Acceleration of Composite Order Bilinear Pairing on Graphics Hardware

Ye Zhang[1], Chun Jason Xue[2], Duncan S. Wong[2], Nikos Mamoulis[1] and S.M. Yiu[1]

[1] Department of Computer Science, The University of Hong Kong, Hong Kong
{yzhang4, nikos, smyiu}@cs.hku.hk
[2] Department of Computer Science, City University of Hong Kong, Hong Kong
{jasonxue, duncan}@cityu.edu.hk

Abstract. Recently, composite-order bilinear pairing has been shown to be useful in many cryptographic constructions. However, it is time-costly to evaluate. This is because the composite order should be at least 1024bit and, hence, the elliptic curve group order n and base field become too large, rendering the bilinear pairing algorithm itself too slow to be practical (e.g., the Miller loop is $\Omega(n)$). Thus, composite-order computation easily becomes the bottleneck of a cryptographic construction, especially, in the case where many pairings need to be evaluated at the same time. The existing solution to this problem that converts composite-order pairings to prime-order ones is only valid for certain constructions. In this paper, we leverage the huge number of threads available on Graphics Processing Units (GPUs) to speed up composite-order pairing computation. We investigate suitable SIMD algorithms for base field, extension field, elliptic curve and bilinear pairing computation as well as mapping these algorithms into GPUs with careful considerations. Experimental results show that our method achieves a record of 8.7ms per pairing on a 1024bit security level, which is a 20-fold speedup compared to state-of-the-art CPU implementation. This result also opens the road to adopting higher security levels and using rich-resource parallel platforms, which for example are available in cloud computing. In fact, we can achieve more than 24 times speedup on a 2048bit security level and a record of 7×10^{-6} USD per pairing on the Amazon cloud computing environment.

1   Introduction

A bilinear pairing $\hat{e} : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ is said to be over a composite-order group if the order $\mathbb{G}$ and $\mathbb{G}_T$ is composite. Pairings with this property are commonly used in recent cryptographic constructions, e.g., [5,6,12,14]. On the other hand, evaluating a pairing over a composite-order group is much more expensive compared to its prime-order counterpart. The composite order should be large enough (e.g., at least 1024bit) to be difficult to factorize, while a much smaller prime order (e.g., 160bit) is enough to achieve the same security level. As a result, the underlying finite field, elliptic curve operations and the pairing evaluating algorithm itself become much slower. An estimation [9] shows that the composite-order pairing would be 50x times slower than its prime-order counterpart. Thus, composite-order pairing computation easily becomes the bottleneck of a cryptographic construction, especially in cases where many such pairings need to be evaluated at the same time (e.g., a product of pairings in [12]).

There are some efforts to address this problem. Freeman [9] proposed a method that can convert a scheme constructed with a composite-order pairing to a prime-order pairing construction with the same functionality. However, Freeman's method is not black-box; it is only valid for certain cryptographic constructions. [15] also points out that some schemes inherently require composite-order groups and cannot be transformed mechanically from one setting to the other, by using the methodology of [9].

In this paper, we leverage the huge number of threads available on GPUs (Graphics Processing Units) to speed up the composite-order bilinear pairing computation. The proposed method considers parallelism both within and between pairings. To compute a pairing, we use a block of threads, while we concurrently run many blocks to compute many pairings in parallel. We first implemented 32bit modular addition, subtraction and multiplication on each thread. Addition, subtraction and multiplication operations on finite field $\mathbb{F}_q$ are conducted on a block of threads via Residue Number System (RNS) [13]. Multiplication and square operations on extension field $\mathbb{F}_{q^2}$ and addition and double operations on elliptic curve are implemented upon $\mathbb{F}_q$ operations, which in turn are based on a block of threads. Putting all together, the bilinear pairing algorithm [3] is implemented upon the $\mathbb{F}_q$ operations, $\mathbb{F}_{q^2}$ operations, and the elliptic curve operations. Compared to the existing work, our method is transparent and generic to cryptographic schemes. It can serve for all cryptographic schemes constructed in composite-order pairings.

To the best of our knowledge, this work is the first on evaluation of bilinear pairings (over composite-order group) on graphics card hardware. Porting the existing CPU-version code into the GPU is not trivial, due to the different levels of parallelism provided by CPUs and GPUs. As a result, we need to find and implement the best parallel (e.g., SIMD-fashion) algorithms for GPU that evaluate arithmetic operations on base field, extension field, elliptic curve, and the bilinear pairing algorithm itself. Different design decisions were made compared to the CPU code. For example, $\mathbb{F}_q$ operations in our implementation is done by a block of threads via RNS instead of the serialized method on CPU. Due to RNS, we had to seek the formulas that can minimize the number of modular reductions. Moreover, the multiplication inverse in the proposed implementation needs to be avoided which motivates us to choose a projective coordinate system to represent elliptic curve points and to postpone the final powering operation back to CPU. The experimental results show that the proposed method achieves a 20-fold speedup on a 1024bit security level, compared to state-of-the-art implementation [20] for CPUs. Specifically, it achieves a record of 8.7ms per pairing on average, which is comparable with prime-order group pairings.

The rest of this paper is organized as follows. Related work is discussed in Section 2. Sections 3 and 4 present background on the mathematics and GPU programming. In Sections 5 and 6, we present the arithmetic operations and the bilinear pairing algorithm in detail. Section 7 discusses the implementation considerations on mapping the algorithms discussed in Section 5 and 6 onto CUDA. The experimental results are shown in Section 8. We conclude this paper in Section 9.

