# Towards High-performance X25519/448 Key Agreement in General Purpose GPUs

Jiankuo Dong*†‡, Fangyu Zheng*†✉, Juanjuan Cheng*†‡, Jingqiang Lin*†‡, Wuqiong Pan*† and Ziyang Wang*†‡
*State Key Laboratory of Information Security, Institute of Information Engineering,
Chinese Academy of Sciences, Beijing, China
†Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China
‡School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

***Abstract*—Widely used in a large range of Internet security protocols such as TLS/SSL, the key exchange provides a method to establish a shared secret between two parties in unprotected channel. Among the key exchange algorithms, Elliptic Curve Diffie-Hellman is currently preferred and popularized by the industry. The prevailing ECDH employs NIST P Curves as the underlying elliptic curve, however, with the requirement of high performance and questioning of its security, in January, 2016, IRTF officially applied Curve25519/448 to key exchange in RFC 7748, called X25519/448 key exchange protocol. Now many mainstream open-source projects recommend X25519/448 as the default key exchange protocol. The bottleneck of X25519/448 lies in the scalar multiplication, whose performance in traditional CPU cannot serve the rapidly expanding demand in scenes with massive concurrent requests, such as cloud-computing, e-commerce, etc. In this contribution, we accelerate X25519/448 with Graphics Processing Units via various optimizations. The results of X25519/448 in GeForce GTX 1080 respectively reach 2.86 and 0.358 million operations per second, outperforming the previous fastest work by a large margin.**

***Keywords*—Graphics processing unit; Elliptic curve cryptography; Curve25519; Curve448; Diffie-Hellman**

## I. Introduction

Nowadays, as the continuous promotion of computers in various fields, the demands for security are stronger than ever before. At the same time, the applications of Internet finance, offline payment and online transaction, which have strong demands for privacy data protection and transmission security, are still hot topics of current Internet development. Public key cryptography is one of the core cryptography technologies of all kinds of security protocols. Elliptic Curve Cryptography (ECC) [1], a public key cryptography, can provide more security per bit than most of the ones currently in use. Elliptic curve is widely used in applications which need to guarantee the transmission security, privacy protection of users, including key exchange [2], digital signature [3], etc. Because of relatively complex mathematical structure of ECC, existent high performance cryptogram servers and similar equipment are still too slow to satisfy high-throughput tenants. The performance of ECC implementation remains a crucial bottleneck confronting the massive volumes of transactions.

Diffie-Hellman key exchange protocol is a method of establishing shared keys in unprotected channels and the basis for many security protocols. Elliptic Curve Diffie-Hellman (ECDH) key exchange [4, 5] is currently favored by the industry due to its high performance. As reported in [6], there are 72 websites offering an HTTPS connection with certificate in top 100 most visited websites, and of 72 websites, 69 sites use some form of ECC as key exchange protocol. Among all the elliptic curves, the curve family published by the National Institute of Standards and Technology (NIST) are currently the most commonly used one in the world. However, the safety of NIST $P$ curves began to be under suspicion [7] of which the coefficients are generated by hashing unexplained random seeds. Especially, after Edward Snowden exposed the backdoor in Dual_EC_DRBG, suspicious aspects of the NIST's $P$ curve constants led to concerns that the NSA had chosen values that gave them an advantage in factoring public keys [8].

*Curve25519* was introduced by Daniel J. Bernstein in 2006 [9] and has been widely popularized since 2013. In 2016, Internet Research Task Force (IRTF) released Request for Comments (RFC) 7748 [10], recommending two Montgomery curves (Curve25519 and Curve448), respectively for Diffie-Hellman key exchange protocol, X25519 and X448. Curve25519 is not only designed to be safe [7] but also quick, the performance of Curve25519 is significantly better than NIST P-256. Curve25519 has been widely used in a variety of libraries and applications. In 2014, Open Secure Shell (OpenSSH) defaults to Curve25519-based ECDH [11]. Since version 1.1.0 OpenSSL has already supported for X25519 [12]. Although the performance of Curve25519 is superior to $P$-256, the implementation of high-speed and reliable elliptic curve cryptography is still a significant and challenging task. There is an urgent need to employ high-speed computing technologies to improve computational performance to support the development of business in all kinds of fields.

### A. Related Work

Many previous works reported ECC implementations on different high-performance processors/co-processors, such as common CPUs (e.g., OpenSSL [12]), Graphics Processing Units (GPUs), etc. Among them, inspired by the gaming and Artificial Intelligence (AI) industry, GPUs develop rapidly and continuously in particular. And with the advent of CUDA [13] and OpenCL [14], it is now possible for GPUs to bear the workload of general-purpose computation. Antão et al. [15, 16] and Pu et al. [17] employed the Residue Number System (RNS) to parallelize the modular multiplication into several threads. RNS is easy to implement the large integer multiplication in parallel based on GPU platform, because of the independent operation of single word. However, it takes too many instructions to transform between large integer and RNS representations. Bernstein et al. [18] and Leboeuf et al. [19] based on GPU, completed the Montgomery multiplication

978-1-5386-4586-4/18/$31.00 ©2018 IEEE

using a single thread. Montgomery multiplication, a modular multiplication algorithm, is widely implemented in GPU [18, 20, 21, 22, 23]. In ECC implementations, the modulus is small enough, so that one thread can be applied to complete the multiplication within the range of latency acceptance.

For generalized Mersenne prime fields [24], fast reduction algorithm can be applied to the modular operations. The computational cost of modular multiplication algorithm based on the fast reduction algorithm is about half of the Montgomery multiplication. The fast reduction algorithm can be used in most of the prevailing ECC algorithms, such as the ECDSA based on NIST $P$-{192, 224, 256, 384, 521} and SM2 [25]. Pan et al. [26], Zheng et al. [27], Bos et al. [28] and Szerwinski et al. [29] used fast reduction to implement modular multiplication over the generalized Mersenne prime fields [24]. In 2017, Pan et al. [26] based on GTX 780Ti, achieved $920,000$ NIST P-256 signature verification operations, which is known as the best performance of scalar multiplication.

A number of works also reported Curve25519 implementations in different high-performance processors/co-processors. Michael et al. [30] implemented the X25519 key-exchange protocol for AVR ATmega 8-bit, MSP430X 16-bit , and ARM Cortex-M0 32-bit micro-controllers. Philipp et al. [31] reported a latency of 92$\mu$s for X25519 on FPGA. Armando et al. [32] achieved ECDH protocol based on Curve25519 using AVX2 instruction set. Mahe et al. [33] implemented Curve25519 based on OpenCL on AMD R9 290X, GTX TITAN, etc.

