Acceleration of Composite Order Bilinear Pairing on Graphics Hardware

Ye Zhang[1,⋆], Chun Jason Xue[2], Duncan S. Wong[2],
Nikos Mamoulis[3], and Siu Ming Yiu[3]

[1] Pennsylvania State University, USA
yxz169@cse.psu.edu
[2] City University of Hong Kong, Hong Kong
{jasonxue,duncan}@cityu.edu.hk
[3] University of Hong Kong, Hong Kong
{smyiu,nikos}@cs.hku.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/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 80bit security level, which is a 20-fold speedup compared to the 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. For example, we can achieve 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, specifically in functional encryption schemes, e.g., [2,5,7]. On the other hand, evaluating a pairing over a composite-order group is much more expensive compared to its prime-order counterpart. To achieve the

⋆ Part of this work was done while the author was with the University of Hong Kong.

T.W. Chim and T.H. Yuen (Eds.): ICICS 2012, LNCS 7618, pp. 341–348, 2012.

same 80bit (AES) security level, the composite order should be at least 1024 bit to be difficult to factorize, while a much smaller prime order (e.g., 160bit) is enough. As a result, the underlying finite field, elliptic curve operations and the pairing evaluating algorithm itself become much slower. An estimation [3] 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 multiple such pairings need to be evaluated at the same time (e.g., decryption algorithm in the scheme [5]). Furthermore, one typical scenario of functional encryption schemes is the outsourced database scenario where the database server needs to decrypt the whole encrypted data with particular decryption key that embeds the query predicate. As a result, the database needs to evaluate mass amount of composite-order parings as fast as possible.

There are some efforts to address this problem. Freeman [3] 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. [8] points out that some schemes inherently require composite-order groups and cannot be transformed mechanically by using Freeman's method.

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) [6]. Multiplication and square operations on extension field $\mathbb{F}_{q^2}$ and addition and double operations on an 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 [1] 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 optimized 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 80bit security level, compared to the state-of-the-art implementation [9] 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. The arithmetic operations and algorithms are presented in Section 2. Section 3 discusses the implementation considerations on mapping the algorithms. The experimental results are shown in Section 4.

2 Arithmetic Operations

We employ Barreto et al.'s algorithm [1] to evaluate bilinear pairing. Its details (including the algorithm to evaluate $g_{U,V}$) can be found in the full version [11] of this paper, which is also specifically designed for the composite-order pairing. We note that we choose Barreto et al.'s algorithm because the flow of computations in it only depends on the system parameters but not on the input points. Therefore, their algorithm fit well with SIMD fashion of GPUs.

The arithmetic operations required by Barreto et al.'s algorithm 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 operations in the base field $\mathbb{F}_q$. Specifically, given $a, b \in \mathbb{F}_{q^2}; P, Q \in E(\mathbb{F}_q)$, we consider $a \times b$, a^2, $P + Q$ and $2P$ operations.

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 [5]. 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.

2.1 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 [6]). 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 [6, 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.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 [6]).

Algorithm 1. Montgomery Modular Reduction in Residue Number Systems [6]

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: $\gcd(B, q) = 1$, $\gcd(A, B) = 1$, $4q \leq B$ and $2q \leq A$.

1 $\langle t \rangle_b \leftarrow \langle s \rangle_b \cdot \langle -q^{-1} \rangle_b$ $\quad\quad$ $\langle t \rangle_{a \cup b} \Leftarrow \langle t \rangle_b$
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 $\langle w \rangle_a \leftarrow \langle v \rangle_a \cdot \langle B^{-1} \rangle_a$ $\quad\quad$ $\langle w \rangle_a \Rightarrow \langle w \rangle_{a \cup b}$
5 **return** $\langle w \rangle_{a \cup b}$

The symbol $\Rightarrow$ (or $\Leftarrow$) represents a base extension algorithm [10,4]. Given an RNS representation $\langle x \rangle_c$, this algorithm outputs $\langle x \rangle_d$ for $\boldsymbol{d} \neq \boldsymbol{c}$. 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 (whose proof is given in the full version [11]) states the correctness of **Algorithm 1**.

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

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.

2.2 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 $a \times b$:

$$a \times b = (x_1 + iy_1)(x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_1 + x_2y_2)$$

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.

$$a^2 = (x_1^2 - y_1^2) + i2x_1y_1$$

2.3 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. As we will often use Z^2, we store Z^2 in the coordinates as well and we call this modified Jacobian coordinates: (X, Y, Z, Z^2). To make the addition formula simpler, Q is also given in affine coordinates (X_2, Y_2).

As in the previous section, we are interested to find patterns like $\sum A_iB_i$ 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 provided that $P = (X_1, Y_1, Z_1, Z_1^2)$ and $Q = (X_2, Y_2)$.

3 Implementation and Analysis

In this section, we discuss how the previous presented algorithms are mapped to CUDA programming model. In CUDA programming model, programmers can define the block size for their own kernel function. The block size defines how many threads are within each block. CUDA guarantees that threads in the same block can communicate and will execute on the same physical SM (streaming Multiprocessors). A detailed description on SM can be found in the full version.

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

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

In this paper, we consider 1024/2048 bit composite order (w.r.t. 80/112bit security levels). As the word length in GPU is 32 bits, we need at least 32/64 bases to represent a 1024/2048bit number in RNS. In fact, the least number we can choose is 33/65. To complete Montgomery modular reduction (Algorithm 1), we need additional 32/64 bases. Moreover, we employ Shenoy's base extension algorithm which requires one more base. Therefore, for the 80bit security level, we need 33+33+1=67 bases to represent a singel element in $\mathbb{F}_q$. Specifically, each $\mathbb{F}_q$ element is mapped to a block of 67 threads. For each thread, a 32bit unsigned integer (UINT32) is used to represent the element (under the particular base of that thread).