2   Related Work

There have been many efforts to speed up cryptographic primitives over GPUs by exploring the large number of available cores. [17] and [8] first considered modular multiplication (i.e., the multiplication operation over a finite field $\mathbb{F}_q$ where q is a large prime number) over GPUs. At that time, GPUs were still designed for processing graphics only and therefore a large effort was required for researchers to map their programs to the GPU architecture. Later on, [21], [11], and [4] also provided a framework for implementing $\mathbb{F}_q$ operations, but this time on a CUDA-architecture GPU that provides researchers and programmers the ability to write general-purpose programs. More recently, Guillermin [10] investigated scalar multiplication of elliptic curve on an FPGA hardware. Still, none of the works above aims at speeding up a bilinear pairing (especially, over a composite-order group) which acts as one of the dominating tools in the recent cryptographic constructions design.

The first algorithm for computing a Tate pairing on CPUs was introduced by Miller [16] and an improved algorithm was proposed by Barreto et al. [3]. In this paper, we adopt the method proposed in [3]. The well-known implementation of bilinear pairings over CPUs is the Pairing-Based Cryptography (PBC) library [20], which we include for comparison in experiments.

3   Mathematics of Composite Order Bilinear Pairing

In this section, we introduce the basic concept on bilinear pairings and the group on which a bilinear pairing is defined. We describe the relationship between group size and security level and why the composite order in a bilinear pairing should be much larger than a prime order.

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two cyclic additive groups and $\mathbb{G}_T$ a cyclic multiplicative group. A bilinear map (of order $l \in \mathbb{N}$) is defined as

$$e_l : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T \tag{1}$$

with the following properties: (1) bilinear: for all $P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}$, $e_l(aP, bQ) = e_l(P,Q)^{ab}$; (2) non-degenerate: $e_l(P,Q) \neq 1$ for some $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$, where 1 is the identity element of $\mathbb{G}_T$; and (3) computable: there is an efficient algorithm to compute $e_l(P,Q)$ for any $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$. If there exists a distortion map $\phi : \mathbb{G}_1 \to \mathbb{G}_2$, we can define a *symmetric* bilinear pairing $\hat{e}_l : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ so that $\hat{e}_l(P_1, P_2) = e_l(P_1, \phi(P_2))$ for any $P_1, P_2 \in \mathbb{G}_1$.

Specifically, let E be an elliptic curve defined over a finite field $\mathbb{F}_q$ where $q = p^m$, $p, m \in \mathbb{N}$ and p is the *characteristic* of $\mathbb{F}_q$. Let $\mathcal{O}$ be the point at infinity for E. For a nonzero integer l, the set of points P in $E(\mathbb{F}_q)$ such that $lP = \mathcal{O}$ is denoted as $E(\mathbb{F}_q)[l]$. The group $E(\mathbb{F}_q)[l]$ is said to have *security multiplier* or *embedding degree* k for some $k > 0$ if $l \mid q^k - 1$ and $l \nmid q^s - 1$ for any $0 < s < k$. The Tate pairing of order l is a map

$$e_l : E(\mathbb{F}_q)[l] \times E(\mathbb{F}_{q^k})[l] \to \mathbb{F}_{q^k} \tag{2}$$

The pairing-friendly elliptic curve that we use for realizing a composite order bilinear pairing is a supersingular elliptic curve in the following form which is defined over a prime field.

$$E : y^2 = x^3 + (1 - b)x + b, \quad b \in \{0, 1\} \tag{3}$$

The group order l is composite and the embedding degree k is 2. There exists a distortion map $\phi : E(\mathbb{F}_q) \to E(\mathbb{F}_{q^k})$ which allows us to define a *symmetric* bilinear map as

$$\hat{e}_l : E(\mathbb{F}_q)[l] \times E(\mathbb{F}_q)[l] \to \mathbb{F}_{q^k} \tag{4}$$

so that $\hat{e}_l(P,Q) = e_l(P, \phi(Q))$ for any $P, Q \in E(\mathbb{F}_q)[l]$. In our implementation, we set $b = 0$, that is, $E : y^2 = x^3 + x$ over $\mathbb{F}_q$ and the prime $q \equiv 3 \pmod 4$. The order of $E(\mathbb{F}_q)$ is $\#E(\mathbb{F}_q) = q + 1$. This curve is referred to as A1 curve in the PBC software library [20].

3.1   Finite Field Size vs. Security Level

The security of pairing-based cryptosystems generally relies on two problems, elliptic curve discrete logarithm problem (ECDLP) in $\mathbb{G}$ and logarithm problem in the extension field $\mathbb{F}_{q^k}$, that is, $\mathbb{G}_T$. For a pairing-based cryptosystem that requires 1024bit security, the size of the extension field $\mathbb{F}_{q^k}$ should at least be 1024 bits long and the group order of $\mathbb{G}$ should at least be 160 bits long [18].

In most composite-order pairing-based cryptosystems, their security also relies on the intractability of a problem called *Subgroup Decisional Problem* (SDP) [5]: for a bilinear map $\hat{e} : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ of composite order l, without knowing the factorization of the group order l, the SDP is to decide if an element x is in a subgroup of $\mathbb{G}$ or in $\mathbb{G}$. For the intractability of SDP, the group order l of $\mathbb{G}$ should be at least 1024 bits long. As $l|q+1$, q should also be at least 1024 bits long. As the embedding degree k is 2, the size of the extension field $\mathbb{F}_{q^2}$ is at least 2048 bits long.

4   NVIDIA's CUDA Framework

NVIDIA's CUDA consists a set of software tools (e.g., CUDA Toolkit) and the graphics card hardware. CUDA facilitates the design and implementation of general-purpose programs on NVDIA's graphics cards. A program includes one kernel function written in the CUDA C/C++ language (i.e., an extension to C/C++ language designed for CUDA) in a .cu file. This .cu file is complied via NVCC (i.e., NVidia Cuda Compiler), which sends the host code to the native GCC/Visual C++ Compiler and compiles the GPU code (e.g., the kernel function) into a NVIDIA's virtual machine code (PTX) and in turn the graphics card's machine code (.cubin). NVCC also combines the compiled host and GPU codes to finally generate a single host executable file which includes the host code on how to launch the kernel's GPU machine code.

