Parallel Implementation of SM2 Elliptic Curve Cryptography on Intel Processors with AVX2

Junhao Huang[1], Zhe Liu[1,2(✉)], Zhi Hu[3], and Johann Großschädl[4]

[1] College of Computer Science and Technology,
Nanjing University of Aeronautics and Astronautics, Nanjing, China
jhhuang_nuaa@126.com
[2] State Key Laboratory of Cryptology, P.O. Box 5159, Beijing 100878, China
sduliuzhe@gmail.com
[3] Central South University, Hunan, China
huzhi_math@csu.edu.cn
[4] University of Luxembourg, Esch-sur-Alzette, Luxembourg
johann.groszschaedl@uni.lu

Abstract. This paper presents an efficient and secure implementation of SM2, the Chinese elliptic curve cryptography standard that has been adopted by the International Organization of Standardization (ISO) as ISO/IEC 14888-3:2018. Our SM2 implementation uses Intel's Advanced Vector Extensions version 2.0 (AVX2), a family of three-operand SIMD instructions operating on vectors of 8, 16, 32, or 64-bit data elements in 256-bit registers, and is resistant against timing attacks. To exploit the parallel processing capabilities of AVX2, we studied the execution flows of Co-Z Jacobian point arithmetic operations and introduce a parallel 2-way Co-Z addition, Co-Z conjugate addition, and Co-Z ladder algorithm, which allow for fast Co-Z scalar multiplication. Furthermore, we developed an efficient 2-way prime-field arithmetic library using AVX2 to support our Co-Z Jacobian point operations. Both the field and the point operations utilize branch-free (i.e. constant-time) implementation techniques, which increase their ability to resist Simple Power Analysis (SPA) and timing attacks. Our software for scalar multiplication on the SM2 curve is, to our knowledge, the first constant-time implementation of the Co-Z based ladder that leverages the parallelism of AVX2.

1 Introduction

Roughly 35 years ago, Koblitz and Miller proposed to use the group of points on an elliptic curve defined over a finite field for the implementation of discrete logarithm cryptosystems [17,19]. Today, *Elliptic Curve Cryptography (ECC)* is enjoying wide acceptance in the embedded/mobile domain due to the benefits of smaller key size, faster computation time, and reduced memory requirements compared to classic public-key cryptosystems [30]. Furthermore, ECC becomes increasingly popular in application domains where high data transmission rates

J. K. Liu and H. Cui (Eds.): ACISP 2020, LNCS 12248, pp. 204–224, 2020.
https://doi.org/10.1007/978-3-030-55304-3_11

(i.e. high throughput) are important, e.g. networking, web services, and cloud computing. The 64-bit Intel architecture plays a major role in the latter two domains, which makes a good case to optimize ECC software with respect to the computing capabilities of modern Intel processors, especially their parallel processing capabilities. In 2011, Intel presented a new set of SIMD instructions called *Advanced Vector Extensions 2 (AVX2)* that was first integrated into the Haswell microarchitecture. AVX2 instructions support integer operations with 256-bit vectors, which allows one to do calculations on e.g. four 64-bit integers in parallel, and have a "non-destructive" three-operand format, i.e. two source registers and one destination register. Even though AVX2 was mainly designed to accelerate graphics and video processing, it can also be leveraged to speed up cryptographic workloads like ECC computations.

The security of elliptic curve cryptosystems relies on the (presumed) hardness of the *Elliptic Curve Discrete Logarithm Problem (ECDLP)*, which asks to find the scalar k given two points P and $Q = kP$ on an elliptic curve [12]. An operation of the form kP, called *scalar multiplication*, is an integral part of all ECC schemes and, in general, their most computation-intensive component. In addition, the scalar multiplication can be vulnerable to side-channel attacks, in particular timing analysis or Simple Power Analysis (SPA), when implemented carelessly [4,28]. Therefore, an efficient and secure (in the sense of side-channel resistant) implementation of the scalar multiplication is crucial for any elliptic curve cryptosystem. The Montgomery ladder algorithm, originally proposed in [20], has a very regular execution profile, which lends itself to implementations with constant execution time, provided the underlying field arithmetic satisfies certain requirements. This algorithm is not only suitable for Montgomery curves like Curve25519 [1], but has also been generalized to Weierstrass elliptic curves [4,16]. The present paper focusses on a parallel constant-time implementation of the Montgomery ladder for Weierstrass curves (like SM2) using AVX2. An AVX2 implementation of Curve25519 can be found in e.g. [7].

SM2 was introduced by the State Cryptography Administration of China in 2010 [26] and is supposed to replace RSA and other public-key cryptographic algorithms for electronic authentication systems, key management systems, and application systems. In addition, SM2 was standardized by ISO/IEC in 2018 as ISO/IEC 14888-3:2018 [15]. Hence, in the next couple of years, SM2 will have excellent application prospects in both Chinese and international commercial electronic products. For all these reasons, it makes sense to investigate how the prime-field arithmetic, elliptic curve operations, and protocols using SM2 can be implemented efficiently, e.g. by utilizing the parallel computing capabilities of AVX2. It should be noted that many of the optimizations we present in this paper can also be applied to NIST P-256 or other Weierstrass curves by simply providing a parallel 2-way implementation of the field arithmetic.

1.1 Overview of Related Work and Motivation for Our Work

Currently, there exist only a few papers about implementing SM2 starting from the basic prime-field arithmetic up to the protocol level. Most implementations

of SM2 are based on the field arithmetic and elliptic curve operations provided by the open-source software OpenSSL [21]; typical examples are GmSSL[1] and TASSL[2]. However, implementation details of the prime-field arithmetic and the point (i.e. group) operations of SM2 are, to our knowledge, not documented in any form, which makes it difficult to reason about their efficiency.

To improve the execution time of public-key cryptographic algorithms like RSA and ECC on Intel processors, the SIMD-level parallelism of AVX2 can be exploited. Vector implementations of Montgomery modular multiplication and efficient modular exponentiation for RSA were introduced in [10,29]. Gueron and Krasnov presented in [11] a highly-optimized AVX2 software for fixed-base scalar multiplication on NIST's P-256 curve that executes four point additions in parallel. Taking advantage of AVX2 instructions, Faz-Hernández and López [7] developed an optimized Montgomery ladder for Curve25519, which performs two field-operations (e.g. two field-multiplications) simultaneously. In order to further reduce the latency, each field-multiplication (resp. squaring) multiplies two pairs of 25 or 26-bit limbs in parallel, whereby two limbs belonging to one operand are stored in a 128-bit lane of an AVX2 register. In a recent follow-up work, Faz-Hernández et al. [8] presented fast 2-way and 4-way implementations of the field-arithmetic and point operations using both the Montgomery model and the Edwards model of Curve25519. There are various other studies exploring the optimization of ECC for different vector instruction sets, such as Intel SSE2, Intel AVX-512, and ARM NEON, see e.g. [2,24].

Parallel implementations of the Montgomery ladder for GPUs and FPGAs have also been reported, some of which use Meloni's Co-Z Jacobian arithmetic from [18]. Bos [3] introduced a low-latency 7-way GPU implementation of an (X, Z)-only Co-Z ladder for the NIST curve P-224. Peng et al. [22] presented an optimized multi-core FPGA implementation of the X-only Co-Z ladder from [13] for a set of Weierstrass curves, whereby they combined a number of Montgomery modular multipliers to work in parallel. They concluded that a 3-core implementation achieves the best throughput-resource ratio.

1.2 Our Contributions

The contribution of this paper is twofold and can be summarized as follows:

1. We present novel parallel 2-way Co-Z Jacobian point arithmetic algorithms that utilize the parallel processing capabilities of Intel's AVX2 instruction set. Our parallel Co-Z addition, Co-Z conjugate addition, and combination thereof (i.e. the Co-Z ladder step) outperform their sequential counterparts by factors of about 1.26, 1.60, and 1.33, respectively. By pre-computing two values, we managed to resolve data dependencies in the parallel execution of the Co-Z ladder algorithm and minimize its execution time. Thanks to these parallel Co-Z point operations, our Co-Z based Montgomery ladder is 1.31 times faster than a sequential Co-Z Montgomery ladder.

[1] See http://gmssl.org (accessed on 2020–05–24).
[2] See http://github.com/jntass/TASSL (accessed on 2020–05–24).