### B. Contribution and Paper Organization

In this work, we propose a comprehensive approach based on General Purpose GPU (GPGPU) to implement high-performance, low-latency and high-flexibility X25519/448 functions, which are used to implement Diffie-Hellman key agreement. The contributions of this paper are threefold.

1) First, we propose an efficient and concise reduction arithmetic for modular addition (subtraction) and multiplication (square) of Curve25519/448, by carefully scheduling and optimizing the algorithm at assembly level using CUDA parallel thread execution (PTX) instruction set architecture (ISA) instructions.
2) Second, for the best practice of the X25519/448 functions, we adjust the order of steps for scalar multiplication reasonably with reducing arithmetic operations and reusing the variables as far as possible to take full advantage of the limited registers.
3) Finally, from the experimental exploration, we find the optimal performance parameters. And compared with previous works, our throughput is 266% performance of the X25519 implementation [33] using the same GPU GTX TITAN. Also, our work is 171% performance of the existing fastest NIST P-256 implementation [26] based on uniform GTX 780Ti. In particular, on GeForce GTX 1080, the results of X25519/448 functions reach the throughput of 2,860,412/357,944 operations per second (ops/s).

The rest of our paper is organized as follows. Section II presents the overview of Diffie-Hellman protocol, X25519/448, GPU and CUDA. Section III describes implementations from finite field arithmetic, especially fast reduction algorithm, and curve arithmetic operations. Section IV analyses the performance of proposed algorithm and compares it to previous works. Section V concludes the paper.

## II. Preliminaries

This section firstly details the Diffie-Hellman Key Exchange and ECDH. Then we introduce the basics of Curve25519/448, and finally provide the brief introduction to GPUs and our target platform GTX 1080.

### A. Diffie-Hellman Key Exchange and ECDH

Diffie-Hellman key exchange is a method of exchanging cryptographic keys securely over a public channel and was one of the first public-key protocols as originally conceptualized by Ralph Merkle [2] and named after Whitfield Diffie and Martin Hellman [34]. It is one of the earliest practical examples of public key exchange implemented within the field of cryptography and is used to provide forward secrecy in Transport Layer Security's ephemeral modes.

ECDH is a variant of the Diffie-Hellman protocol using ECC. The difference between DH and ECDH is mainly the group which is being chosen to compute the secret key(s). While DH uses a multiplicative group of integers modulo a prime $p$, ECDH uses a multiplicative group of points on an elliptic curve. Compared with the traditional DH algorithm based on finite field, ECDH key exchange is currently favored by the industry due to its performance advantages and is gradually promoted. The ECDH process is shown as following steps:

1) Alice and Bob agree on an elliptic curve $E$ over a Field $\mathbb{F}_p$ and a base point $P \in E/\mathbb{F}_p$.
2) Alice generates a (random) secret $k_A$ and computes $P_A = k_A P$.
3) Bob generates a (random) secret $k_B$ and computes $P_B = k_B P$.
4) Alice and Bob exchange $P_A$ and $P_B$.
5) Alice and Bob compute $P_{AB} = k_A P_B = k_B P_A$.

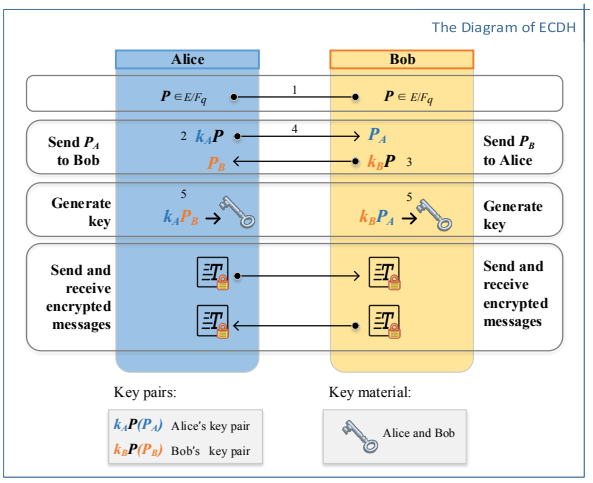


Fig. 1. ECDH Diagram

The secret $k_A$ and $k_B$ is a random value $\in \{1, ..., n-1\}$ where $n$ is the order of the group generated by $P$.

### B. Curve25519 and Curve448

In general, elliptic curves are defined by the Weierstrass form:

$$y^2 = x^3 + ax + b. \tag{1}$$

Montgomery curve is a form of the elliptic curve, different from the usual Weierstrass form, introduced by Peter L. Montgomery in 1987. A Montgomery curve is defined by the equation

$$By^2 = x^3 + Ax^2 + x. \tag{2}$$

Curve25519 is a Montgomery curve originally proposed by Bernstein in 2006 [9]. Curve25519 over a 255-bit prime field ($p = 2^{255} - 19$) provides 128 bits of security. As one of the fastest implementations Curve25519 has been applied on many platforms. By carefully designed, it is immune to timing attacks and accepts any 32-byte string as a valid public key and does not require validation. Curve448 [35] is also constructed on the Montgomery curve, defined over the Goldilocks prime $\mathbb{F}_p$, where $p = 2^{448} - 2^{224} - 1$. Curve448 provides 224 bits of security and accepts any 56-byte string as a valid key.

$$Curve25519: \ y^2 = x^3 + 486662x^2 + x \tag{3}$$

$$Curve448: \ y^2 = x^3 + 156326x^2 + x \tag{4}$$

The X25519 and X448 functions which are used to implement Diffie-Hellman, perform scalar multiplication on the Montgomery form of the above curves. As shown in RFC 7748 [10], the functions are efficient $x$-coordinate-only algorithm, which compute the $x$-coordinate of $kP$ based only the $x$-coordinate of $P$.

### C. GPU

Specialized for the compute-intensive and high-parallel computations required by graphics rendering, GPU has been proved as a viable choice for high parallelism computations. GPU provides much greater computing capability than CPU by devoting more transistors to arithmetic processing unit rather than data caching and flow control [13]. From 2010 to the present, the computing power of GPUs has grown nearly tenfold, from Fermi architecture to Volta architecture [36], but there is no significant development for CPUs. In November 2006, the Compute Unified Device Architecture (CUDA) is proposed by NVIDIA [13], so that it is possible to perform general-purpose computation on GPUs. Many researchers resort to GPUs to perform various cryptographic acceleration based GPU, including RSA [37, 22], ECC [26].