Currently, there are many NVIDIA's graphic cards supporting CUDA [3]. In our experiments, we use GeForce GTX 285 (240 CUDA cores) and GTX 480 (480 CUDA cores) cards which are the top-end products of the second and third generations ("Fermi") of CUDA-enable graphic cards separately. A CUDA-enable graphic card contains tens of Streaming Multiprocessors (SMs) of which each could run hundreds of threads seemingly concurrently. In fact, a SM schedules those threads based on a group of 32 threads (called a "warp"). At one time, a warp of 32 threads is active and those 32 threads will be mapped to the 8 (in GTX 285) or 32 (in GTX 480) CUDA cores (i.e., the physical processing units) to execute. As there is only one instruction decoding unit available for each SM, all 32 threads within one warp should share the same instruction, otherwise the divergent instructions will be serialized.

At the logical level of the design, programmers can define the grid size and the block size for their own kernel function. The grid size defines how many blocks within one grid run for this kernel function and block size defines how many threads are within each block. CUDA guarantees that threads within the same block can communicate and will execute on the same SM. The logical design hides the difference between graphic cards and the different power between different GPUs. For example, without changing the code, a card with a larger number of SMs would run more blocks each time, resulting in a better performance automatically.

Each thread can also access a few (private) register files while threads within each block can access to the pre-block "shared memory". The device (global) memory, located off-chip, is the largest available memory that can be read/written by all threads, however, its access time is 400-600 times higher, compared to the registers and the shared memory. Constant and texture memory is also located on the device memory with special 1D and 2D caches available. A CUDA program should carefully choose which memory to use to achieve an optimized performance. For more details on CUDA architecture and programming with CUDA, the reader can refer to [19].

---

[3] `http://www.nvidia.com/object/cuda_gpus.html`

5   Arithmetic Operations

The arithmetic operations required by a bilinear pairing are the operations in the extension field $\mathbb{F}_{q^2}$ and the elliptic curve $E(\mathbb{F}_q)$ which are in turn based on the base field operations in $\mathbb{F}_q$. In this section, we first describe the algorithms for based field operations and then algorithms on the extension field and elliptic curve.

Specifically, the operations in $\mathbb{F}_{q^2}$ include multiplication $a \times b$ and square a^2 where $a, b \in \mathbb{F}_{q^2}$. The operations in $E(\mathbb{F}_q)$ are double $2P$ and addition $P + Q$ where $P, Q \in E(\mathbb{F}_q)$. The operations in $\mathbb{F}_q$ considered in this paper include multiplication, addition and subtraction.

The multiplication inverse in $\mathbb{F}_q$ is expensive in our GPU implementation, which motivates us to avoid it. However, there are two occasions which may require multiplication inverse. One is in the addition and double operations of $E(\mathbb{F}_q)$. This can be avoided by using a projective coordinate system to represent elliptic curve points and we do so. The second one is in the final powering of bilinear pairing. However, we identify that the final powering is not a bottleneck of the whole system. In fact, through the experiments, we find that the final powering is 500+ times faster than the Miller's loop on the CPU. Therefore, we can leave the work of final powering (and therefore multiplication inverse in $\mathbb{F}_q$) to the CPU.

Furthermore, cryptographic constructions may only require the result of a product of bilinear pairings [12]. In this case, we can calculate the multiple pairings result (without the final power) on the GPU, then multiply them and do the single final powering to get the result. In this way, the cost to compute the final powering would be even ignored.

5.1   Base Field Operations

Motivated by the feasibility of performing fast and parallelized operations on multi-core graphics hardware, we choose to represent the base field elements of $\mathbb{F}_q$ in Residue Number System (RNS). In RNS, an n-length vector $\boldsymbol{a} = (a_1, a_2, ..., a_n)$ is chosen such that $\gcd(a_i, a_j) = 1$ for all $i \neq j$ and $q < A$ where $A = \prod_{i=1}^{n} a_i$ is called the dynamic range of $\boldsymbol{a}$. For any x, $0 \leq x \leq q$, it can be represented uniquely in RNS as $\langle x \rangle_a = (x \bmod a_1, x \bmod a_2, \ldots, x \bmod a_n)$, and recovered uniquely in the form of $x \bmod A$ due to the Chinese Remainder Theorem.

The purpose of using RNS is to break down some basic arithmetic operations that include $\odot \in \{+, -, \times\}$ to small pieces which can be parallelized and computed using the multiple cores of the GPU. That is, $\langle x \rangle_a \odot \langle y \rangle_a = ((x_1 \odot y_1) \bmod a_1, \ldots, (x_n \odot y_n) \bmod a_n)$ where $\langle x \rangle_a = (x_1, \ldots, x_n)$ and $\langle y \rangle_a = (y_1, \ldots, y_n)$. Note that division (and therefore multiplication inverse in $\mathbb{F}_q$) and comparison in RNS are non-trivial and usually avoided from using as they do not offer speed advantage over conventional methods.

It is known that the multiplication operation on $\mathbb{F}_q$ can be done in RNS using the RNS Montgomery multiplication algorithm (see [13]). But there are few papers dealing with addition and subtraction on $\mathbb{F}_q$ in RNS. If we see the RNS Montgomery multiplication algorithm as the first step to compute multiplication (the second step is the mod q operation), we can find a uniform way to handle addition and subtraction in RNS as well. Basically, given two elements $a, b \in \mathbb{F}_q$, we calculate addition $a + b$, subtraction $a - b$ and multiplication $a \times b$ without any modular operations. The result may grow up; when it becomes larger than a threshold, we employ an explicit modular reduction (i.e., $\bmod q$) to bring back the result to the allowed range again. This idea makes the operations in base field $\mathbb{F}_q$ simple and clear. Moreover, since the first step addition, subtraction and multiplication are cheap in RNS, this method allows us to fully focus on the most expensive part; that is, the second step: modular reduction.