2. To speed up the field arithmetic, we developed a fast 2-way implementation of modular reduction and carry propagation for the SM2 prime using the AVX2 instruction set. Both are integrated into our modular multiplication and modular squaring functions, which employ a radix-2^{26} representation of the operands. We aimed for resistance against timing/SPA attacks and avoided conditional statements like branch instructions to ensure the field arithmetic (and also point arithmetic) has operand-independent execution time. To achieve this, we utilized constant-time techniques such as operand masking, Fermat-based inversion, and a highly regular ladder algorithm.

The rest of this paper is structured as follows. In Sect. 2, we firstly provide a brief introduction to AVX2, the representation of operands, and the notation used throughout this paper. Section 3 presents the new parallel Co-Z Jacobian arithmetic and the Co-Z based Montgomery ladder algorithm. Thereafter, in Sect. 4, we introduce our implementation of the 2-way field-arithmetic for SM2 using AVX2 instructions. The results of our implementation are summarized in Sect. 5 and compared with the results from some previous papers. Finally, we give concluding remarks in Sect. 6.

2 Preliminaries

Overview of AVX2. Starting with the Haswell microarchitecture (released in 2013), modern 64-bit Intel processors support AVX2, which is, in essence, an extension of AVX to include 256-bit integer operations (classical AVX provides 256-bit floating-point instructions, but only 128-bit integer instructions). There are various AVX2 integer instructions that can be used to speed up prime-field arithmetic; the most important is `VPMULUDQ` (in the following abbreviated as MUL), which executes four (32×32)-bit multiplications in parallel and places the four 64-bit products in a 256-bit AVX2 register. Similarly, AVX2 contains instructions for parallel addition and subtraction of four packed 64-bit integers (abbreviated as ADD and SUB) [14]. Other members of the AVX2 instruction set with relevance for ECC include instructions to combine data elements from two AVX2 registers into a single one (BLEND), to shuffle data elements within a register (SHUF), to permute elements (PERM), to left/right shift elements by the same or different distances (SHL, SHR, SHLV, SHRV), to concatenate 128-bit lanes from two registers (ALIGN), and to carry out bit-wise operations (e.g. AND, XOR). We refer to [14] for a detailed description these instructions and to [9] for information about their latency and throughput.

Representation of Field Elements. It is common practice to represent the elements of a prime field $\mathbb{F}_p$ as integers in the range of $[0, p-1]$, which means they have a length of up to $m = \lceil \log_2(p) \rceil$ bits. An m-bit integer can be stored in an array of words ("limbs") whose bitlength equals the register size n of the target platform, e.g. $n = 64$. Arithmetic algorithms for addition, multiplication (and other operations) in $\mathbb{F}_p$ process these words using the instructions of the

processor, e.g. $(n \times n)$-bit multiply, n-bit add, n-bit add-with-carry, etc. While such a canonical radix-2^n representation of integers has the advantage that the total number of words $k = \lceil m/n \rceil$ is minimal for the target platform, it entails a lot of carry propagation and, as a consequence, sub-optimal performance on modern 64-bit processors [1,7]. Fortunately, it is possible to avoid most of the carry propagations by using a *reduced-radix representation* (also referred to as redundant representation [8]), which means the number of bits per limb n' is slightly less than the bitlength n of the processor's registers, e.g. $n' = 51$ when implementing Curve25519 for a 64-bit processor. In this way, several limbs can be added up in a 64-bit register without causing overflow and the result of the field-addition (and other arithmetic operations) does not necessarily need to be fully reduced, i.e. can be larger than p. Only at the very end of a cryptographic operation (e.g. scalar multiplication), a full reduction to the least non-negative residue and conversion to canonical form has to be carried out.

Formally, when using a reduced radix of $2^{n'}$ (i.e. $n' < n$ bits per limb), an m-bit integer A is represented via a sequence of limbs $(a_{k'-1}, a_{k'-2}, \ldots, a_0)$ so that $A = \sum_{i=0}^{k'-1} a_i 2^{in'}$, whereby a limb a_i does not necessarily need to be less than $2^{n'}$ but can (temporarily) become as big as $2^n - 1$. Although a reduced-radix representation may increase the number of limbs $k' = \lceil m/n' \rceil$ versus the full-radix setting (i.e. $k' > k$), there is typically still a net-gain in performance when taking advantage of "lazy carrying" and "lazy reduction" [8]. We will use uppercase letters to denote field elements and indexed lowercase letters for the individual limbs they consist of. As is usual practice, we analyze and compare the efficiency of point operations (i.e. addition and doubling) by counting the number of multiplications (M), squarings (S), additions/subtractions (A), and inversions (I) in the underlying finite field.

SM2 Elliptic Curve. The specific elliptic curve used for the implementation described in the following sections is SM2 [27], which is defined by a simplified Weierstrass equation $E : y^2 = x^3 + ax + b$ over a prime field $\mathbb{F}_p$. This field is given by the pseudo-Mersenne prime $p = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$ and allows for a special modular reduction method [25]. The curve parameter a is fixed to -3 to reduce the cost of the point arithmetic when using Jacobian projective coordinates [12]. A Jacobian projective point $(X : Y : Z)$, $Z \neq 0$ corresponds to the affine point $(x, y) = (X/Z^2, Y/Z^3)$. The projective form of the Weierstrass equation is

$$E : \; Y^2 = X^3 - 3XZ^4 + bZ^6. \tag{1}$$

Like other standardized Weierstrass curves, the cardinality $\#E(\mathbb{F}_p)$ of the SM2 curve is prime, i.e. it has a co-factor of $h = 1$. The full specification of the SM2 curve can be found in [27].

Co-Z Jacobian Arithmetic. First proposed by Meloni [18], Co-Z Jacobian arithmetic is based on the observation that the addition of two distinct points in projective coordinates can be accelerated when they are represented with the same Z-coordinate. As specified by Eq. (2), the sum $R = P + Q$ of the points

$P = (X_P, Y_P, Z)$ and $Q = (X_Q, Y_Q, Z)$ can computed at an overall cost of five multiplications (5M), two squarings (2S) and seven additions/subtractions (7A) in $\mathbb{F}_p$, which is significantly less than the cost of a conventional point addition using Jacobian projective coordinates (12M+4S+7A, see [6, Sect. 2.3]) and lies even below the 8M+3S+7A for a "mixed" Jacobian-affine addition [6,12]. This Co-Z point-addition technique was applied by Rivain [23] to develop a fast and regular Montgomery ladder algorithm that is suitable for scalar multiplication on Weierstrass curves and does not require the order to be divisible by 4.

$$\begin{aligned} &A = (X_Q - X_P)^2,\ B = X_P A,\ C = X_Q A,\ D = (Y_Q - Y_P)^2,\ E = Y_P(C - B) \\ &X_R = D - (B + C),\ Y_R = (Y_Q - Y_P)(B - X_R) - E,\ Z_R = Z(X_Q - X_P) \end{aligned} \quad (2)$$

Note that the Co-Z addition formula also yields a new representation of the point $P = (X_P, Y_P, Z)$ because $B = X_P(X_Q - X_P)^2$, $E = Y_P(X_Q - X_P)^3$, and $Z_R = Z(X_Q - X_P)$ [23]. Consequently, $(X_P, Y_P, Z) \sim (B, E, Z_R)$, which means this new representation of the point P and the sum $R = P + Q$ have the same Z-coordinate. According to Eq. (3) the difference $R' = P - Q = (X'_R, Y'_R, Z_R)$ can be computed with very little extra cost and has the same Z-coordinate as $R = P + Q$. In total, 6M+3S+11A are needed to obtain $P + Q$ and $P - Q$.

$$\begin{aligned} &A, B, C, D, \text{ and } E \text{ as in Eq. (2)},\ F = (Y_P + Y_Q)^2 \\ &X'_R = F - (B + C),\ Y'_R = (Y_P + Y_Q)(X'_R - B) - E \end{aligned} \quad (3)$$

Venelli and Dassance [28] presented a further optimization of Co-Z arithmetic by eliminating the computation of the Z-coordinate from the formulae for the Co-Z addition and Co-Z conjugate addition. Concretely, they proposed a novel Co-Z Montgomery ladder algorithm based on addition formulae that compute only the X and Y-coordinate of the intermediate points (we refer to this kind of operation as "(X, Y)-only addition"). The Z-coordinate can be recovered at the end of the ladder at little extra cost. Omitting the Z-coordinates reduces the computational cost of the Co-Z addition and the Co-Z conjugate addition by 1M to 4M+2S+7A and 5M+3S+11A, respectively. The implementation we present in this paper is based on (X, Y)-only Co-Z operations.

