where a, b can be represent by 6 words
Output: $c = a \cdot b$

Allocate temporary storage $hs0[3], hs1[3], hl2[6]$
$hs0[0] \leftarrow A[0] \oplus A[3]$
$hs0[1] \leftarrow A[1] \oplus A[4]$
$hs0[2] \leftarrow A[2] \oplus A[5]$
$hs1[0] \leftarrow B[0] \oplus B[3]$
$hs1[1] \leftarrow B[1] \oplus B[4]$
$hs1[2] \leftarrow B[2] \oplus B[5]$
$\texttt{MUL\_3}(c, a, b)$
$\texttt{MUL\_3}(c+6, a+3, b+3)$
$\texttt{MUL\_3}(hl2, hs0, hs1)$
$hl2[0] \leftarrow hl2[0] \oplus C[0] \oplus C[6]$
$hl2[1] \leftarrow hl2[1] \oplus C[1] \oplus C[7]$
$hl2[2] \leftarrow hl2[2] \oplus C[2] \oplus C[8]$
$hl2[3] \leftarrow hl2[3] \oplus C[3] \oplus C[9]$
$hl2[4] \leftarrow hl2[4] \oplus C[4] \oplus C[10]$
$hl2[5] \leftarrow hl2[5] \oplus C[5] \oplus C[11]$
$C[3] \leftarrow C[3] \oplus hl2[0]$
$C[4] \leftarrow C[4] \oplus hl2[1]$
$C[5] \leftarrow C[5] \oplus hl2[2]$
$C[6] \leftarrow C[6] \oplus hl2[3]$
$C[7] \leftarrow C[7] \oplus hl2[4]$
$C[8] \leftarrow C[8] \oplus hl2[5]$
return c

on the arithmetic implementation problems on GPU.

1) Data Structure for Point on Elliptic Curve: Although using projective coordinate can avoid the expensive inversion operation in point addition, it is inconvenient to judge whether a point is distinguished or not. So we choose affine coordinate for points and each point is represent by two finite field elements. The point at infinity is represented by two zero elements.

Algorithm 5: Reduction with $p(x) = x^{163}+x^8+x^2+x+1$

Input: Polynomial$c(x)$,$degree(c(x)) \leq 324$
Output: $d(x) = c(x)\ mod\ p(x)$

for i *from* 10 *to* 6 **do**
  $T \leftarrow C[i]$
  $C[i-6] \leftarrow C[i-6] \oplus (T << 29) \oplus (T << 30) \oplus (T << 31)$
  $C[i-5] \leftarrow C[i-5] \oplus (T >> 3) \oplus (T >> 2) \oplus (T >> 1) \oplus (T << 5)$
  $C[i-4] \leftarrow C[i-4] \oplus (T >> 27)$
end
$T \leftarrow C[5] >> 3$
$C[0] \leftarrow C[0] \oplus (T << 8) \oplus (T << 2) \oplus (T << 1) \oplus T$
$C[1] \leftarrow C[1] \oplus (T >> 24) \oplus (T >> 30) \oplus (T >> 31)$
$C[5] \leftarrow C[5]\ \wedge\ 0x07$
for i *from* 0 *to* 5 **do**
  $D[i] \leftarrow C[i]$
end
return d

Algorithm 6: Multiplication of Finite Field Element and Bit

Input: Finite field element a, a bit b saved in a word
Output: $c = a \cdot bit$

$T \leftarrow bit \cdot \texttt{0xFFFFFFFF}$
$C[0] \leftarrow A[0] \wedge T$
$C[1] \leftarrow A[1] \wedge T$
$C[2] \leftarrow A[2] \wedge T$
$C[3] \leftarrow A[3] \wedge T$
$C[4] \leftarrow A[4] \wedge T$
$C[5] \leftarrow A[5] \wedge T$
return c

2) Addition of Points on Elliptic Curve E: Let $P_3 = P_1 + P_2, P_i = (x_i, y_i)(i = 1, 2, 3)$, and

$$\lambda = \frac{y_1 + k_1 y_2 + k_2 x_1^2}{x_1 + k_3 x_2} \cdot t, \tag{4}$$

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + \ell, \tag{5}$$

$$y_3 = m_1 \lambda x_1 + m_2 x_1^2 + \lambda x_3 \tag{6}$$

$$+ m_3 y_1 + m_4 y_2 + m_5 x_3. \tag{7}$$

TABLE III
GPU AND CPU TIME CONSUMPTION COMPARISION FOR ITERATION

Hardware	Intel i7	NVIDIA Tesla C1060
Number of Iteration	$60 \times 96 \times 16$	$60 \times 96 \times 16$ (each thread does 16 iterations)
Time	447.007 sec	2.732 sec

where $k_1, k_2, k_3, \ell, m_1, m_2, m_3, m_4, m_5, t \in \{0, 1\}$.