To perform modular reduction, we employ the Montgomery Modular Reduction algorithm in RNS. **Algorithm 1** shows the algorithm (derived from [13, Alg. 3], as we discussed). In the algorithm, the dynamic ranges of bases $\boldsymbol{a}$ and $\boldsymbol{b}$ are denoted as A and B, respectively.[4] Also note that the output of **Algorithm 1** is $sB^{-1}(\bmod q)$ where the component B^{-1} should be removed in the conventional way of using the Montgomery Multiplication algorithm (see [13]).

Algorithm 1: Montgomery Modular Reduction in Residue Number Systems [13]

> **Input**: $\langle s \rangle_{a \cup b}$.
> **Output**: $\langle w \rangle_{a \cup b}$, where $w < 2q$ and $w \equiv sB^{-1} \pmod{q}$.
> **Ensure**: $\gcd(B, q) = 1$, $\gcd(A, B) = 1$, $4q \le B$ and $2q \le A$.

1   $\langle t \rangle_b \leftarrow \langle s \rangle_b \cdot \langle -q^{-1} \rangle_b$ $\langle t \rangle_{a \cup b} \Leftarrow \langle t \rangle_b$;
2   $\langle u \rangle_a \leftarrow \langle t \rangle_a \cdot \langle q \rangle_a$;
3   $\langle v \rangle_a \leftarrow \langle s \rangle_a + \langle u \rangle_a$;
4   $\langle w \rangle_a \leftarrow \langle v \rangle_a \cdot \langle B^{-1} \rangle_a$ $\langle w \rangle_a \Rightarrow \langle w \rangle_{a \cup b}$;
5   **return** $\langle w \rangle_{a \cup b}$

The symbol $\Rightarrow$ (or $\Leftarrow$) represents a base extension algorithm [21,11]. Given an RNS representation $\langle x \rangle_c$, this algorithm outputs $\langle x \rangle_d$ for $\boldsymbol{d} \ne \boldsymbol{c}$. The two base extensions $\langle t \rangle_{a \cup b} \Leftarrow \langle t \rangle_b$ and $\langle w \rangle_a \Rightarrow \langle w \rangle_{a \cup b}$ are the most computationally expensive parts of **Algorithm 1**. The following theorem states the correctness of **Algorithm 1**.

Theorem 1. *For any integer s such that $0 \le s < \alpha q^2$, **Algorithm 1** outputs w such that $0 \le w < 2q$ if $B > \alpha q$ and $A > 2q$.*

Proof. From **Algorithm 1**, we have

$$
\begin{aligned}
w &= \frac{v}{B} = \frac{s + tq}{B} < \frac{\alpha q^2 + Bq}{B} \\
w &< q \cdot \left(\frac{\alpha q}{B} + 1 \right) < 2q.
\end{aligned}
\tag{5}
$$

Therefore, when the result of $a\{+, -, \times\}b$ grows beyond threshold αq^2, we can reduce it back to $w < 2q$. Furthermore, we can control parameter α, such to trade off between the number of reductions and the number of threads; a larger α results a larger threshold, but $B > \alpha q$ will be larger as well, requiring a larger number of bases to represent.

5.2   Extension Field Operations

Given an element $a \in \mathbb{F}_{q^2}$, a can be written as $x + iy$ where $x, y \in \mathbb{F}_q$ and $i^2 = -1$. The multiplication $a \times b$:

$$
a \times b = (x_1 + iy_1)(x_2 + iy_2) = (x_1 x_2 - y_1 y_2) + i(x_1 y_1 + x_2 y_2)
$$

which requires two reductions with four cheap multiplications and two cheap additions in RNS. Since the number of reductions meets with the lower bound (two), we do not resort to more advanced methods (e.g., Karatsuba multiplication). Similarly, squaring a^2 requires two reductions as well.

$$
a^2 = (x_1^2 - y_1^2) + i2x_1 y_1
$$

---

[4] We refer readers to [1] for information on how to choose a good base $\boldsymbol{a}$ and $\boldsymbol{b}$.

5.3   Elliptic Curve Operations

As we discussed, we adopt the Jacobian projective coordinate system for representing points in elliptic curve to avoid multiplication inverse in $\mathbb{F}_q$. A point $P = (X, Y, Z)$ in Jacobian projective coordinates can be mapped to $(\frac{X}{Z^2}, \frac{Y}{Z^3})$ in affine coordinates.

Let $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2, Z_2)$ be two points in $E(\mathbb{F}_q)$. Below is the formula from [7] for computing double, that is, $R = 2P = (X_3, Y_3, Z_3)$:

$$X_3 = T, \quad Y_3 = -8Y_1^4 + M(S - T), \quad Z_3 = 2Y_1 Z_1$$

where $S = 4X_1 Y_1^2$, $M = 3X_1^2 + Z_1^4$, $T = -2S + M^2$. Addition $R = P + Q = (X_3, Y_3, Z_3)$:

$$X_3 = -H^3 - 2U_1 H^2 + r^2, \quad Y_3 = -S_1 H^3 + r(U_1 H^2 - X_3), \quad Z_3 = Z_1 Z_2 H$$

where $U_1 = X_1 Z_2^2$, $U_2 = X_2 Z_1^2$, $S_1 = Y_1 Z_2^3$, $S_2 = Y_2 Z_1^3$, $S_2 = Y_2 Z_1^3$, $H = U_2 - U_1, r = S_2 - S_1$.