3 Parallel Co-Z Jacobian Arithmetic for SM2

In this section we first demonstrate that most of the field-arithmetic operations of Co-Z addition and Co-Z conjugate addition can be executed in parallel and then we present a ladder that exploits the processing capabilities of AVX2.

3.1 Parallel Co-Z Jacobian Point Addition

In order to utilize the parallelism of AVX2, we carefully analyzed the execution flow of the (X, Y)-only Co-Z Jacobian arithmetic and found that many of the field operations have no sequential dependency and can, therefore, be executed

Algorithm 1. SIMD_XYCZ_ADD: Parallel (X, Y)-only Co-Z addition

Input: $P = (X_P, Y_P)$, $Q = (X_Q, Y_Q)$.
Output: $(R, P') = (P + Q, P)$ where $P' \sim P$ has the same Z-coordinate as $P + Q$.

1:	$T_1 = X_P - X_Q$	$T_2 = Y_Q - Y_P$	{sub}
2:	$A = T_1^2$	$D = T_2^2$	{sqr}
3:	$B = X_P A$	$C = X_Q A$	{mul}
4:	$T_1 = B + C$		{add}
5:	$X_R = D - T_1$	$T_3 = C - B$	{sub}
6:	$T_1 = B - X_R$		{sub}
7:	$T_1 = T_1 T_2$	$E = Y_P T_3$	{mul}
8:	$Y_R = T_1 - E$		{sub}
9:	**return** $((X_R, Y_R), (B, E))$		

Algorithm 2. SIMD_XYCZ_ADDC: Parallel (X, Y)-only Co-Z conjugate addition

Input: $P = (X_P, Y_P)$, $Q = (X_Q, Y_Q)$, $A' = (X_Q - X_P)^2$, $T' = (X_Q - X_P)A' = C' - B'$
Output: $(R, R') = (P + Q, P - Q)$

1:	$C = X_Q A'$	$E = Y_P T'$	{mul}
2:	$B = C - T'$	$T_1 = Y_Q - Y_P$	{sub}
3:	$T_2 = B + C$	$T_3 = Y_P + Y_Q$	{add}
4:	$D = T_1^2$	$F = T_3^2$	{sqr}
5:	$X_R = D - T_2$	$X'_R = F - T_2$	{sub}
6:	$T_2 = B - X_R$	$T_4 = X'_R - B$	{sub}
7:	$T_2 = T_1 T_2$	$T_3 = T_3 T_4$	{mul}
8:	$Y_R = T_2 - E$	$Y'_R = T_3 - E$	{sub}
9:	**return** $((X_R, Y_R), (X'_R, Y'_R))$		

in parallel. This applies, for example, to the temporary values B and C of the formulae for the Co-Z addition given in Eq. (2), which means it is possible to obtain them simultaneously with a 2-way parallel field-multiplication. Also the computation of A and D can be "paired" and performed simultaneously if the used field-arithmetic library supports 2-way parallel squaring. Algorithm 1 and Algorithm 2 are optimized implementations of the (X, Y)-only Co-Z addition and Co-Z conjugate addition, respectively, whereby the prime-field operations are scheduled to facilitate a 2-way parallel execution. Each line performs two times the same operation in parallel using two sets of operands (the operation being carried out is commented on the right). Unfortunately, some operations of Algorithm 1 could not be paired (line 4, 6, and 8), but those operations are relatively cheap additions and subtractions. On the other hand, all operations of Algorithm 2 are performed pair-wise, but this became only possible because of the pre-computation of A' and T' (we will discuss further details of this pre-computation below). Without pre-computation of A' and T', the latency of the (X, Y)-only Co-Z conjugate addition would be significantly worse.

The 2-way parallel execution of the Co-Z point addition almost halves the latency compared to the straightforward (i.e. sequential) scheduling of the field operations. More concretely, the latency of the (X, Y)-only Co-Z addition

Algorithm 3. SIMD_XYCZ_ADDC_ADD: Parallel Co-Z ladder step

Input: $P = (X_P, Y_P) = R_a$, $Q = (X_Q, Y_Q) = R_{1-a}$, $A = (X_Q - X_P)^2$, $T' = (X_Q - X_P)A' = C' - B'$ where $a \in \{0, 1\}$ and R_a, R_{1-a} are two Co-Z Jacobian points that are intermediate results of the Montgomery ladder algorithm.

Output: $(R_a, R_{1-a}) = (2R_a, R_a + R_{1-a})$ and update of $A' = (X_{R_a} - X_{R_{1-a}})^2$ and $T' = (X_{R_{1-a}} - X_{R_a})A'$.

1:	$C' = X_Q A'$	$E' = Y_P T'$	{mul}
2:	$B' = C' - T'$	$T_1 = Y_Q - Y_P$	{sub}
3:	$T_2 = B' + C'$	$T_3 = Y_P + Y_Q$	{add}
4:	$D' = T_1^2$	$F' = T_3^2$	{sqr}
5:	$X_R = D' - T_2$	$X'_R = F' - T_2$	{sub}
6:	$T_2 = B' - X_R$	$T_4 = X'_R - B'$	{sub}
7:	$T_2 = T_1 T_2$	$T_4 = T_3 T_4$	{mul}
8:	$Y_R = T_2 - E'$	$Y'_R = T_4 - E'$	{sub}
9:	$T_1 = X'_R - X_R$	$T_2 = Y'_R - Y_R$	{sub}
10:	$A = T_1^2$	$D = T_2^2$	{sqr}
11:	$X_P = B = X_R A$	$C = X'_R A$	{mul}
12:	$T_3 = T_2 + B$	$T_4 = B + C$	{add}
13:	$X_Q = D - T_4$	$T_1 = C - B$	{sub}
14:	$T_4 = X_Q - X_P$	$T_3 = T_3 - X_Q$	{sub}
15:	$A' = T_4^2$	$T_3 = T_3^2$	{sqr}
16:	$T' = T_4 A'$	$X_P = E = Y_R T_1$	{mul}
17:	$T_1 = D + A'$	$T_2 = E + E$	{add}
18:	$T_3 = T_3 - T_1$		{sub}
19:	$Y_Q = \frac{1}{2}(T_3 - T_2)$		{sub}
20:	**return** $((X_Q, Y_Q), (X_P, Y_P))$		

decreases from 4M+2S+7A to 2M̈+1S̈+5Ä, i.e. the delay due to multiplications and squarings is reduced by 50% (assuming that 2-way parallel field-operations have the same delay as single field-operations). We abbreviate a 2-way parallel multiplication, squaring, and addition (resp. subtraction) in $\mathbb{F}_p$ by M̈, S̈, and Ä, respectively, to distinguish them from the corresponding simple 1-way field operations. The 2-way parallel scheduling of the field-arithmetic decreases the latency of the (X, Y)-only Co-Z conjugate addition from 5M+3S+11A for the sequential variant given by Eq. (3) to 3M̈+1S̈+6Ä (this latency includes the pre-computation of A' and T', which will be discussed below).

As shown in [28] is is possible to convert the basic Montgomery ladder into a Co-Z based ladder algorithm by simply replacing the operations in the main loop by a (X, Y)-only Co-Z conjugate addition followed by a (X, Y)-only Co-Z addition as shown in Eq. (4). Algorithm 3 combines these two operations into a single "ladder step," which we optimized for an arithmetic library that is capable to execute the field operations in a 2-way parallel fashion. We designed Algorithm 3 by firstly analyzing the sequential versions of the Co-Z addition and CoZ conjugate addition. Their combined latency is 9M+5S+18A, but an optimization described in [23, Sect. A.2] (which replaces a field-multiplication by one squaring and four additions) makes it possible to reduce the latency to

Algorithm 4. Co-Z based Montgomery ladder algorithm

Input: A point $P \neq \mathcal{O}$, a scalar $k \in \mathbb{F}_p$ satisfying $k_{n-1} = 1$.
Output: The result of the scalar multiplication $R = k \cdot P$.