Our target platform, GeForce GTX 1080 (codename GP-104) based Pascal architecture [38] is released by NVIDIA in 2016. There are 2560 CUDA cores (Coumpute Capability 6.1) distributed in 20 streaming multiprocessors (SM). 32 threads (grouped as a warp) within one SM concurrently run in a clock [13]. Following the Single Instruction, Multiple Threads (SIMT) architecture, each GPU thread runs one instance of the kernel function. Multiple warps of threads assigned to one SM for better utilization of the pipeline of each SM are called a block. Maximum number of threads per block is 1024. Each SM possesses 64K 32-bit registers, and there are 8GB 256-bit width global memory for all SMs.

The NVIDIA CUDA programming environment provides a parallel thread execution instruction set architecture for using the GPU as a data-parallel computing device [39]. And PTX offers a stable programming model and instruction set for general purpose parallel programming, which spans multiple GPU generations. Assembler statements, *asm()*, provide a way, called inline PTX Assembly in GPU [40], to insert arbitrary PTX code into the CUDA program. In order to improve the efficiency of instruction execution, this work consists of CUDA C programming with inline PTX assembly language, in particular, the modular addition, modular subtraction and modular multiplication are implemented by PTX ISA instructions. The instructions used in our work are explained in Table I.

TABLE I. PTX ISA INSTRUCTION EXPLANATION

| | Instructions [*] | Meanings |
|---|---|---|
| Addition | $s = add(a, b)$ | $s = a + b$ |
| | $s = addc(a, b)$ | $s = a + b + CF$ |
| Subtraction | $s = sub(a, b)$ | $s = a - b$ |
| | $s = subc(a, b)$ | $s = a - b - CF$ |
| Multiplication | $s = mul.lo(a, b)$ | $s = (a \times b)_{lo}$ |
| | $s = mul.hi(a, b)$ | $s = (a \times b)_{hi}$ |
| Multiply-Add | $s = mad.lo(a, b, c)$ | $s = (a \times b)_{lo} + c$ |
| | $s = mad.hi(a, b, c)$ | $s = (a \times b)_{hi} + c$ |
| | $s = madc.lo(a, b, c)$ | $s = (a \times b)_{lo} + c + CF$ |
| | $s = madc.hi(a, b, c)$ | $s = (a \times b)_{hi} + c + CF$ |

[*] If *.cc* specified, carry-out written to CC.CF.

## III. METHODOLOGY

This section introduces a bottom-up overall methodology of X25519/448, including optimization on finite field arithmetic and curve arithmetic.

### A. Large Integer Algorithm

*1) Addition and Subtraction:* The calculations are based on 32-bit integers in our implementation. The optimized large integer addition and subtraction are operated with PTX ISA on GPUs. As shown in Figure 2, we take the (5-integer+5-integer) as an example to introduce our addition diagram of Curve25519/448. For Curve25519, the addition is implemented with only eight $s = addc(a, b)$ instructions, as for Curve448, is 14. Note that the result of the addition is a large integer $C$ and $carry$ which is no more than 1. The method of carry resolution will be described particularly in Section III-B. The subtraction operation for Curve25519/448 is similar to the addition algorithm but using PTX ISA integer subtraction instruction $s = subc(a, b)$.

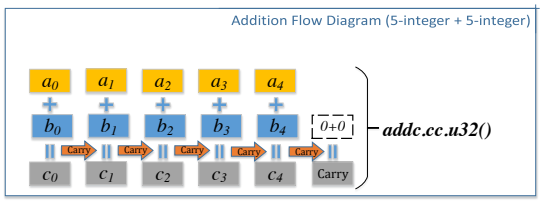


Fig. 2. Large Integer Addition Algorithm

*2) Multiplication and Multiply-Add:* The large integer multiplication is on the basis of $mad.lo()$ and $mad.hi()$ instructions. Our implementations of multiplication are similar to [27]. The primary units of $C = A \times B$ are $c_{i+j} = madc.lo.u32(a_i, b_j, c_{i+j})$ and $c_{i+j+1} = madc.hi.u32(a_i, b_j, c_{i+j+1})$. We also accomplish a large integer multiply-add $C = A \times B + D$. Compared with traditional multiplication, the integrated multiply-add can provide an additional adder, but of which the instruction number remains unchanged. As shown in Figure 3, we also present a 5-integer diagram as an example of multiply-add. And the Figure 3 can be also regarded as multiplication algorithm in case of replacing the $(d_0, d_1..., d_4)$ (the yellow parts) by zero.

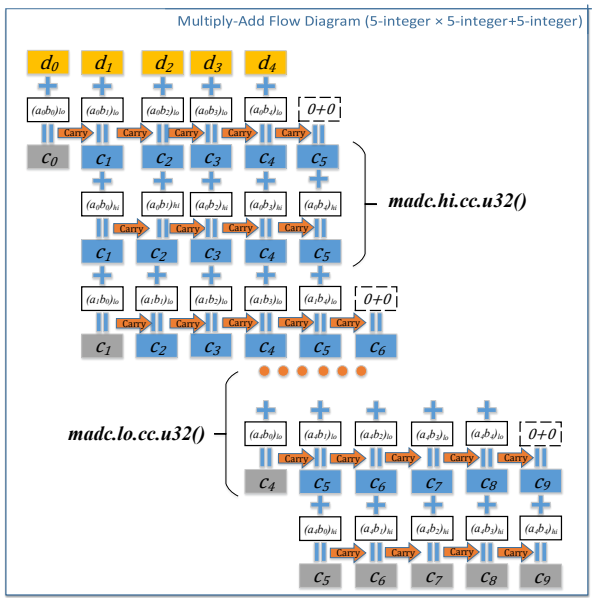


Fig. 3. Large Integer Multiply-Add Algorithm

*3) Square:* The square operation can be implemented with less instructions than the ordinary multiplication. To reduce the number of multiply-add operations, the process of large integer square is elaborately designed and implemented. In general, the large integer square can be divided into three steps:

1) Calculate $a_i \times a_j$ where $i < j$ just like the ordinary multiplication.
2) Double the results of the calculation, so that results of $a_i \times a_j$ where $i \neq j$ are obtained, which can be done with the simple addition instructions.
3) Add results of $a_i \times a_i$ to the original calculation results, making full use of $madc.cc.u32()$ instructions and no demand for addition instructions.