In our implementation, to make the addition formula simpler, Q is given in affine coordinates (X_2, Y_2). Equivalently, we can view it as $Q = (X_2, Y_2, 1)$ in Jacobian projective coordinates. Hence the formula above can be refined as follows:

$$X_3 = -H^3 - 2X_1 H^2 + r^2, \quad Y_3 = -Y_1 H^3 + r(X_1 H^2 - X_3), \quad Z_3 = Z_1 H$$

where $H = X_2 Z_1^2 - X_1$ and $r = Y_2 Z_1^3 - Y_1$.

As in the previous section, we are interested to find patterns like $\sum A_i B_i$ in operations, to minimize the number of modular reductions. The refined formulas to compute addition and double in $E(\mathbb{F}_q)$ are shown in Table 1.

Table 1: $E(\mathbb{F}_q)$ Operations

$2(X_1, Y_1, Z_1, Z_1^2)$	$(X_1, Y_1, Z_1) + (X_2, Y_2)$
Y_1^2	$H = X_2 Z_1^2 - X_1$
$S = 2Y_1^2 X_1$	$e_0 = Y_2 Z_1$
$M = (Z_1^2)^2 + 3X_1^2$	$r = Z_1^2 e_0 - Y_1$
$X_3 = T = M^2 - 2S$	$H^2 = (H)^2$
$Y_3 = -MT + MS - 8(Y_1^2)^2$	$X_3 = (r)^2 - (HH^2) - 2X_1 H^2$
$Z_3 = 2Y_1 Z_1$	$e_1 = -X_3 + X_1 H^2$
$Z_3^2 = (Z_3)^2$	$e_2 = Y_1 H$
	$Y_3 = e_1 r - e_2 H^2$
	$Z_3 = Z_1 H$
	$Z_3^2 = (Z_3)^2$

6   Bilinear Pairing Algorithms

Based on the operations above, in this section, we describe the bilinear pairing algorithm itself. **Algorithm 2** shows the Barreto et al.'s algorithm [3] to compute bilinear pairings. The algorithm is described specifically for composite-order bilinear pairing in Eq. (4).

Algorithm 2: Barreto et al.'s Algorithm [3]

Input: $P, Q \in \mathbb{G}$.
Output: $\hat{e}(P, Q) = e_n(P, \phi(Q))$ where $n = \#E(\mathbb{F}_q) = q + 1$.
Ensure: $E : y^2 = x^3 + x$ over $\mathbb{F}_q$; $q > 3$, $q \equiv 3 \pmod{4}$; $\phi(x, y) = (-x, iy) \in E(\mathbb{F}_{q^2})[n]$ for
      $(x, y) \in E(\mathbb{F}_q)[n]$, $i \in \mathbb{F}_{q^2}$, $i^2 = -1$.

1 Let $n = (n_t, ..., n_0)$, $n_i \in \{0, 1\}$ and $n_t = 1$;
2 Set $f \leftarrow 1$ and $V \leftarrow P$;
3 for $i \leftarrow t - 1$ **to** 0 **do**
4   Set $f \leftarrow f^2 \cdot g_{V,V}(\phi(Q))$;
5 $V \leftarrow 2V$;
6   **if** $i = 0$ **then**
7 | break ;
8   **end**
9   **if** $n_i = 1$ **then**
10 | Set $f \leftarrow f \cdot g_{V,P}(\phi(Q))$;
11 | $V \leftarrow V + P$;
12   **end**
13 end
     ; // $f^{(q^2-1)/n}$ `The final powering`
14 return f

Algorithm 3: $g_{U,V} \cdot \phi$ [20]

Input: $U = (x_1, y_1, z_1, z_1^2) \in E(\mathbb{F}_q)[n]$ in Jacobian projective coordinate ; $V = (x_2, y_2)$ and
      $Q = (x_3, y_3) \in E(\mathbb{F}_q)[n]$ in affine coordinate where $n = \#E(\mathbb{F}_q)$.
Output: $\hat{G} \in \mathbb{F}_{q^2}$.
Ensure: $E : y^2 = x^3 + x$ over $\mathbb{F}_q$; $q > 3$, $q \equiv 3 \pmod{4}$; $\phi(x, y) = (-x, iy) \in E(\mathbb{F}_{q^2})[n]$ for
      $(x, y) \in E(\mathbb{F}_q)[n]$, $i \in \mathbb{F}_{q^2}$, $i^2 = -1$.

1 if $U \neq V$ *(either* $\frac{x_1}{z_1^2} \neq x_2$ *or* $\frac{y_1}{z_1^3} \neq y_2$*)* **then**
     | /* $ax + by + c = 0$ is the line that goes through U and V.   */
2 | $a \leftarrow y_1 - z_1^3 y_2$;
3 | $b \leftarrow z_1^3 x_2 - z_1 x_1$;
4 | $c \leftarrow -(ax_2 + by_2)$;
5 end
6 else
     | /* $ax + by + c = 0$ is the tangent to the elliptic curve $E(\mathbb{F}_q)$ at the point U.   */
7 | $a \leftarrow -(3x_1^2 + z_1^4)z_1^2$;
8 | $b \leftarrow 2y_1 z_1^3$;
9 | $c \leftarrow -(2y_1^2 + ax_1)$;
10 end
11 return $(c - ax_3) + by_3 i$.

Line 5 and 11 in the algorithm are the double and addition in $E(\mathbb{F}_q)$; lines 4 and 10 are the multiplication operations in $\mathbb{F}_{q^2}$ which all have been discussed in the previous section. The function $g_{U,V} \cdot \phi : E(\mathbb{F}_q) \rightarrow \mathbb{F}_{q^2}$ is shown in **Algorithm 3**.

The flow of computations in **Algorithm 2** and **Algorithm 3** only depends on the system parameters (e.g., $n = q + 1$) but not on the input points. Since these two algorithms fit well with the SIMD fashion of a GPU, we do not further refine the bilinear pairing algorithms.