1: $(R_1, R_0) = \text{XYCZ\_IDBL}(P)$
2: $a = k_{n-2}$
3: $A' = (X_{R_{1-a}} - X_{R_a})^2$, $T' = (X_{R_{1-a}} - X_{R_a})A'$
4: **for** i from $n-2$ by 1 down to 1 **do**
5: $\quad a = (k_i + k_{i+1}) \bmod 2$
6: $\quad (R_a, R_{1-a}, A', T') = \text{SIMD\_XYCZ\_ADDC\_ADD}(R_a, R_{1-a}, A', T')$
7: **end for**
8: $a = (k_0 + k_1) \bmod 2$
9: $(R_{1-a}, R_a) = \text{SIMD\_XYCZ\_ADDC}(R_a, R_{1-a}, A', T')$
10: $\frac{\lambda}{Z} = \text{FinalInvZ}(R_{1-a}, R_a, P, a)$
11: $(R_0, R_1) = \text{SIMD\_XYCZ\_ADD}(R_0, R_1)$
12: **return** $((\frac{\lambda}{Z})^2 X_{R_0}, (\frac{\lambda}{Z})^3 Y_{R_0})$

8M+6M+22A. This indicates that, in theory, a parallel implementation of the ladder step using 2-way parallel field operations could have a latency as low as 4M̈+3S̈+11Ä. However, the latency of a parallel Co-Z addition (Algorithm 1) together with the parallel Co-Z conjugate addition (Algorithm 2) amounts to 5M̈+2S̈+11Ä and does not reach this (theoretical) lower bound. We then tried to reschedule the field operations of the sequential 8M+6M+22A ladder step in order to optimize it for 2-way parallel execution, but some data dependencies did not allow us to reach the best possible latency of 4M̈+3S̈+11Ä.

In order to obtain the minimal latency, we propose to pre-compute the two terms $A' = (X_Q - X_P)^2$ and $T' = (X_Q - X_P)A'$ before entering the main loop of the ladder algorithm and update A' and T' in each iteration (as part of the ladder step, see Algorithm 3). In this way, we managed to perfectly resolve the data dependencies and achieve a latency of 4M̈+3S̈+13Ä, which is close to the minimum (all field operations except two subtractions at the very end could be properly paired, which makes Algorithm 3 very well suited for a 2-way parallel execution of field operations). Compared to the combination of Co-Z addition and Co-Z conjugate addition, the proposed ladder step trades 1M̈ for 1S̈ and 2Ä, which reduces the latency in our case (see Sect. 5).

3.2 Parallel Co-Z Based Montgomery Ladder

The Montgomery ladder can not only be used for Montgomery curves, but also for general Weierstrass curves [4,16], which includes the SM2 curve. Venelli and Dassance [28] proposed (X, Y)-only Co-Z arithmetic and further optimized the Co-Z based Montgomery ladder algorithm by avoiding the computation of the Z-coordinate during the main loop of the scalar multiplication.

Algorithm 4 shows our (X, Y)-only Co-Z Montgomery ladder based on the parallel ladder step described before. It starts by computing the initial points $(R_1, R_0) = (2P, P)$ for the ladder using a doubling operation with Co-Z update

(called XYCZ_IDBL, see [23, Sect. C] for details). Thereafter, the two values A' and T' are pre-computed, which is necessary to minimize the latency of the parallel Co-Z ladder step as discussed before. During the execution of the main loop, our parallel Co-Z ladder algorithm with minimum latency maintains the following relationship between the ladder points:

$$\begin{aligned}(R_{1-a}, R_a) &= (R_a + R_{1-a}, R_a - R_{1-a}) \\ R_a &= R_{1-a} + R_a \end{aligned} \tag{4}$$

where $a = (k_i + k_{i+1}) \bmod 2$. As Algorithm 4 shows, our parallel Co-Z based Montgomery ladder executes a parallel Co-Z ladder step in each iteration, and has therefore a regular execution profile and constant execution time. The two constants A', T' get updated with each call of the ladder-step function. At the end of the ladder, a conversion from Co-Z Jacobian coordinates to affine coordinates needs to be carried out. We perform this conversion with the function FinalInvZ, which computes $Z = X_P Y_{R_a}(X_{R_0} - X_{R_1})$, $\lambda = y_P X_{R_a}$ and outputs $\frac{\lambda}{Z}$ at a cost of 1I+3M+1A, i.e. this conversion requires an inversion in $\mathbb{F}_p$.

Due to the parallel (X, Y)-only Co-Z Jacobian arithmetic, our Co-Z based ladder outperforms the sequential Co-Z ladder by a factor of roughly 1.31. To the best of our knowledge, the parallel Co-Z Montgomery ladder we presented in this section is the first attempt of minimizing the latency of a variable-base scalar multiplication by combining (X, Y)-only Co-Z Jacobian point arithmetic with a 2-way parallel implementation of the prime-field arithmetic.

4 2-Way Parallel Prime-Field Arithmetic for SM2

The Co-Z based Montgomery ladder presented in the previous section requires a 2-way parallel implementation of the arithmetic operations in the underlying prime field $\mathbb{F}_p$. As explained in Sect. 2, the prime field used by SM2 is defined by the 256-bit generalized-Mersenne prime $p = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$. The special form of p allows one to speed up the modular reduction [25].

We explained in Sect. 2 that, in order to reduce carry propagation, it makes sense to use a reduced-radix representation on modern 64-bit processors. This is also the case when implementing multi-precision integer arithmetic for SIMD engines like AVX2 since they do not offer an add-with-carry instruction. The implementation we introduce in this section adopts a radix-2^{26} representation for the field elements, i.e. a 256-bit integer consists of $k' = \lceil 256/26 \rceil = 10$ limbs and each limb is $n' = 26$ bits long (but can temporarily become longer). When putting four limbs into an AVX2 register, it is possible to perform four limb-multiplications in parallel, each producing a 52-bit result. However, each of the 10 limbs can become as long as 29 bits without causing an overflow during the multiplication of field elements since the sum of 10 limb-products still fits into 64 bits: $10 \times 2^{29} \times 2^{29} < 2^{62}$. In addition, since 10 is a multiple of two, we can split a field element evenly into five limb-pairs for 2-way parallel execution.

The AVX2 implementation of the $\mathbb{F}_p$-arithmetic we describe below performs an arithmetic operation in a 2-way parallel fashion, which two times the same

operation is executed, but with different operands. Each of the two operations uses pairs of limbs instead of a single limb as "smallest unit" of processing. We put a limb-pair of operand A into the higher 128-bit lane of a 256-bit AVX2 register and a limb-pair of operand B into the lower 128-bit lane, i.e. there are four limbs altogether in an AVX2 register. Consequently, we need five registers to store all limbs of A and B. Similar to [7], we use a set of *interleaved tuples* $\langle A, B\rangle_i$ for $i \in [0, 5)$ to denote such five AVX2 registers, whereby the i-th tuple $\langle A, B\rangle_i$ contains the four limbs $[a_{2i+1}, a_{2i}, b_{2i+1}, b_{2i}]$.

4.1 Addition and Subtraction

Due to the redundant representation, the 2-way addition/subtraction over two sets of interleaved tuples $\langle A, B\rangle_i \pm \langle C, D\rangle_i = \langle A \pm C, B \pm D\rangle_i$ can be performed by executing five AVX2 add (ADD)/subtract (SUB) instructions that operate on 64-bit data elements in parallel. To avoid overflow during the addition, we assert that the length of each limb of the operands must not exceed 63 bits. On the other hand, to avoid underflow during subtraction, we add an appropriate multiple of the SM2 prime p to $\langle A, B\rangle_i$ and then perform the subtraction. We do not reduce the result modulo p unless the next operation would overflow.

4.2 Modular Multiplication and Squaring

Multiplication/Squaring. Our implementation of the 2-way parallel multiplication using AVX2 instructions was inspired by the work of Faz-Hernández and López for Curve25519 [7] (with some modifications for the SM2 prime). As shown in Algorithm 5, the outer loop (starting at line 4) traverses through the set of interleaved tuples $\langle A, B\rangle_i$. Since there are two limbs of each A and B in an interleaved tuple, we have two inner loops and use the SHUF instruction to separate the two limbs of A and B. Each of the inner loops traverses through the set $\langle C, D\rangle_j$, multiplies the tuple $\langle C, D\rangle$ (or $\langle C', D'\rangle$) by the shuffled tuple $\langle A, B\rangle$, and adds the obtained partial-product to Z_{i+j} (or Z_{i+j+1}). Due to the radix-2^{26} representation, we can assure that this multiply-accumulate process does not overflow the 64-bit data element in which the sum is kept. When the multiplication is finished, we call the function FastRed (Algorithm 6) to get the final result $\langle E, F\rangle_i = \langle A \cdot C \bmod p, B \cdot D \bmod p\rangle_i$. Squaring is quite similar to the multiplication, except that a number of MUL instructions can be replaced by left-shift (i.e. SHL) instructions, see e.g. [7, Algorithm 4].