## B. Fast Reduction

*1) Curve25519 Fast Reduction:* For Curve25519, the finite field is $\mathbb{F}_p$, where $p = 2^{255} - 19$. But the bit-length of the target GPU is 32, thus we present the 255-bit elements ($A$ and $B$) with eight 32-bit integers, which now range in $[0, 2^{8\times32})$. Therefore, after $C = A \times B$, the result $C$ is in the range $[0, 2^{512})$, where

$$C \equiv \sum_{i=0}^{15} c_i 2^{32i} \equiv \sum_{i=8}^{15} c_i 2^{32i} + \sum_{i=0}^{7} c_i 2^{32i} (mod \quad p). \tag{5}$$

Note that $p = 2^{255} - 19$, which implies that

$$2^{255} \equiv 19 \ mod \ p, \tag{6}$$

and

$$2^{256} \equiv 38 \ mod \ p, \tag{7}$$

thus Equation (5) can be deduced as:

$$\begin{aligned} C \equiv & \sum_{i=0}^{7} 38 c_{i+8} 2^{32i} + \sum_{i=0}^{7} c_i 2^{32i} (mod \ p) \\ \equiv & \sum_{i=0}^{7} (c_i + 38 c_{i+8}) 2^{32i} (mod \ p). \end{aligned} \tag{8}$$

**Algorithm 1** Modular Reduction from 512 Bits to 256 Bits (with carry) for Curve25519

**Input:**
The 512-bit (16-integer) large number $C = \sum_{i=0}^{15} 2^{32i} c_i$,where $c_i \in [0, 2^{32})$;
**Output:**
The 256-bit (8-integer) large number $C = \sum_{i=0}^{7} 2^{32i} c_i$, where $c_i \in [0, 2^{32})$ and $carry$;

1: $carry = 0$
2: **for** $i = 0$ *to* $3$ **do**
3: $\quad c_{2i} = madc.lo.cc.u32(c_{2i+8}, 38, c_{2i})$
4: $\quad c_{2i+1} = madc.hi.cc.u32(c_{2i+8}, 38, c_{2i+1})$
5: **end for**
6: $carry = addc.u32(0, 0)$
7: **for** $i = 0$ *to* $2$ **do**
8: $\quad c_{2i+1} = madc.lo.cc.u32(c_{2i+9}, 38, c_{2i+1})$
9: $\quad c_{2i+2} = madc.hi.cc.u32(c_{2i+9}, 38, c_{2i+2})$
10: **end for**
11: $c_7 = madc.lo.cc.u32(c_{15}, 38, c_7)$
12: $carry = madc.hi.cc.u32(c_{15}, 38, carry)$

As demonstrated in Equation (8), the 512-bit product is composed of 16 integers. Adding $38 \times (c_8, ...c_{15})$ to $(c_0, ...c_7)$ can reduce the results from 512 bits to 256 bits. After the lager integer addition, there is also a carry left which is less than 6 bits. As shown in Algorithm 1, we just employ 16 MAD instructions to reduce the 512-bit multiplication result to 256-bit $C$ and $carry$. This carry, similar to the carry of addition result in Section III-A1, can be disposed as followed methods. According to Equation (7), to eliminate the addition carry, we should add $38 \times carry$ to the result $C$, but it may produce another carry. To deal with the carry, the basic method is to repeat the addition calculations until the carry is zero with the loop instruction `while()`. We find this carry-reduction method has some obvious defects. Diversified carry of threads may lead diverse execution logic in a warp. These different instructions in a warp would cause warp divergence, greatly reducing the overall performance. And because of the differences of run duration, the adverse factor of the implementation may be struck by timing attacks.

To resolve the carry in a constant time, we first come up with a universal solution that can be applied to both Curve25519 and Curve448. The range of Curve25519 multiplication reduction result after Algorithm 1 is

$$(carry_0|C_0) \in [0, 38 \times (2^{256}-1) + (2^{256}-1)]. \quad (9)$$

The $carry_0$ after Algorithm 1 is less than 38 in the result $(carry_0|C)$. The addition of $carry \times 38$ and 256-bit $C$ is represented as

$$(carry|C) + carry \times 38 - carry \times 2^{256} \Rightarrow (carry'|C'). \quad (10)$$

And it is also equivalent to

$$C + carry \times 38 \Rightarrow (carry'|C'). \quad (11)$$

We name the Equation (11) as *25519-carry-addition* for Curve25519 carry resolution method. After the first round 25519-carry-addition, the range of $(carry_1|C_1)$ is

$$\begin{cases} [0, 2^{256}-1] & \text{if } carry_0 = 0 \\ [38, 2^{256}+1405] & \text{if } carry_0 > 0 \end{cases} \quad (12)$$

From Equation (12), the result $(carry_1|C_1) \in [0, 2^{256}+1045]$, note that the $carry_1$ belongs to $\{0, 1\}$. Then we perform the second round of 25519-carry-addition, and $(carry_2|C_2)$ belongs to

$$\begin{cases} [0, 2^{256}-1] & \text{if}(carry_1|C_1) \in [0, 2^{256}-1] \\ [38, 1443] & \text{if}(carry_1|C_1) \in [2^{256}, 2^{256}+1405] \end{cases} \quad (13)$$

Note that $(carry_2|C_2) \in [0, 2^{256}-1]$, which means the $carry_2$ is zero. we completely eliminate carry using two rounds additions of $38 \times carry$ and 256-bit $C$. From above process of resolution, regardless of if the carry is zero, performing the addition with $38 \times carry$ twice can handle all multiplication reduction results after Algorithm 1. This two-addition-round method without `while()` loop instruction for handling carry of Curve25519 can be extended to Curve448, which is also timing attack proof.