We don't consider parallelism within the operations in the $\mathbb{F}_{q^2}$ and $E(\mathbb{F}_q)$, as our goal is to compute as many as possible pairings at one time (a typical goal in the server setting). Therefore, we simply represent $\mathbb{F}_{q^2}$ elements to be a vector (x, y) where x, y are UINT32. Similarly, we represent $P = (x, y, z, z^2)$ in $E(\mathbb{F}_q)$ (x, y, z are UINT32). Therefore, each block handles eactly one pairing calculation. This grid/block arrangements also simplify the design.

The base field operations include $a+b \mod m$, $a-b \mod m$ and $a \times b \mod m$ where $a, b < m$ and $m < 2^{32}$. For example, to compute $a + b \mod m$, there are two cases: $a + b < m$ and $m \leq a + b < 2m$. In the second case, we have to output $a+b-m$ and therefore we need test whether $a+b < m$ or not. However, this case handling, depending on the input values, causes a branch divergence on GPU. In the full version [11] of this paper, we present methods to minimize the divergence for all the base field operations.

GPU also provides some memory to use. Some of it can be accessed only within a thread; some can be shared among threads of the same block. Some may have special (1D/2D) caches. We have to carefully choose which memory to use to achieve an optimized performance. To allocate memory, 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. For the inter-thread data generated in the

modular reduction algorithm, we use shared memory to store it, as the content in a register file can be only used within one thread. Moreover, 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, we verified that we indeed exploit caching well and the cache-hit rate is very high.

4 Experimental Results

The experiments were conducted on NVIDIA GeForce GTX 285, GTX 480 and Amazon EC2 Cloud [1] (equipped with two Tesla M2050). Specifically, GTX 285, 480 and EC2 have 240, 480 and 448x2 cores separately where each with 1.476GHz, 1.4GHz and 1.15GHz clock. Moreover, GTX 285, 480, EC2 also have 1GB, 1.5GB and 3GBx2 graphical memory on board. Their CUDA versions are 1.3 (GT200), 2.0 (Fermi) and 2.0 (Fermi).

We incorporate Amazon EC2 cloud because it is a popular way to instantiate the outsourced database scenario that requires mass evaluation of composite-order bilinear pairings. We study the real price paid to the EC2 cloud to evaluate each pairing. For comparison, we also choose Pairing-Based Cryptography (PBC) library version 0.5.11 (built upon GMP library[2] version 5.0.1) as the benchmark that runs on Intel Core 2 E8300 CPU at 2.83GHz and 3GB memory. GMP library is designed to be as fast as possible with highly optimized assembly code. Through the experiments, we choose random points $P, Q \in E(\mathbb{F}_q)$ as the input to evaluate $\hat{e}(P, Q)$.

We 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 80bit 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 dashed lines in Fig. 1), where both A and D179 [9] pairing are for 80bit security and A is the fastest. With 2.1 USD charged per hour, 11.9ms on Amazon EC2 also means that the cost to compute a single pairing is as low as $(2.1 \times 11.9)/(60 \times 60 \times 1000) = 7 \times 10^{-6}$ USD.

[1] `http://aws.amazon.com/ec2/`
[2] `http://gmplib.org/`

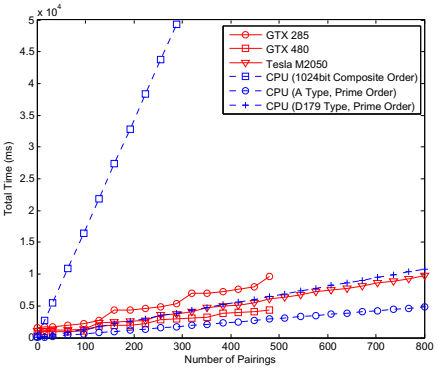

Fig. 1. Running Time on different GPUs and CPU (80bit Security Level)

References

1. Barreto, P.S.L.M., Kim, H.Y., Lynn, B., Scott, M.: Efficient Algorithms for Pairing-Based Cryptosystems. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 354–368. Springer, Heidelberg (2002)
2. Boneh, D., Goh, E., Nissim, K.: Evaluating 2-DNF Formulas on Ciphertexts. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 325–341. Springer, Heidelberg (2005)
3. Freeman, D.: Converting Pairing-Based Cryptosystems from Composite-Order Groups to Prime-Order Groups. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 44–61. Springer, Heidelberg (2010)
4. 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)
5. Katz, J., Sahai, A., Waters, B.: Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 146–162. Springer, Heidelberg (2008)
6. Kawamura, S., Koike, M., Sano, F., Shimbo, A.: Cox-Rower Architecture for Fast Parallel Montgomery Multiplication. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 523–538. Springer, Heidelberg (2000)
7. Lewko, A., Okamoto, T., Sahai, A., Takashima, K., Waters, B.: Fully Secure Functional Encryption: Attribute-Based Encryption and (Hierarchical) Inner Product Encryption. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 62–91. Springer, Heidelberg (2010)
8. Meiklejohn, S., Shacham, H., Freeman, D.: Limitations on Transformations from Composite-Order to Prime-Order Groups: The Case of Round-Optimal Blind Signatures. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 519–538. Springer, Heidelberg (2010)
9. PBC Library. The pairing-based cryptography library, `http://crypto.stanford.edu/pbc/`
10. 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)
11. Zhang, Y., Xue, C.J., Wong, D.S., Mamoulis, N., Yiu, S.: Acceleration of composite order bilinear pairing on graphics hardware. Full Version