Fast Reduction. There exist some well-known modular reduction techniques for arbitrary primes, such as the algorithms of Barrett or Montgomery [12]. The 2-way modular reduction we implemented takes advantage of the generalized-Mersenne form of the SM2 prime, which allows for a special reduction method with linear complexity [25]. However, we had to re-design the fast reduction to make it compatible with our radix-2^{26} representation, see Algorithm 6. We use the congruence relations specified by Eq. (5) to "fold" the upper half of the

Algorithm 5. 2-way parallel multiplication using AVX2 instructions

Input: Two sets of interleaved tuples $\langle A, B\rangle_i$, $\langle C, D\rangle_i$ with $A, B, C, D \in \mathbb{F}_p$.
Output: Modular product $\langle E, F\rangle_i = \langle AC \bmod p, BD \bmod p\rangle_i$.

1: $Z_i = 0$ for $i \in [0, 10)$
2: **for** i from 0 by 1 up to 4 **do**
3: $\quad\langle C', D'\rangle_i = \text{ALIGN}(\langle C, D\rangle_{i+1 \bmod 5}, \langle C, D\rangle_i)$ $\{[c_{2i+2}, c_{2i+1}, d_{2i+2}, d_{2i+1}]\}$
4: **end for**
5: **for** i from 0 by 1 up to 4 **do**
6: $\quad U = \text{SHUF}(\langle A, B\rangle_i, \langle A, B\rangle_i, 0\text{x}44)$ $\{[a_{2i}, a_{2i}, b_{2i}, b_{2i}]\}$
7: $\quad$**for** j from 0 by 1 up to 4 **do**
8: $\quad\quad Z_{i+j} = \text{ADD}(Z_{i+j}, \text{MUL}(U, \langle C, D\rangle_j))$ $\{[a_{2i}c_{2j+1}, a_{2i}c_{2j}, b_{2i}d_{2j+1}, b_{2i}d_{2j}]\}$
9: $\quad$**end for**
10: $\quad V = \text{SHUF}(\langle A, B\rangle_i, \langle A, B\rangle_i, 0\text{xEE})$ $\{[a_{2i+1}, a_{2i+1}, b_{2i+1}, b_{2i+1}]\}$
11: $\quad$**for** j from 0 by 1 up to 3 **do**
12: $\quad\quad Z_{i+j+1} = \text{ADD}(Z_{i+j+1}, \text{MUL}(V, \langle C', D'\rangle_j))$
$\{[a_{2i+1}c_{2j+2}, a_{2i+1}c_{2j+1}, b_{2i+1}d_{2j+2}, b_{2i+1}d_{2j+1}]\}$
13: $\quad$**end for**
14: $\quad W = \text{MUL}(V, \langle C, D\rangle_4)$ $\{[a_{2i+1}c_0, a_{2i+1}c_9, b_{2i+1}d_0, b_{2i+1}d_9]\}$
15: $\quad Z_i = \text{ADD}(Z_i, \text{BLEND}(W, [0,0,0,0], 0\text{x}33))$ $\{[a_{2i+1}c_0, 0, b_{2i+1}d_0, 0]\}$
16: $\quad Z_{i+5} = \text{ADD}(Z_{i+5}, \text{BLEND}(W, [0,0,0,0], 0\text{xCC}))$ $\{[0, a_{2i+1}c_9, 0, b_{2i+1}d_9]\}$
17: **end for**
18: $\langle E, F\rangle_i = \text{FastRed(Z)}$ {Algorithm 6}
19: **return** $\langle E, F\rangle_i$

20-limb product Z, i.e. the 10 limbs z_{10} to z_{19}, which have a weight of between 2^{260} and 2^{494}), into the lower half of Z. Our fast modular reduction technique replaces the large powers of two on the left side of Eq. (5), which all exceed 2^{260}, by sums of smaller powers of two (i.e. less than 2^{260}) based on the special form of $p = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$. Consequently, the modular reduction boils down to basic shifts, additions, and subtractions of limbs. However, note that Eq. (5) assumes the limbs of the product Z to be 26 bits long.

$$
\begin{aligned}
z_{10}2^{260} &\equiv z_{10}(2^{228} + 2^{100} - 2^{68} + 2^4) \bmod p \\
z_{11}2^{286} &\equiv z_{11}(2^{254} + 2^{126} - 2^{94} + 2^{30}) \bmod p \\
z_{12}2^{312} &\equiv z_{12}(2^{248} + 2^{152} - 2^{88} + 2^{56} + 2^{24}) \bmod p \\
z_{13}2^{338} &\equiv z_{13}(2^{242} + 2^{178} + 2^{50} + 2^{18}) \bmod p \\
z_{14}2^{364} &\equiv z_{14}(2^{236} + 2^{204} + 2^{108} + 2^{44} + 2^{12}) \bmod p \\
z_{15}2^{390} &\equiv z_{15}(2^{231} + 2^{134} + 2^{102} + 2^{38} + 2^6) \bmod p \\
z_{16}2^{416} &\equiv z_{16}(2^{257} + 2^{160} + 2^{128} + 2^{64} + 2^{32}) \bmod p \\
z_{17}2^{442} &\equiv z_{17}(2^{251} + 2^{186} + 2^{154} + 2^{123} - 2^{90} + 2^{58} + 2^{27}) \bmod p \\
z_{18}2^{468} &\equiv z_{18}(2^{245} + 2^{212} + 2^{180} + 2^{149} + 2^{116} - 2^{84} + 2^{53} + 2^{21}) \bmod p \\
z_{19}2^{494} &\equiv z_{19}(3 \cdot 2^{238} + 2^{206} + 2^{175} + 2^{142} + 2^{110} + 2^{47} + 2^{15}) \bmod p
\end{aligned}
\tag{5}
$$

The modular reduction function specified in Algorithm 6 first converts the limbs in the upper half of the AVX2 registers into a radix-2^{28} form, i.e. 28 bits

Algorithm 6. 2-way parallel modular reduction using AVX2 instructions

Input: A set of interleaved tuples Z consisting of 20 limbs.
Output: Modular-reduced set of interleaved tuples $\langle E, F \rangle_i$ consisting of 10 limbs.

1: **for** i from 4 by 1 up to 8 **do**
2: $L_i = \text{AND}(Z_i, [2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1])$
3: $M_i = \text{SHR}(Z_i, [n', n', n', n'])$
4: $M_i = \text{AND}(M_i, [2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1])$
5: $H_i = \text{SHR}(Z_i, [2n', 2n', 2n', 2n'])$
6: **end for**
7: $L_9 = \text{AND}(Z_9, [2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1])$
8: $M_9 = \text{SHR}(Z_9, [n', n', n', n'])$
9: **for** i from 5 by 1 up to 9 **do**
10: $M'_i = \text{ALIGN}(M_i, M_{i-1})$
11: $Z_i = \text{ADD}(\text{ADD}(L_i, M'_i), H_{i-1})$
12: **end for**
13: $Z_9 = \text{ALIGN}(Z_9, M_9)$
14: $\langle E, F \rangle_i = \text{SimpleRed(Z)}$
15: **return** $\langle E, F \rangle_i$

per limb. Since the reduction is carried out immediately after a multiplication or squaring, the maximum limb-length can be 60 bits. In order to reduce the length of the upper limbs to 28 bits, we first split them into three parts of up to $n' = 26$ bits: a lower part containing the 26 least significant bits, a middle part consisting of the next 26 bits, and a higher part with the rest. Each of the parts has a certain weight, and parts of the same weight (which can be up to three) are added together, yielding limbs of a length of at most 28 bits. This conversion is performed by the two loops of Algorithm 6 (i.e. line 1–12). At the end of these loops, we have an intermediate result Z of which the lower limbs are less than 60 bits long, while the upper limbs can have a length of up to 28 bits (with exception of the limbs in Z_9, which can be up to 30 bits long). The actual modular reduction based on the congruence relations of Eq. (5) is then carried out in line 14 by the SimpleRed operation (explained in Sect. A). Note that SimpleRed produces a result consisting of 10 limbs, whereby each limb is less than $2^4 \cdot 2^{28} \cdot 2^{24} + 2^{60} < 2^{61}$ and easily fits in a 64-bit data element (this remains correct when the maximum limb-length is 30 instead of 28 bits).