But we find a more efficient way, which requires only one-addition-round carry resolution. We can use Equation (6) directly instead of Equation (7). Note that the most significant bit (the 256th bit, $C_{256}$) of the result $C$ can also be treated as another bit of carry. The least 255-bit of $C$ can be regarded as $\widehat{C}$. Then we can add $(carry_0|C_{256}) \times 19$ to the $\widehat{C}$ only once. Note that $carry_0$ is no more than 38, the $(carry_0|C_{256}) \in [0, 38 \times 2 + 1]$, after the addition

$$(carry_0|C_{256}) \times 19 + \widehat{C} \Rightarrow (carry_1|C_1), \quad (14)$$

and the range of $(carry_1|C_1)$ is

$$(carry_1|C_1) \in [0, 2^{255} + 1462]. \quad (15)$$

The range of result is less than $2^{256}$, which means no more carry resolution is required. This procedure can improve the performance of Curve25519 about 3.5% compared with two-addition-round method, but cannot be applied to Curve448. The one-addition-round carry resolution is shown in Algorithm 2.

The overall reduction process of Curve25519 is shown in Figure 4. For Curve25519 modular multiplication (square), the reduction method is processed as *Step(1-3)*. Algorithm 1 which reduce the 512-bit multiplication result to 256-bit, is displayed as *Step(1)* of Figure 4. *Step(2-3)* are processed to resolve the carry after Algorithm 1. As for Curve25519 modular addition (subtraction), the reduction approach is just to resolve the carry which is begun from *Step(2)* to *Step(3)* in Figure 4.

**Algorithm 2** Constant-time Carry Resolution for Curve25519 with One-addition-round correction

**Input:**
The 256-bit (8-integer) large number $C = \sum_{i=0}^{7} 2^{32i} c_i$, where $c_i \in [0, 2^{32})$ and $carry \in [0, 38]$;

**Output:**
The 256-bit (8-integer) large number $C = \sum_{i=0}^{7} 2^{32i} c_i$, where $c_i \in [0, 2^{32})$;

1: $carry = carry << 1$
2: $carry += c_7 >> 31$
3: $c_7 \& = (1 << 31) - 1$
4: $c_0 = mad.lo.cc.u32(carry, 19, c_0)$
5: **for** $i = 1$ *to* $6$ **do**
6: $\quad c_i = addc.cc.u32(c_i, 0)$
7: **end for**
8: $c_7 = addc.u32(c_7, 0)$

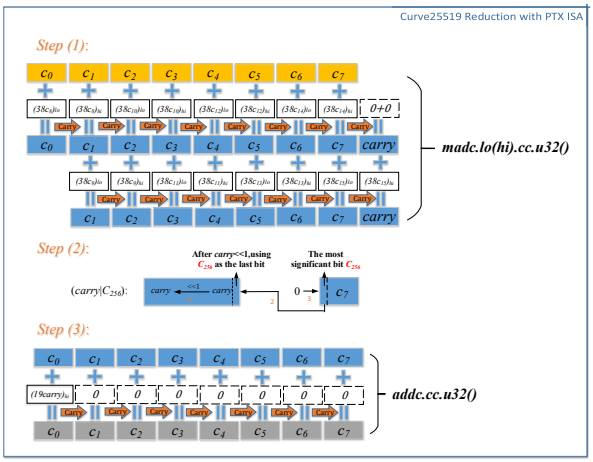


Fig. 4. Curve25519 Reduction with PTX ISA

*2) Curve448 Fast Reduction:* Compared with Curve25519, Curve448 fast reduction is more complex. The result $C$, after the multiplication $C = A \times B$, can be represented as the Equation (16). The result has a total of 28 32-bit integers (896-bit length) which has to be reduced to 14-integer length (448-bit length).

$$C \equiv \sum_{i=0}^{27} c_i 2^{32i} \equiv \sum_{i=14}^{27} c_i 2^{32i} + \sum_{i=0}^{13} c_i 2^{32i} \ (mod\ p) \quad (16)$$

Note that $p = 2^{448} - 2^{224} - 1$, which implies

$$2^{448} \equiv 2^{224} + 1 \ (mod\ p). \quad (17)$$

Following Equation (17), Equation (16) can be transformed

into

$$C \equiv \sum_{i=7}^{13}(c_i + c_{i+7} + 2c_{i+14})2^{32i} + \sum_{i=0}^{6}(c_i + c_{i+14} + c_{i+21})2^{32i} \ (mod\ p). \quad (18)$$

**Algorithm 3** Modular Reduction from 896 Bits to 448 Bits (with carry) for Curve448

**Input:**
The 896-bit (28-integer) large number $C = \sum_{i=0}^{27} 2^{32i}c_i$, where $c_i \in [0, 2^{32})$;
**Output:**
The 448-bit (14-integer) large number $C = \sum_{i=0}^{13} 2^{32i}c_i$, where $c_i \in [0, 2^{32})$ and $carry$;

$carry = 0$
**for** $i = 0$ *to* $6$ **do**
  $c_i = addc.cc.u32(c_i, c_{i+14})$
**end for**
**for** $i = 7$ *to* $13$ **do**
  $c_i = addc.cc.u32(c_i, c_{i+7})$
**end for**
$carry = addc.u32(0, 0)$
**for** $i = 0$ *to* $6$ **do**
  $c_i = addc.cc.u32(c_i, c_{i+21})$
**end for**
**for** $i = 7$ *to* $13$ **do**
  $c_i = addc.cc.u32(c_i, c_{i+14})$
**end for**
$carry = addc.u32(0, carry)$
**for** $i = 7$ *to* $13$ **do**
  $c_i = addc.cc.u32(c_i, c_{i+14})$
**end for**
$carry = addc.u32(0, carry)$

The proposed reduction method of Curve448 is also based on $addc.cc.u32()$ instruction. Algorithm 3 is generated to reduce the 896-bit result $C$ to 448-bit length and also a $carry$ left, and the significant length of carry is no more than 2. For Curve448, the result $carry$ in Algorithm 3 has consistent resolution method with large integer addition (subtraction) result in Section III-A1. Note that with Equation (17), the $carry$ can be resolved as Equation (19):

$$C \equiv \sum_{i=7}^{13} c_i 2^{32i} + carry \times 2^{32\times 7} + \sum_{i=0}^{6} c_i 2^{32i} + carry\ (mod\ p), \quad (19)$$

which is named as 448-carry-addition in our implementation. But there may be also another carry after the Equation (19). Just like the first resolution method for Curve25519, we can also settle the carry with two rounds of 448-carry-addition no matter if the carry is zero, The process of proof is similar to Curve25519 with Equation (9-13). For Curve448, the carry, which is produced by Algorithm 3 and large integer addition (subtraction), can be disposed of as Algorithm 4.