7   Implementation and Analysis

In this section, we discuss how the previous presented algorithms are mapped to CUDA programming model. Specifically, we discuss what data structures that we use to represent base field, extension field and elliptic curve elements. We also describe the building algorithms for single thread and how we store variables and constants onto GPUs.

In this paper, we consider 1024/2048bit composite order. As the word length in GPU is 32 bits, we need at least 1024/32=32 (64) bases to represent a 1024/2048bit number (i.e., $\mathbb{F}_q$ element) in RNS. Each base is handled by one thread. In fact, the number 32/64 only acts a lower bound, the least number we can choose is 33/65. We also need another set of bases for the base extension operation, therefore, the total number of bases to represent one $\mathbb{F}_q$ element is 33+33+1=67 (65+65+1=131). The additional base comes from the Shenoy's base extension algorithm. Hence, each $\mathbb{F}_q$ element is mapped to a block of 67 (and 131) threads and the data structure to represent one $\mathbb{F}_q$ element is simply a 32bit unsigned integer (UINT32).

We do not consider parallelism within the operations of extension field and elliptic curve, as our goal in this paper is to compute as many as possible pairings at one time (a typical goal in a server setting). Therefore, we build extension field and elliptic curve directly on the base field. The data structure for the extension field elements consists of a two-dimension vector (x, y) where x, y are UINT32. The data structure to represent elliptic curve points in projective coordinates consists of (x, y, z, z^2) of UINT32s. The bilinear pairing algorithm is straightforwardly built upon the base field, extension field and elliptic curve operations. Therefore, each block handles one pairing calculation. Our grid and block arrangements simplify the design. Specifically, the base field operations consist of two parts. One is to compute addition $a + b \bmod m$, subtraction $a - b \bmod m$ and multiplication $a \times b \bmod m$ for the base $m < 2^{32}$. The other is to do Montgomery modular reduction via base extension. To compute $a + b \bmod m$ (similarly, $a - b \bmod m$), there are two cases: $a + b < m$ and $m \le a + b < 2m$ where we assume that $0 \le a, b < m$. In the second case, we need to output $a + b - m$ as the result. However, this case handling, depending on the input values, causes a branch divergence on the GPU (since GPU is SIMD). To minimize the divergence, we compute both $u = a + b$ and $v = u - m$ first, no matter what the inputs are. Then, we do the condition test and output u or v accordingly, where now the divergence is minimized as the output operation.

The multiplication $a \times b \bmod m$ follows the method in [2,21]. Given $0 \le a, b < m$, let $d = 2^{32} - m$ and $p = ab = p_h 2^{32} + p_l$. Then,

$$
\begin{aligned}
p &= p_h(d + m) + p_l \\
p &\quad \bmod m \equiv p_h d + p_l
\end{aligned}
\tag{6}
$$

If m is large enough, then $d = 2^{32} - m$ will be quite small and $p_h d$ will be smaller than $p_h 2^{32}$. Following this direction, we can further reduce $p_h d$ and prove that $p \bmod m \equiv d u_h + u_l + p_l$ and $d u_h + u_l + p_l < 2m$, where $p_h d = u_h 2^{32} + u_l$. We also note that CUDA does not provide direct functions to output the lowest 32bit of two 32bit number multiplication and the NVCC compiler does not do a good job when translating C code $a \times b$ into a proper PTX code. Here we use the method in [22] that hardcodes a proper PTX multiplication code as "asm("mul.wide.u32 %0, %1, %2:": "=l"(p), "=r"(a), "=r"(b))" which has a better performance than NVCC.

The memory is allocated as follows. The basic idea is that we (try to) store all variables to the register file of their threads such that the access time to them can be ignored. We also store 67 (and 131) bases and those one-dimensional precomputed values in the constant memory to

facilitate its 1D cache. Although the time for the fist access to them is large (400-600 cycles), the overall access time could be small as the algorithms and their threads fetch them frequently. For example, in each algorithm, the first thing is to load the associated base of that thread to the register. We also store the 2D array of the base extension algorithm to the texture memory so that we can benefit from the spatial locality and the 2D cache of the texture memory. Through the CUDA profiler's report, it indeed exploits caching well and the cache-hit rate is very high.

8   Experimental Results

The experiments were conducted on NVIDIA GeForce GTX 285, GTX 480 and Amazon EC2 Cloud Computing[5] Cluster GPU Instances (equipped with two Tesla M2050). The detailed system configurations are shown in Table. 2. For comparison, we also choose Pairing-Based Cryptography (PBC) library version 0.5.11 (built upon GMP library[6] version 5.0.1) as the benchmark that runs on Intel Core 2 E8300 CPU at 2.83GHz and 3GB memory. Through the experiments, we choose random points $P, Q \in E(\mathbb{F}_q)$ as the input to evaluate $\hat{e}(P, Q)$.

Table 2: Experimental Configurations

GPU Model	CUDA Cores	Clock (GHz)	Graphical Memory (GB)	Compute Capability
GTX 285	240	1.476	1	1.3 (GT200)
GTX 480	480	1.4	1.5	2.0 (Fermi)
Amazon EC2	448 (x2)	1.15	3 (x2)	2.0 (Fermi)

We first compare the running time on CPU and GPUs. The results are shown in Fig. 1. The GPUs method seems not to have advantage when the number of pairings is small (< 32), as the hardware is not fully occupied. With the number becoming larger, the speedup in running time increases. This indicates that the GPUs method is especially suitable for the case that multiple composite-order pairings should be evaluated at the same time.