4.3 Carry Propagation (Conversion to 28-Bit Limbs)

The result of the SimpleRed operation consists of 10 limbs (which are stored in five AVX2 registers), whereby each limb is smaller than 2^{61}, i.e. no more than 60 bits long. However, a result given in such a form needs to be converted into a representation with limbs of a length of $n' = 26$ bits (or a little longer). This conversion requires a method to "carry" the excess bits of a limb over to the next-higher limb, and to reduce the excess bits of the highest limb modulo the prime p). To achieve this, each 60-bit limb has to be split into three parts as follows: $a_i = h_i || m_i || l_i$ where $|l_i| = |m_i| = n'$ and $|h_i| = 60 - 2n'$ (similar as

Algorithm 7. 2-way parallel carry propagation using AVX2 instructions

Input: A set of interleaved tuples $\langle A, B\rangle_i$.
Output: A set of interleaved tuples $\langle A, B\rangle_i$ with $|a_i|, |b_i| \leq n' + 1$ for $i \in [0, 10)$.

1: $H_4 = \text{SHRV}(\langle A, B\rangle_4, [n', 2n', n', 2n'])$
2: $\langle A, B\rangle_4 = \text{AND}(\langle A, B\rangle_4, [2^{n'} - 1, 2^{2n'} - 1, 2^{n'} - 1, 2^{2n'} - 1])$
3: $Q = \text{ADD}(H_4, \text{SHUF}(H_4, H_4, 0\text{x}4\text{E}))$
4: $Q' = \text{SUB}([0,0,0,0], Q)$
5: $\langle A, B\rangle_0 = \text{ADD}(\langle A, B\rangle_0, \text{SHLV}(\text{BLEND}(Q, [0,0,0,0], 0\text{xCC}), [0,4,0,4]))$
6: $\langle A, B\rangle_1 = \text{ADD}(\langle A, B\rangle_1, \text{SHLV}(\text{BLEND}(Q, Q', 0\text{x}33), [22,16,22,16]))$
7: $\langle A, B\rangle_4 = \text{ADD}(\langle A, B\rangle_4, \text{SHLV}(\text{BLEND}(Q, [0,0,0,0], 0\text{xCC}), [0,20,0,20]))$
8: **for** i from 0 by 1 up to 4 **do**
9: $\quad L_i = \text{AND}(\langle A, B\rangle_i, [2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1])$
10: $\quad M_i = \text{SHR}(\langle A, B\rangle_i, [n', n', n', n'])$
11: $\quad M_i = \text{AND}(M_i, [2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1, 2^{n'} - 1])$
12: $\quad H_i = \text{SHR}(\langle A, B\rangle_i, [2n', 2n', 2n', 2n'])$
13: **end for**
14: **for** i from 0 by 1 up to 4 **do**
15: $\quad M'_i = \text{ALIGN}(M_i, M_{i-1 \bmod 5})$
16: $\quad \langle A, B\rangle_i = \text{ADD}(\text{ADD}(L_i, M'_i), H_{i-1 \bmod 5})$
17: **end for**
18: **return** $\langle A, B\rangle$

in the previous subsection). Algorithm 7 specifies the conversion. In line 1–7 we estimate the excess bits of the highest limb, which means we estimate a value q of weight 2^{260} by computing $q = (h_{k'-1} || m_{k'-1} + h_{k'-2})$. Then, we reduce the value q via the congruence $q \cdot 2^{260} \equiv q \cdot (2^{228} + 2^{100} - 2^{68} + 2^4) \bmod p$, i.e. we add q to (or subtract it from) limbs with the corresponding terms. The code in line 8–17 reduces the bit-length of the remaining limbs based on the equations $a'_0 = l_0$, $a'_1 = l_1 + m_0$, and $a'_i = l_i + m_{i-1} + h_{i-2}$ for $i \in [2, 10)$. This algorithm ensures $|a'_i| <= 28$ for $i \in [0, k' - 1]$, which means the limbs are within a safe range so that they can serve as operand in any of our field operations.

4.4 Modular Inversion

Modular inversion is the most costly among the prime-field operations needed in ECC. Using Jacobian projective coordinates, we only need one inversion to convert the result from projective to affine coordinates. The Binary Extended Euclidean Algorithm (BEEA) is a well-known algorithm for inversion, but has an irregular execution flow and operand-dependent execution time, which can enable timing and SPA attacks [5]. Therefore, we chose Fermat's little theorem and perform the inversion through an exponentiation: $a^{-1} \equiv a^{p-2} \bmod p$. When utilizing an addition chain as in e.g. [30], the modular inversion can be carried out at an overall cost of 15M+255S.

5 Performance Evaluation

We benchmarked the described implementation on a 64-bit Intel Cascade Lake processor clocked at 2.5 GHz. The execution times we present in this section were obtained by measuring the cycles for 10^6 iterations of the field-operations and 10^4 iterations of the point operations, on a single core and single thread.

Table 1. Comparison of the execution time of the 1-way and 2-way implementation of the prime-field operations (in clock cycles)

Implementation	Add/sub	Mul	Sqr	Inv	Carry prop.
1-way (1 op)	5	75	65	18,459	23
2-way (2 ops)	8	99	81	–	30
Speed-up ratio	1.25	1.52	1.60	–	1.53

Table 1 provides the timings of a standard (i.e. 1-way) and a 2-way parallel implementation of the prime-field operations. The results show that, when one and the same operation has to be performed on two sets of field-elements, the 2-way parallel implementation is much faster than two subsequent executions of the basic 1-way version; the speed-up factors range from 1.25 to 1.60. Since our 2-way field arithmetic is based on techniques from [7], the execution times of the operations are similar. However, the reduction modulo the SM2 prime is more complicated (and thus slower) than the reduction for Curve25519.

Table 2. Execution time (in clock cycles) of Co-Z addition, Co-Z conjugate addition, Co-Z ladder step, and Co-Z based Montgomery ladder algorithm.

Implementation	Co-Z ADD	Co-Z ADDC	Co-Z L-Step	Co-Z Ladder
Sequential	555	786	1,334	359,868
2-way parallel	439	489	1,001	274,908
Speed-up ratio	1.26	1.60	1.33	1.31

Table 2 shows the execution times of the sequential and the 2-way parallel version of the Co-Z addition, Co-Z conjugate addition, Co-Z ladder step, and Co-Z based full Montgomery ladder algorithm. Similarly as above, the parallel versions clearly outperform their sequential counterparts. The 2-way parallel CoZ Montgomery ladder has an execution time of 275k clock cycles, which is 1.31 times faster than the sequential ladder. To the best of our knowledge, this paper is the first to present a Co-Z based Montgomery ladder utilizing AVX2 instructions and to demonstrate the ability of a 2-way parallel implementation of the field arithmetic to speed up Co-Z based Jacobian point operations.

Table 3 compares our parallel Co-Z Montgomery ladder with similar AVX2 implementations of variable-base scalar multiplication on Curve25519 and the

Table 3. Comparison of the computational cost and execution time (in clock cycles on a Cascade Lake or Haswell processor) of our Co-Z based Montgomery ladder and other AVX2 implementations of variable-base scalar multiplication.

Implementation	Cost per bit	Additional cost	Execution time
SM2 (this work)	4M̈+3S̈+13Ä	1I+8M+7S+12A	274,908 (CL)
Curve25519 [7]	3M̈+2S̈+1C̈+4Ä	1I+1M	156,500 (H)
Curve25519 [8]	3M̈+2S̈+1C̈+4Ä	1I+1M	121,000 (H)
NIST P-256 [11]	n/a	n/a	312,000 (H)

NIST curve P-256. Since Curve25519 is Montgomery curve [20], it supports an efficient "X-coordinate-only" algorithm for variable-base scalar multiplication that costs only 5M+4S+1C+8A per scalar-bit ("C" stands for a multiplication of a field-element by a curve constant, which is normally much faster than an ordinary field-multiplication). Furthermore, as already mentioned, a reduction modulo the 255-bit pseudo-Mersenne prime used by Curve25519 can be carried out more efficiently than a reduction modulo the SM2 prime. Faz-Hernández and López reported in [7] an execution time of roughly 156,500 Haswell cycles for their AVX2 implementation of Curve25519, which is significantly better than our 274,908 clock cycles for SM2 on a more recent Cascade Lake CPU. There are three main reasons for this difference in execution time. First, as shown in Table 3, the parallel version of the Co-Z based ladder-step for the SM2 curve is 1M̈+1S̈+9Ä more costly than the parallel ladder-step for Montgomery curves (i.e. [7, Algorithm 1]). The second reason is the higher additional cost outside the ladder loop for such tasks like the initial point doubling, the computation of the values A' and T', and the recovery of the Z-coordinate at the end of the ladder. Finally, the reduction modulo the SM2 prime is more complicated, and therefore slower, than the reduction modulo $p = 2^{255} - 19$.