The integrated Curve448 reduction method is shown in Figure 5. We carefully design the well-structured multiplication

**Algorithm 4** Constant-time Carry Resolution for Curve448

**Input:**
The 14-integer (448-bit) large number $C = \sum_{i=0}^{13} 2^{32i}c_i$, where $c_i \in [0, 2^{32})$ and $carry$;
**Output:**
The 14-integer (448-bit) large number $C = \sum_{i=0}^{13} 2^{32i}c_i$, where $c_i \in [0, 2^{32})$;

$carry = 0$
**448-carry-addition:**
$c_0 = add.cc.u32(c_0, carry)$
**for** $i = 1$ *to* $6$ **do**
  $c_i = addc.cc.u32(c_i, 0)$
**end for**
$c_7 = add.cc.u32(c_7, carry)$
**for** $i = 8$ *to* $13$ **do**
  $c_i = addc.cc.u32(c_i, 0)$
**end for**
$carry = addc.u32(carry, 0)$
**Repeat the 448-carry-addition once**

(square) reduction method which is completely from *Step(1)* to *Step(2)*. And *Step(1)* is implemented in strict accordance with Algorithm 3, which serves to reduce the 996-bit $C$ to 448-bit length, and also a carry may be left. The *Step(2)* is processed to resolve the carry which is in the light of Algorithm 4. In addition to resolving the carry in Algorithm 3, *Step(2)* in Figure 5 can also be used to deal with the carry in Curve448 large integer addition (subtraction). In conclusion, our overall Curve448 reduction method is elaborately designed to be timing-attack proof.

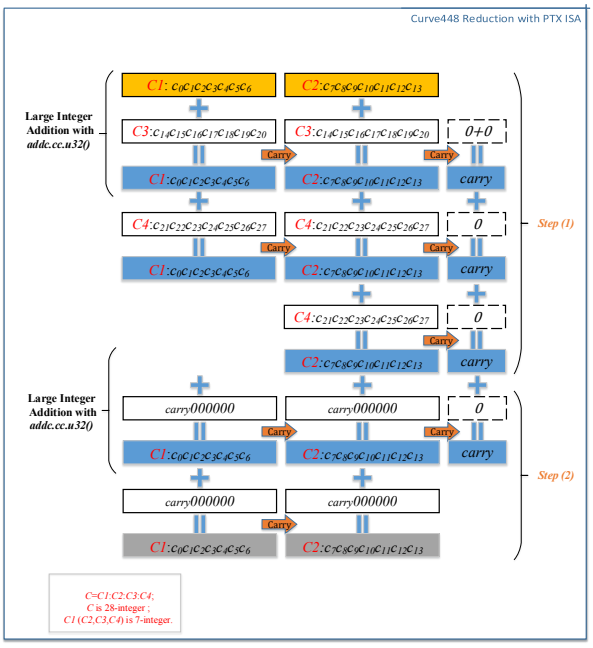


Fig. 5. Curve448 Reduction with PTX ISA

## C. Curve Arithmetic

After accomplishing the efficient implementation of finite field arithmetic, we also focus on the improvements of the

curve arithmetic level. The X25519 and X448 functions using Montgomery ladder is shown in Algorithm 5. Because of the $x$-coordinate-only algorithm, the inputs and outputs are 32 integers for X25519 and 56 integers for X448. There are two noteworthy factors which can affect the performance of X25519 and X448 functions: one is the number of registers used by the algorithms which is the limited resource of GPUs, and the other is the number of finite field arithmetic operations during the implementations.

Without affecting the correctness of the results, we carefully schedule the algorithm of scalar multiplication reasonably and reuse the variables as much as possible to take full advantage of the register resource. Our implementation uses only 7 sets of field variables which are half of the raw RFC 7748 implementation [10]. Furthermore, in *Step 22* of Algorithm 5, a supernumerary modular multiply-add arithmetic with `madc.lo(high)` instructions is implemented to replace a separate modular multiplication and an addition in $\mathbb{F}_p$. The constant $a24$ is $(486662-2)/4 = 121665$ for Curve25519 and $(156326-2)/2 = 39081$ for Curve448. There is a conditional swapping (`cswap()`) in the implementation of Montgomery point multiplication. This swapping is judged by the scalar bits which are the key of scalar multiplication shown in Algorithm 5.

**Algorithm 5** The X25519 and X448 Functions using Montgomery Ladder

**Input:**
A $n$-bit scalar $k$ and the $x$-coordinate $u$ of some point $P$
**Output:**
The $x$-coordinate of $kP$

1: $x_1 = u;\ x_2 = 1;\ x_3 = u;$
2: $z_2 = 0;\ z_3 = 1;\ swap = 1$
3: **for** $i = n-1\ to\ 0$ **do**
4: $k\_t = (k >> t)\ \&\ 1$
5: $swap = swap \oplus k\_t$
6: $(x_2, x_3)$=cswap$(swap, x_2, x_3)$
7: $(z_2, z_3)$=cswap$(swap, z_2, z_3)$
8: $swap = k\_t$
9: $tmp0 = x_3 - z_3$
10: $tmp1 = x_2 - z_2$
11: $x_2 = x_2 - z_2$
12: $z_2 = x_3 + z_3$
13: $z_3 = tmp0 \times x_2$
14: $z_2 = z_2 \times tmp1$
15: $tmp0 = tmp1^2$
16: $tmp0 = x_2^2$
17: $x_3 = z_3 + z_2$
18: $z_2 = z_3 - z_2$
19: $x_2 = tmp1 \times tmp0$
20: $tmp1 = tmp1 - tmp0$
21: $z_2 = z_2^2$
22: $tmp0 = tmp1 \times (a24 + 1) + tmp0$
23: $x_3 = x_3^2$
24: $z_3 = x_1 \times z_2$
25: $z_2 = tmp1 \times tmp0$
26: **end for**
27: $(x_2, x_3)$=cswap$(swap, x_2, x_3)$
28: $(z_2, z_3)$=cswap$(swap, z_2, z_3)$
29: **return** $x_2 \times (z_2^{p-2})$

## IV. Performance Evaluation and Related Work Comparison

This section presents the implementation performance and summarizes the results for the proposed algorithm. Relative assessment is also presented by considering related implementations. The hardware and software configurations used in the experiment are listed in Table II.