Specifically, in the 1024bit security level, GTX 285, M2050 (Amazon EC2) and GTX 480 achieve a running time of 17.4ms, 11.9ms and 8.7ms per pairing respectively, which is 9.6, 14.3 and 19.6 times faster respectively compared to the state-of-the-art CPU implementation (171.1ms per pairing). We note that this result has been comparable with prime-order pairing implementation on CPU (see the dash lines in Fig. 1). Where both A and D179 [20] pairing are for 1024bit security and A is the fastest. With a 2.1 USD charge per hour, 11.9ms on Amazon EC2 also means that the cost to compute a pairing is as low as $(2.1 \times 11.9)/(60 \times 60 \times 1000) = 7 \times 10^{-6}$ USD. For example, assuming that the CPU machine in our experiments is with 400 USD price, such a low cost means this machine should continuously for 2.65 year to recover the cost.

In a higher 2048bit security level, the speedup on GTX 480 is even more than 24x (48.9ms per pairing, compared with 1189.8ms per pairing on CPU). GTX 280 and M2050 also achieve high speedups of 12 (98.2ms per pairing) and 21 (54.7ms per pairing) times at such a security level, which suggests that our method is promising for higher security levels.
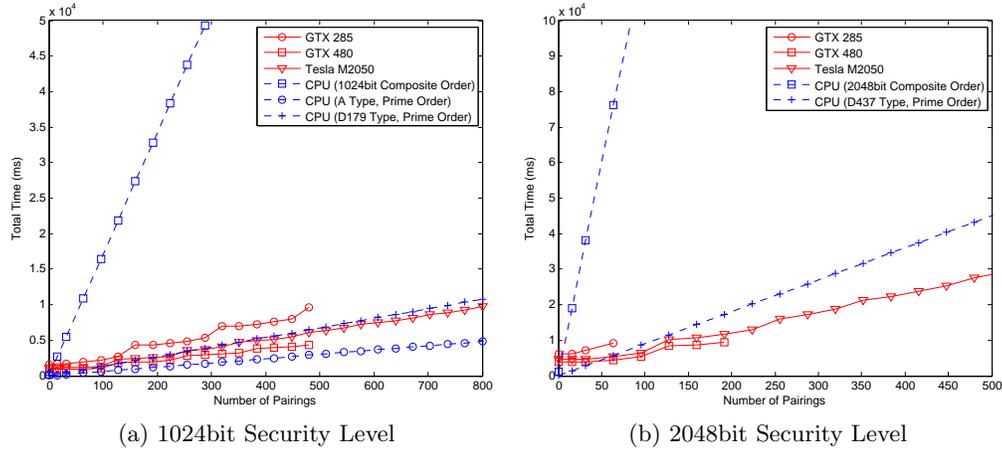
---

[5] http://aws.amazon.com/ec2/
[6] http://gmplib.org/

(a) 1024bit Security Level   (b) 2048bit Security Level

Fig. 1: Running Time on different GPUs and CPU

8.1   Number of Registers vs. Optimized Average Running Time

In the next experiment, we analyze how the implementation parameters impact the performance. We record the running time by changing the maximum number of registers that one thread can use, from 16 to 32 to find which allocation minimizes the average running time per pairing. The results are shown in Table. 3 for GTX 285. In Table. 3, the number of pairings which achieves the optimized average running time for each different maximum numbers of registers (16–32) can be predictable. In fact, we show in Fig. 2 the predicted and experimentally recorded running time, for the case where the maximum number of registers per thread is set to 26. We can see that the optimized average time only reaches 120 [7] (and 240, ...) where a big skip appears, which indicates that all hardware resource is occupied.

Table 3: Optimized Average Running Time

Number of Registers	16	17	18	19	20	21
Number of Pairings	240	180	180	180	180	150
Average Time (ms)	18.5	17.8	17.7	17.5	**17.4**	17.8
Number of Registers	22	23	24	25	26	27
Number of Pairings	150	150	150	120	120	120
Average Time (ms)	17.8	17.8	18.4	19.3	19.4	19.2
Number of Registers	28	29	30	31	32	
Number of Pairings	120	120	120	120	120	
Average Time (ms)	19.0	20.0	N/A	22.5	22.0	

(a) 1024bit Security Level

Number of Registers	16	17	18	19	20	21
Number of Pairings	150	120	120	120	120	120
Average Time (ms)	N/A	99.0	99.1	98.6	98.4	**98.2**
Number of Registers	22	23	24	25	26	27
Number of Pairings	90	90	90	90	90	60
Average Time (ms)	101.0	101.2	105.4	102.7	103.7	127.7
Number of Registers	28	29	30	31	32	
Number of Pairings	60	60	60	60	60	
Average Time (ms)	123.7	127.5	N/A	147.3	145.2	

(b) 2048bit Security Level

---

[7] This number can be digged out by CUDA GPU Occupancy Calculator, `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.
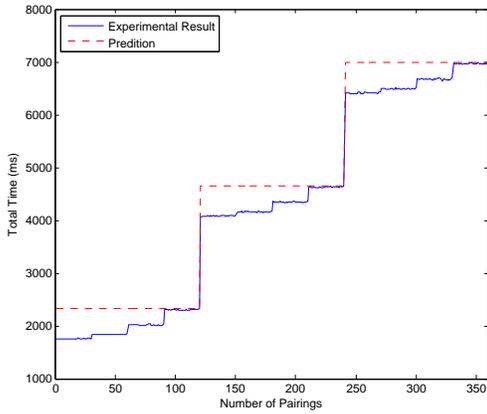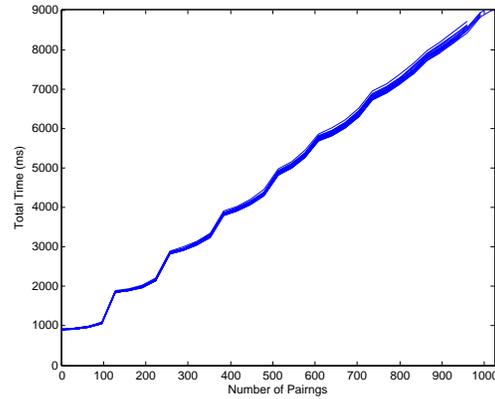
Fig. 2: Prediction of Running Time

Fig. 3: Running Time vs. Number of Registers

Different numbers of registers per thread have no impact on Fermi's architecture (Fig. 3) (GTX 480, M2050). we believe that this is because local memory is cached by its L1/L2 cache.