These values are set to fit different point addition situations, as stated in Table II.

We will prove that formula 4,5, 6 are consistent with the original point addition formula if the parameters are set according to Table II.

1) If at least one of the points is infinity point, then according to Table II

$$\begin{aligned} x_3 &= x_1 + x_2 \\ y_3 &= y_1 + y_2 \end{aligned}$$

2) If one point is the negative of the other, then according to Table II

$$\begin{aligned} x_3 &= x_1 + x_2 = 0 \\ y_3 &= 0 \end{aligned}$$

3) If $P_1 \neq P_2 \neq \mathcal{O}$, then according to Table II

$$\begin{aligned} \lambda &= (y_1 + y_2)/(x_1 + x_2) \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + 1 \\ y_3 &= \lambda x_1 + \lambda x_3 + y_1 + x_3 \end{aligned}$$

4) If $P_1 = P_2$, then according to Table II

$$\begin{aligned} \lambda &= (y_1 + x_1^2)/x_1 = x_1 + y_1/x_1 \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + 1 = \lambda^2 + \lambda + 1 \\ y_3 &= x_1^2 + \lambda x_3 + x_3 \end{aligned}$$

In summary, in all the conditions, formula 4, 5, 6 are consistent with the point addition.

V. EXPERIMENTAL RESULTS AND ANALYSIS

The experimental environment is as follows: NVIDIA Tesla C1060 GPU, Intel i7 CPU with 8G memory. The operation system is Windows XP Professional 64, and the programming environment is Visual Studio 2008 SP1 + CUDA 3.1.

Because the efficiency of parallel Pollard ρ method is largely determined by the speed of iteration, we record the time of running the same number of iterations.

- for CPU, use one thread to do the iteration;
- for GPU, use 60×96 threads (60 blocks, and 96 threads per block).

The result is summarized in Table III.

Note that our branch free implementation of addition of elliptic curve points is also useful to prevent side channel attacks. Also note that there are many tricks to accelerate the parallel Pollard ρ method, such as considering the negative point. But we don't involve such tricks in our current implementation. So we believe there is still much room for further improvement.

REFERENCES

[1] A. J.Menezes, T. Okamoto, and S. A.Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field," *IEEE Transactions on Information Theory*, vol. 39, pp. 1639 – 1646, 1993.

[2] P. C. van Oorschot and M. J. Wiener, "Parallel collision search with cryptanalytic applications," *Journal of Cryptology*, vol. 12, pp. 1–28, 1999.

[3] NVIDIA. (2009) Nvidia cuda c programming guide. NVIDIA. [Online]. Available: www.nvidia.com/object/cuda_home.html

[4] ——. Nvidia cuda c programming best practices guide. NVIDIA. [Online]. Available: www.nvidia.com/object/cuda_home.html

[5] D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H.-C. Chen, C.-M. Cheng, G. van Damme, G. de Meulenaer, L. J. D. Perez, J. Fan, T. Gneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar, F. Regazzoni, P. Schwabe, L. Uhsadel, A. V. Herrewege, and B.-Y. Yang, "Breaking ecc2k-130," Cryptology ePrint Archive, Report 2009/541, 2009, http://eprint.iacr.org/.

[6] J. M. Pollard, "Monte carlo methods for index computation (mod p)," *Mathematics of Computation*, vol. 32, pp. 918–924, 1978.

[7] V. Shoup. Ntl: A library for doing number theory. [Online]. Available: http://www.shoup.net/ntl/

[8] A. A. Karatsuba and Y. P. Ofman, "Multiplication of multidigit numbers on automata." *Soviet Physics Doklady*, vol. 7, pp. 595 – 596, 1963.

[9] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1981, vol. 2.Seminumerical Algorithms.

[10] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[11] J. H.Silverman, *The Arithmetic of Elliptic Curves*, ser. Graduate Texts in Mathematics. Springer-Verlag, 1986.