TABLE II. Target Platform Configuration

| | |
|---|---|
| CPU | Intel Xeon CPU E5-2620 v2 at 2.10GHz |
| GPU | GeForce GTX 1080 |
| OS | Ubuntu 16.04 |
| Tool Chain | CUDA 8.0 |

### A. Performance Evaluation

As the process presented in Section III, the CUDA kernels of X25519/X448 functions are accomplished for scalar multiplication of the base point and scalar multiplication of a unknown point multiplication required for the Diffie-Hellman key exchange protocol. For each scalar multiplication, we use one thread to implement the entire X25519/X448 function avoiding the data exchange between threads. There are some configuration parameters which may affect the performance of the kernel, including the following:

- *Batch Size*: the number of X25519/X448 scalar multiplication per GPU kernel launches;
- *Threads/Block*: the number of CUDA threads contained in a CUDA block;
- *Regs/Thread*: the maximum number of registers assigned for each CUDA thread.

*Batch Size* is an important parameter which affects the performance of GPU kernels, the size of which is the product of the number of blocks and *Threads/Block*. Figure 6 summarizes the impact of batch size on performance, which indicates the larger batch size always leads to higher throughput within a certain range. Sufficient threads can keep the pipeline busy most of time and fully utilize the computing power in GPUs. But with the limitation of the resource, excessive threads per block also can largely decrease the overall throughput. Note that the configuration *Reg/Thread* is restricted within the GPU hardware limitation, and *Threads/block* $\times$ *Regs/thread* = 65536 32-bit registers per CUDA block in GTX 1080. When overburdening the threads, the number of registers is so insufficient that many variables are spilled into off-chip local memory which is hundreds times slower than registers, resulting in a significant decline in performance.

TABLE III. Performance of Scalar Multiplication on GTX 1080

| Curve | Batch Size | Reg. | Throughput (ops/s) | Latency (ms) |
|---|---|---|---|---|
| 25519 | 20×896 | 72 | 2,860,412 | 6.26 |
| 448 | 20×512 | 128 | 357,944 | 28.61 |

We set new speed records for ECDH key-exchange of 128-bit and 224-bit security strength based on GeForce GTX 1080.

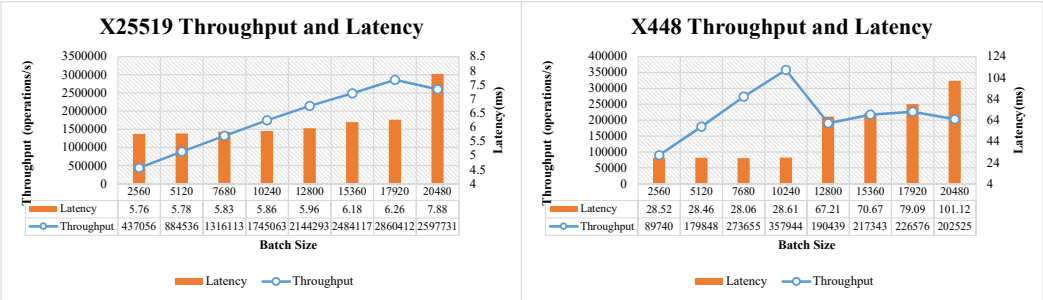


Fig. 6. The Impact of *Batch Size* on Performance of Scalar Multiplication

TABLE IV. PERFORMANCE COMPARISON WITH RELATED WORKS

| | Philipp et al. [31] | Armando et al. [32] | Pan et al. [26] | Mahe et al. [33] | ours | | |
|---|---|---|---|---|---|---|---|
| Platform | FPGA Zynq-7030@115MHz | CPU Core i7-4770@3.5GHz | GPU GTX 780Ti | GPU GTX TITAN | GPU GTX TITAN | GPU GTX 780Ti | GPU GTX 1080 |
| ECC Curve | Curve25519 | Curve25519 | NIST *P*-256 | Curve25519 | Curve25519 | Curve25519 | Curve25519 |
| Throughput (ops/s) | 10,869 | 89,456 | 929,000 | 524,288 | 1,394,031 | 1,588,809 | 2,860,412 |

### B. Related Work Comparison

The implementations of the recent research efforts are summarized in Table IV. For a fair comparison of performance, the proposed X25519 implementation is also tested on GPU GTX TITAN and GTX 780Ti.

*1) vs. CPU and FPGA:* Armando et al. [32] based AVX2 instruction set on Intel Haswell processor used $156.5 \times 10^3$ clock cycles to finish a Curve25519 scalar multiplication, the theoretically peak throughput on Core i7-4770@3.5GHz with four cores is 89.5k ops/s. Philipp et al. [31] reported a latency of 92$\mu$s for X25519, namely, 10.8k ops/s. And our peak throughput of X25519 based on GTX1080 is 2860.4k ops/s, which outperforms their works by 1-2 orders of magnitude. Compared with CPU and FPGA works, GPU platforms with massively parallel processors are more adept at throughput-oriented X25519 implementations, but weak in computation latency and power consumption.

*2) vs. GPU:* Mahe et al. [33] reported that it took 2.0 seconds to finish 1,048,576 Curve25519 scalar multiplications on GPU GTX Titan, i.e., 524,288 ops/s. Compared with this work, our proposed implementation shows a great advantage on the same GPU GTX TITAN, 256% performance of the work [33]. Compared with our implementations using CUDA, the work [33] was based on OpenCL, which cannot fully utilize the low-level computing resource of GPUs. Meanwhile, we optimize the X25519/448 algorithms based on a low-level PTX instruction set architecture to speed up the computations.

Pan et al. [26] reported the existing highest throughput of NIST P-256, i.e., 929,000 ops/s on GTX780Ti; while our Curve25519 scalar multiplication achieves 1,588,809 ops/s based on the same GTX 780Ti, 171% performance of their work. Our performance advantage lies mainly in the more efficient Curve25519 and our optimization methods to the implementations.

## V. CONCLUSION

In this contribution, we exploit the potential of the PTX ISA instructions in general purpose GPUs, where the implementations are relied on two optimizations: one is the state-of-the-art implementations of reduction for *Curve25519/448*; and the other is sheduling the steps of X25519/448 functions. Furthermore, we explore the optimal parameters of the experiments on Geforce GTX 1080, and our throughput-oriented results for X25519/448 are 2.86/0.358 million operations per second. Our optimization methods in GPGPUs can also be extended to other architectures.