8.2 Effect of Unrolling

We also unroll the for loops inside the base extension algorithms. The results are shown in Table. 4. We do not test on unrolling 8 or more loops on GTX 285 as the compiler fails at the configurations. The experimental results suggest that unrolling at 1024/2048bit security levels does not speed up the performance obviously. Therefore, our implementation is optimized at these levels.

Table 4: Unrolling v.s. Performance

	Unroll 1 (ms)	Unroll 2 (ms)	Unroll 4 (ms)	Unroll 8 (ms)	Unroll 16 (ms)
GTX480, 1024bit	2208.0	2227.2	2406.2	2801.3	2991.0
GTX480, 2048bit	10677.7	10399.3	11639.5	12994.5	13762.9
GTX285, 1024bit	3132.0	3225.9	3205.1	–	–

9 Conclusions

This paper is a thorough study on how to compute bilinear pairing using graphics card hardware. To fully utilize the thousands of threads on GPU, we choose RNS system to represent elements in base field $\mathbb{F}_q$. Based on RNS, we further implement the arithmetic operations on $\mathbb{F}_{q^2}$ and $E(\mathbb{F}_q)$, and the bilinear pairing algorithm itself. Experimental results show that our implementation is much faster than state-of-the-art CPU implementation to compute composite-order pairings and is comparable with the prime-order CPU implementation. Specifically, it achieves a record of 8.7ms per pairing, which is 19.6 time faster compared with (composite-order) CPU implementation in the 1024bit security level. At a 2048bit level, the speedup is even higher (24 times). We

also conduct experiments on a cloud computing environment (Amazon EC2), which suggests a low-cost record of 7×10^{-6} USD per pairing. We should also note that our implementation is generally valid for prime-order pairings as well.

References

1. J. C. Bajard, S. Duquesne, M. Ercegovac, and N. Meloni. Residue systems efficiency for modular products summation: application to elliptic curves cryptography. *Advanced Signal Processing Algorithms, Architectures, and Implementations XVI*, 6313(1):631304, 2006.
2. J.-C. Bajard, N. Meloni, and T. Plantard. Efficient rns bases for cryptography. In *IMACS'2005 World Congress*, 2005.
3. P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In *Advances in Cryptology - CRYPTO 2002*, pages 354–368. Springer, 2002. LNCS 2442.
4. D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. Ecm on graphics cards. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '09, pages 483–501, Berlin, Heidelberg, 2009. Springer-Verlag.
5. D. Boneh, E. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Proc. of Theory of Cryptography (TCC) '05*, pages 325 – 341. Springer, 2005. LNCS 3378.
6. D. Boneh, A. Sahai, and B. Waters. Fully collusion resistant traitor tracing. Cryptology ePrint Archive, Report 2006/045, 2006. `http://eprint.iacr.org/`.
7. H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *Advances in Cryptology - ASIACRYPT '98*, pages 51 – 65. Springer, 1998. LNCS 1514.
8. S. Fleissner. Gpu-accelerated montgomery exponentiation. In Y. Shi, G. van Albada, J. Dongarra, and P. Sloot, editors, *Computational Science C ICCS 2007*, volume 4487 of *Lecture Notes in Computer Science*, pages 213–220. Springer Berlin / Heidelberg, 2007.
9. D. Freeman. Converting pairing-based cryptosystems from composite-order groups to prime-order groups. In H. Gilbert, editor, *Advances in Cryptology C EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 44–61. Springer Berlin / Heidelberg, 2010.
10. N. Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 48–64. Springer Berlin / Heidelberg, 2010.
11. O. Harrison and J. Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In *AFRICACRYPT '09: Proceedings of the 2nd International Conference on Cryptology in Africa*, pages 350–367, Berlin, Heidelberg, 2009. Springer-Verlag. LNCS 5580.
12. J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *EUROCRYPT'08: Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology*, pages 146–162, Berlin, Heidelberg, 2008. Springer-Verlag.
13. S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-rower architecture for fast parallel montgomery multiplication. In *Advances in Cryptology - EUROCRYPT 2000*, pages 523 – 538. Springer, 2000. LNCS 1807.
14. A. Lewko, T. Okamoto, A. Sahai, K. Takashima, and B. Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In H. Gilbert, editor, *Advances in Cryptology C EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 62–91. Springer Berlin / Heidelberg, 2010.
15. S. Meiklejohn, H. Shacham, and D. Freeman. Limitations on transformations from composite-order to prime-order groups: The case of round-optimal blind signatures. In M. Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 519–538. Springer Berlin / Heidelberg, 2010.
16. V. S. Miller. Short programs for functions on curves. unpublished manuscript available at http://crypto.standford.edu/miller/miller.pdf.
17. A. Moss, D. Page, and N. Smart. Toward acceleration of rsa using 3d graphics hardware. volume 4487 of *Lecture Notes in Computer Science*, pages 364–383. Springer Berlin / Heidelberg, 2007.
18. NIST. Recommendation for key management, 2007.
19. NVIDIA Corporation. Nvidia cuda c programming guide, 2010. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`.

20. PBC Library. The pairing-based cryptography library. http://crypto.stanford.edu/pbc/.
21. R. Szerwinski and T. Güneysu. Exploiting the power of gpus for asymmetric cryptography. In *CHES '08: Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems*, pages 79–99, Berlin, Heidelberg, 2008. Springer-Verlag. LNCS 5154.
22. K. Zhao. Implementation of multiple-precision modular multiplication on gpu, 2009. `http://www.comp.hkbu.edu.hk/~pgday/2009/10th_papers/kzhao.pdf`.