Table 4. Comparison of the execution time of ECDH key exchange and ECDSA signature generation/verification using the SM2 curve on a 2.5 GHz Cascade Lake processor and the NIST curve P-256 on a 3.4 GHz Haswell processor.

Implementation	Processor	ECDH key ex.	ECDSA sign	ECDSA verify
SM2 (this work)	Cascade Lake	148 µs	24 µs	98 µs
NIST P-256 [11]	Haswell	93 µs	41 µs	122 µs

Gueron and Krasnov presented in [11] optimized implementations of fixed-base and variable-base scalar multiplication, both specifically optimized for the NIST curve P-256. Their paper includes benchmarking results generated on an Intel Haswell processor clocked at 3.4 GHz; some of these results can be found in Table 3 and Table 4 (taken from the operations/second in [11, Fig. 7]). The fixed-base scalar multiplication uses a windowing method with a window size of 7 and performs four point additions in parallel in the AVX2 engine. On the other hand, the variable-base scalar multiplication has smaller windows of size

5 and executes the point operations sequentially (the prime-field arithmetic is written in x86_64 assembly and does not exploit the parallelism of AVX2). The execution time in the variable-base case is 312,000 Haswell cycles (about 92 µs when the clock frequency is 3.4 GHz). Gueron and Krasnov also benchmarked ECDH key exchange (computation of the shared key only), ECDSA signature generation (mainly fixed-base scalar multiplication) and the verification of an ECDSA signature; the corresponding timings are listed in Table 4.

Also given in Table 4 are the execution times of our AVX2 implementation of SM2-based ECDH key exchange, and the generation and verification of an ECDSA signature, which we measured on a 2.5 GHz Cascade Lake CPU. We implemented the fixed-base scalar multiplication (for signature generation) in the same way as [11], i.e. by means of a windowing method with a window size of 7. The execution time of our fixed-base scalar multiplication is about 64,000 Cascade Lake cycles. Note that the SM2 key exchange protocol authenticates the involved parties and, therefore, the computation of the shared key requires two scalar multiplications. This explains why SM2 key exchange is slower than a basic static ECDH key exchange. Unfortunately, a comparison of the results of our implementation with that of [11] is difficult since the micro-architectural properties and features of Haswell and Cascade Lake differ significantly.

6 Conclusions

We introduced a 2-way parallel implementation of SM2 prime-field arithmetic and Co-Z Jacobian point operations that leverage the processing capabilities of AVX2. Due to a careful rescheduling of the field arithmetic along with the pre-computation of two values outside the main loop, we managed to minimize the execution time of the parallel Co-Z ladder algorithm. More concretely, the 2-way parallel field arithmetic and Co-Z Jacobian point operations reduce the execution time of the ladder algorithm for variable-base scalar multiplication by a factor of 1.31 compared to sequential execution. Furthermore, our parallel ladder has a highly regular execution profile, which helps to achieve resistance against timing and SPA attacks. The main take-away message of this paper is that SIMD-level parallelism helps to narrow the performance gap between the classical Montgomery ladder on Montgomery curves and the Co-Z ladder on Weierstrass curves. When executed sequentially, the difference between these two scalar multiplication methods is 3M+2S (i.e. 5M+4S vs. 8M+6S), but this difference shrinks to $1\ddot{\mathrm{M}}+1\ddot{\mathrm{S}}$ ($3\ddot{\mathrm{M}}+2\ddot{\mathrm{S}}$ vs. $4\ddot{\mathrm{M}}+3\ddot{\mathrm{S}}$) in the case of 2-way parallel execution. Finally, we remark that all optimization techniques proposed in this paper can also be applied to the NIST curves.

Acknowledgments. Zhe Liu is supported by the National Natural Science Foundation of China (grant no. 61802180), the Natural Science Foundation of Jiangsu Province (grant no. BK20180421), the National Cryptography Development Fund (grant no. MM-JJ20180105) and the Fundamental Research Funds for the Central Universities (grant no. NE2018106). Zhi Hu is supported by the Natural Science Foundation of China (grants no. 61972420, 61602526) and the Hunan Provincial Natural Science Foundation of China (grants no. 2019JJ50827 and 2020JJ3050).

A SimpleRed Operation

Based on the congruence relations in Eq. (5), we add or subtract each of the upper limbs z_i with $i \in [10, 20)$ to the corresponding lower limbs in Z to obtain the residue $\langle E, F\rangle_i$ from the intermediate result Z. For example, all the terms with weight $2^0 \sim 2^{26}$ and $2^{26} \sim 2^{52}$ will be added to or subtracted from Z_0 to obtain $\langle E, F\rangle_0$. Similarly to Z_0, the terms with other weights will be added to or subtracted from the corresponding terms of the intermediate result Z. The details are fully specified in Algorithm 8, which executes only simple additions (resp. subtractions), shifts, and permutation instructions.

Algorithm 8. 2-way parallel SimpleRed operation using AVX2 instructions

Input: An intermediate result Z consisting of 20 limbs.
Output: A modular residue $\langle E, F\rangle_i$ consisting of ten 28-bit limbs.