Our future work will focus on multi-thread implementations of low-latency DH key agreement, and high-performance secure implementations of signature schemes, such as Edwards-curve Digital Signature Algorithm (EdDSA) [41], etc.

## ACKNOWLEDGMENT

This work was partially supported by National Key R&D Program of China under Award No. 2017YFC0822704 and National Natural Science Foundation of China under Award No. 61772518. Fangyu Zheng is the corresponding author, *E-mail: fyzheng@is.ac.cn*.

## REFERENCES

[1] M. Lochter and J. Merkle, "Elliptic curve cryptography (ecc) brainpool standard curves and curve generation," Tech. Rep., 2010.

[2] R. C. Merkle, "Secure communications over insecure channels," *Communications of the ACM*, vol. 21, no. 4, pp. 294–299, 1978.

[3] U. D. of Commerce, N. I. of Standards, and Technology, "Digital signature standard (DSS)," http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf, 2013.

[4] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in CryptologyCRYPTO85 Proceedings*. Springer, 1986, pp. 417–426.

[5] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[6] R. Harkanson and Y. Kim, "Applications of elliptic curve cryptography: a light introduction to elliptic curves and a survey of their applications," in *Proceedings of the 12th*

*Annual Conference on Cyber and Information Security Research*. ACM, 2017, p. 6.

[7] D. J. Bernstein and T. Lange, "Safecurves: choosing safe curves for elliptic-curve cryptography," *URL: http://safecurves. cr. yp. to*, 2013.

[8] ——, "Failures in nists ecc standards," 2015.

[9] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *International Workshop on Public Key Cryptography*. Springer, 2006, pp. 207–228.

[10] A. Langley and M. Hamburg, "Elliptic curves for security," *order*, vol. 500, p. 39081, 2016.

[11] "Openssh," http://www.openssh.com/index.html, 2017.

[12] O. S. Foundation, "OpenSSL Cryptography and SSL/TLS Toolkit," http://www.openssl.org/, 2016.

[13] NVIDIA, "CUDA C programming guide 9.0," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2017.

[14] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[15] S. Antão, J.-C. Bajard, and L. Sousa, "Elliptic curve point multiplication on GPUs," in *IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*. IEEE, 2010, pp. 192–199.

[16] ——, "RNS-Based elliptic curve point multiplication for massive parallel architectures," *The Computer Journal*, vol. 55, no. 5, pp. 629–647, 2012.

[17] S. Pu and J.-C. Liu, "EAGL: An elliptic curve arithmetic GPU-based library for bilinear pairing," in *Pairing-Based Cryptography–Pairing 2013*. Springer, 2014, pp. 1–19.

[18] D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang, "The billion-mulmod-per-second PC," in *Workshop record of SHARCS*, vol. 9, 2009, pp. 131–144.

[19] K. Leboeuf, R. Muscedere, and M. Ahmadi, "A GPU implementation of the Montgomery multiplication algorithm for elliptic curve cryptography," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 2593–2596.

[20] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang, "ECM on graphics cards," in *Advances in Cryptology-EUROCRYPT 2009*. Springer, 2009, pp. 483–501.

[21] O. Harrison and J. Waldron, "Efficient acceleration of asymmetric cryptography on graphics hardware," in *Progress in Cryptology–AFRICACRYPT 2009*. Springer, 2009, pp. 350–367.

[22] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "Sslshader: cheap ssl acceleration with commodity processors," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. USENIX Association, 2011, pp. 1–1.

[23] B. D. Jeffrey A. Robinson, "Fast GPU based modular multiplication," http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4156_montgomery_multiplication_CUDA_concurrent.pdf, 2014.

[24] J. A. Solinas, *Generalized mersenne numbers*. Citeseer, 1999.

[25] Chinese Commercial Cryptography Administration Office, "Public key cryptographic algorithm SM2 based on elliptic curves (in Chinese)," http://www.oscca.gov.cn/UpFile/2010122214822692.pdf, 2013.

[26] W. Pan, F. Zheng, W. Zhu, and J. Jing, "An efficient elliptic curve cryptography signature server with gpu acceleration," *IEEE Transactions on Information Forensics and Security*, 2017.

[27] F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Exploiting the potential of GPUs for modular multiplication in ECC," in *Information Security Applications - 15th International Workshop, WISA 2014, Jeju Island, Korea, August 25-27, 2014. Revised Selected Papers*, 2014, pp. 295–306.

[28] J. W. Bos, "Low-latency elliptic curve scalar multiplication," *International Journal of Parallel Programming*, vol. 40, no. 5, pp. 532–550, 2012.

[29] R. Szerwinski and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography," in *Cryptographic Hardware and Embedded Systems–CHES 2008*. Springer, 2008, pp. 79–99.

[30] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe, "High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers," *Designs, Codes and Cryptography*, vol. 77, no. 2-3, pp. 493–514, 2015.

[31] P. Koppermann, F. De Santis, J. Heyszl, and G. Sigl, "Low-latency x25519 hardware implementation: breaking the 100 microseconds barrier," *Microprocessors and Microsystems*, vol. 52, pp. 491–497, 2017.

[32] A. Faz-Hernández and J. López, "Fast implementation of curve25519 using avx2," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2015, pp. 329–345.

[33] E. Mahe and J.-M. Chauvet, "Fast gpgpu-based elliptic curve scalar multiplication." *IACR Cryptology ePrint Archive*, vol. 2014, p. 198, 2014.

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

[35] M. Hamburg, "Ed448-goldilocks, a new elliptic curve." *IACR Cryptology ePrint Archive*, vol. 2015, p. 625, 2015.

[36] NVIDIA, "NVIDIA Volta Architecture whitepaper," http://www.nvidia.com/object/volta-architecture-whitepaper.html, 2017.

[37] J. Dong, F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Utilizing the double-precision floating-point computing power of gpus for rsa acceleration," *Security and Communication Networks*, vol. 2017, 2017.

[38] NVIDIA, "NVIDIA Tesla P100 whitepaper," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[39] ——, "Parallel thread execution ISA version 6.0," http://docs.nvidia.com/cuda/parallel-thread-execution/index.html, 2017.

[40] ——, "NVIDIA Volta Architecture whitepaper," http://www.nvidia.com/object/volta-architecture-whitepaper.html, 2017.

[41] S. Josefsson and I. Liusvaara, "Edwards-curve digital signature algorithm (eddsa)," Tech. Rep., 2017.