1: $\langle E, F\rangle_0 = \text{ADD}(Z_0,\text{SHL}(Z_5,4))$
2: $\langle E, F\rangle_0 = \text{ADD}(\langle E, F\rangle_0,\text{SHL}(Z_6,24))$
3: $\langle E, F\rangle_0 = \text{ADD}(\langle E, F\rangle_0,\text{SHL}(\text{SHUF}(Z_6,Z_7,0\text{x}5),18))$
4: $\langle E, F\rangle_0 = \text{ADD}(\langle E, F\rangle_0,\text{SHL}(Z_7,12))$
5: $\langle E, F\rangle_0 = \text{ADD}(\langle E, F\rangle_0,\text{SHL}(\text{SHUF}(Z_7,Z_8,0\text{x}5),6))$
6: $\langle E, F\rangle_0 = \text{ADD}(\langle E, F\rangle_0,\text{SHL}(Z_9,21))$
7: $\langle E, F\rangle_0 = \text{ADD}(\langle E, F\rangle_0,\text{SHLV}(\text{BLEND}(\text{PERM64}(Z_9,0\text{xB1}),Z_8,0\text{xCC}),[1,15,1,15]))$
8: $Z'_8 = \text{SUB}([0,0,0,0],Z_8),\ \ Z'_9 = \text{SUB}([0,0,0,0],Z_9)$
9: $\langle E, F\rangle_1 = \text{SUB}(\text{Z}_1,\text{SHL}(Z_5,16))$
10: $\langle E, F\rangle_1 = \text{ADD}(\langle E, F\rangle_1,\text{SHLV}(\text{BLEND}(Z_6,\text{PERM64}(Z_5,0\text{xB1}),0\text{xCC}),[22,4,22,4]))$
11: $\langle E, F\rangle_1 = \text{SUB}(\langle E, F\rangle_1,\text{SHLV}(\text{SHUF}([0,0,0,0],Z_6,0\text{x}5),[10,0,10,0]))$
12: $\langle E, F\rangle_1 = \text{ADD}(\langle E, F\rangle_1,\text{SHL}(\text{BLEND}(Z_8,Z'_8,0\text{xCC}),12))$
13: $\langle E, F\rangle_1 = \text{ADD}(\langle E, F\rangle_1,\text{SHL}(\text{SHUF}(Z_8,Z'_9,0\text{x}5),6))$
14: $\langle E, F\rangle_1 = \text{ADD}(\langle E, F\rangle_1,\text{SHLV}(\text{BLEND}(Z_9,Z_7,0\text{xCC}),[24,1,24,1]))$
15: $\langle E, F\rangle_2 = \text{ADD}(Z_2,\text{SHL}(\text{SHUF}(Z_5,Z_6,0\text{x}5),22))$
16: $\langle E, F\rangle_2 = \text{ADD}(\langle E, F\rangle_2,\text{SHL}(Z_7,4))$
17: $\langle E, F\rangle_2 = \text{ADD}(\langle E, F\rangle_2,\text{SHL}(Z_8,24))$
18: $\langle E, F\rangle_2 = \text{ADD}(\langle E, F\rangle_2,\text{SHL}(\text{SHUF}(Z_8,Z_9,0\text{x}5),19))$
19: $\langle E, F\rangle_2 = \text{ADD}(\langle E, F\rangle_2,\text{SHL}(Z_9,12))$
20: $\langle E, F\rangle_2 = \text{ADD}(\langle E, F\rangle_2,\text{SHL}(\text{SHUF}(Z_9,[0,0,0,0],0\text{x}5),6))$
21: $\langle E, F\rangle_3 = \text{ADD}(Z_3,\text{SHL}(\text{SHUF}(Z_6,Z_7,0\text{x}5),22))$
22: $\langle E, F\rangle_3 = \text{ADD}(\langle E, F\rangle_3,\text{SHL}(Z_8,4))$
23: $\langle E, F\rangle_3 = \text{ADD}(\langle E, F\rangle_3,\text{SHL}(Z_9,24))$
24: $\langle E, F\rangle_3 = \text{ADD}(\langle E, F\rangle_3,\text{SHL}(\text{SHUF}(Z_9,[0,0,0,0],0\text{x}5),19))$
25: $\langle E, F\rangle_4 = \text{ADD}(Z_4,\text{SHL}(Z_5,20))$
26: $\langle E, F\rangle_4 = \text{ADD}(\langle E, F\rangle_4,\text{SHLV}(\text{SHUF}(Z_7,Z_7,0\text{x}5),[2,23,2,23]))$
27: $\langle E, F\rangle_4 = \text{ADD}(\langle E, F\rangle_4,\text{SHL}(Z_9,4))$
28: $T = \text{ADD}(\text{SHLV}(Z_6,[8,14,8,14]),\text{SHLV}(Z_8,[17,23,17,23]))$
29: $T = \text{ADD}(T,\text{SHLV}(Z_9,[5,11,5,11]))$
30: $\langle E, F\rangle_4 = \text{ADD}(\langle E, F\rangle_4,\text{ADD}(\text{SHUF}([0,0,0,0],T,0\text{x}5),\text{BLEND}([0,0,0,0],T,0\text{xCC}))$
31: **return** $\langle E, F\rangle_{0\ldots4}$

References

1. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_14
2. Bernstein, D.J., Schwabe, P.: NEON crypto. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 320–339. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33027-8_19
3. Bos, J.W.: Low-latency elliptic curve scalar multiplication. Int. J. Parallel Prog. **40**(5), 532–550 (2012). https://doi.org/10.1007/s10766-012-0198-5
4. Brier, É., Joye, M.: Weierstraß elliptic curves and side-channel attacks. In: Naccache, D., Paillier, P. (eds.) PKC 2002. LNCS, vol. 2274, pp. 335–345. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45664-3_24
5. Cabrera Aldaya, A., Cabrera Sarmiento, A.J., Sánchez-Solano, S.: SPA vulnerabilities of the binary extended Euclidean algorithm. J. Cryptogr. Eng. **7**(4), 273–285 (2017). https://doi.org/10.1007/s13389-016-0135-4
6. Cohen, H., Miyaji, A., Ono, T.: Efficient elliptic curve exponentiation using mixed coordinates. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 51–65. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49649-1_6
7. Faz-Hernández, A., López, J.: Fast implementation of curve25519 using AVX2. In: Lauter, K., Rodríguez-Henríquez, F. (eds.) LATINCRYPT 2015. LNCS, vol. 9230, pp. 329–345. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22174-8_18
8. Faz-Hernández, A., López, J., Dahab, R.: High-performance implementation of elliptic curve cryptography using vector instructions. ACM Trans. Math. Softw. **45**(3), 1–35 (2019)
9. Fog, A.: Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. Manual (2019). http://www.agner.org/optimize/instruction_tables.pdf
10. Gueron, S., Krasnov, V.: Software implementation of modular exponentiation, using advanced vector instructions architectures. In: Özbudak, F., Rodríguez-Henríquez, F. (eds.) WAIFI 2012. LNCS, vol. 7369, pp. 119–135. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31662-3_9
11. Gueron, S., Krasnov, V.: Fast prime field elliptic-curve cryptography with 256-bit primes. J. Cryptogr. Eng. **5**(2), 141–151 (2015). https://doi.org/10.1007/s13389-014-0090-x
12. Hankerson, D.R., Menezes, A.J., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer, New York (2004). https://doi.org/10.1007/b97644
13. Hutter, M., Joye, M., Sierra, Y.: Memory-constrained implementations of elliptic curve cryptography in co-Z coordinate representation. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 2011. LNCS, vol. 6737, pp. 170–187. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21969-6_11
14. Intel Corporation: Intel instruction set architecture extensions. Documentation (2013). http://software.intel.com/en-us/isa-extensions

15. International Organization for Standardization: ISO/IEC 14888–3:2018 - IT security techniques - Digital signatures with appendix - Part 3: Discrete logarithm based mechanisms (2018)
16. Izu, T., Takagi, T.: A fast parallel elliptic curve multiplication resistant against side channel attacks. In: Naccache, D., Paillier, P. (eds.) PKC 2002. LNCS, vol. 2274, pp. 280–296. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45664-3_20
17. Koblitz, N.I.: Elliptic curve cryptosystems. Math. Comput. **48**(177), 203–209 (1987)
18. Meloni, N.: New point addition formulae for ECC applications. In: Carlet, C., Sunar, B. (eds.) WAIFI 2007. LNCS, vol. 4547, pp. 189–201. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73074-3_15
19. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_31
20. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243–264 (1987)
21. OpenSSL Software Foundation: OpenSSL. Software (2019). http://www.openssl.org
22. Peng, B.-Y., Hsu, Y.-C., Chen, Y.-J., Chueh, D.-C., Cheng, C.-M., Yang, B.-Y.: Multi-core FPGA implementation of ECC with homogeneous Co-Z coordinate representation. In: Foresti, S., Persiano, G. (eds.) CANS 2016. LNCS, vol. 10052, pp. 637–647. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48965-0_42
23. Rivain, M.: Fast and regular algorithms for scalar multiplication over elliptic curves. Cryptology ePrint Archive, Report 2011/338 (2011)
24. Seo, H., Liu, Z., Großschädl, J., Choi, J., Kim, H.: Montgomery modular multiplication on ARM-NEON revisited. In: Lee, J., Kim, J. (eds.) ICISC 2014. LNCS, vol. 8949, pp. 328–342. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15943-0_20
25. Solinas, J.A.: Generalized Mersenne numbers. Technical report CORR-99-39, University of Waterloo, Waterloo, Canada (1999)
26. State Cryptography Administration of China: Public key cryptographic algorithm SM2 based on elliptic curves. Specification (2010). http://www.sca.gov.cn/sca/xwdt/2010-12/17/content_1002386.shtml
27. State Cryptography Administration of China: Recommended curve parameters of public key cryptographic algorithm SM2 based on elliptic curves. Specification (2010). http://www.sca.gov.cn/sca/xwdt/2010-12/17/content_1002386.shtml
28. Venelli, A., Dassance, F.: Faster side-channel resistant elliptic curve scalar multiplication. In: Kohel, D., Rolland, R. (eds.) Contemporary Mathematics, vol. 512, pp. 29–40. American Mathematical Society (2010)

29. Zhao, Y., Pan, W., Lin, J., Liu, P., Xue, C., Zheng, F.: PhiRSA: exploiting the computing power of vector instructions on Intel Xeon Phi for RSA. In: Avanzi, R., Heys, H. (eds.) SAC 2016. LNCS, vol. 10532, pp. 482–500. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69453-5_26
30. Zhou, L., Su, C., Hu, Z., Lee, S., Seo, H.: Lightweight implementations of NIST P-256 and SM2 ECC on 8-bit resource-constraint embedded device. ACM Trans. Embed. Comput. Syst. **18**(3), 1–13 (2019)