Efficient Additions and Montgomery Reductions of Large Integers for SIMD

Pengchang Ren, Reiji Suda, and Vorapong Suppakitpaisarn
The University of Tokyo, Tokyo, Japan
Email: splight@gmail.com, reiji@is.s.u-tokyo.ac.jp, vorapong@is.s.u-tokyo.ac.jp

***Abstract*—This paper presents efficient algorithms, designed to leverage SIMD for performing additions and Montgomery reductions on integers larger than 512 bits. The existing algorithms encounter inefficiencies when parallelized using SIMD due to extensive dependencies in both operations, particularly noticeable in ARM's SVE where SIMD operations are costly. To mitigate this problem, a novel addition algorithm is introduced that simulates the addition of large integers using a smaller addition, quickly producing the same set of carries. These carries are then utilized to perform parallel additions on large integers. For Montgomery reductions, serial multiplications are replaced with precomputations that can be effectively calculated using SIMD extensions. Experimental evidence demonstrates that these proposed algorithms substantially enhance the performance of state-of-the-art implementations of several post-quantum cryptography algorithms. Notably, they deliver a 30% speed-up from the latest CTIDH implementation, an 11% speed-up from the latest CSIDH implementation in AVX-512 processors, and a 7% speed-up from Microsoft's standard PQCrypto-SIDH for SIKEp503 on A64FX.**

***Index Terms*—arithmetic for cryptography, Montgomery reduction, additions, SIMD, AVX-512, SVE**

I. Introduction

Prime field arithmetic forms the basis of many public-key cryptography algorithms, including RSA, ECDH, and various post-quantum cryptosystems such as SIKE [1], CSIDH [2], and CTIDH [3].

The increasing availability of SIMD instructions on modern processors has highlighted the importance of utilizing these instructions to optimize the latency performance of large prime-field arithmetic. Recent research has shown that the latest support for 52-bit integer multiplication for AVX-512 on x64 can efficiently optimize prime field arithmetic for either throughput or latency [4], [5].

On the other hand, attempts to use ARM's Neon instructions have yielded little improvement for latency [6], [7]. This could be because Neon only allows for 32-bit integer multiplication and 128-bit SIMD, which may not be suitable for prime field arithmetic when dealing with primes as large as 512 bits.

ARM has recently announced the introduction of a new SIMD extension known as SVE, designed to enhance their ARMv8 instruction set. SVE offers several improvements over Neon, including support for 64-bit integer multiplication and the ability to process larger vector lengths. In our prior research, we have enhanced the throughput of SIKE on SVE [8]. Furthermore, to further optimize latency performance, Edamatsu et al. [9] have proposed a low-latency multiplication technique for large numbers specifically designed for SVE.

Previous research has primarily focused on optimizing the latency of multiplication operations using SIMD instructions and assumed that the algorithms for calculating addition and Montgomery reduction are already efficient. However, this assumption may not hold true for certain SIMD instructions, such as SVE, where additions and reductions can be expensive. This research has shown that these operations can become a bottleneck in various calculations, particularly in post-quantum cryptographic algorithms. Therefore, it is crucial to develop efficient algorithms that consider addition, multiplication, and reduction operations to fully leverage the benefits of SIMD instructions.

A. Our Contributions

The costs of additions and Montgomery reductions in SIMD are due to their long dependency chains. When using carry propagate adders, adding more significant limbs requires waiting for carries from less significant limbs before starting the addition. Similarly, Montgomery reductions are typically carried out limb by limb [10], [11]. The results of the calculation of the less significant limbs are required for the calculation of the more significant ones. This research aims to break these dependency chains.

Section IV introduces a novel addition algorithm that utilizes smaller additions to simulate the addition of large integers that require multiple limbs to store. The smaller additions give different results than the original large additions but produce the same set of carries. By using these carries, parallel additions on each limb of the large integers can be performed. For instance, the calculation of carries for an addition of 512-bit integers, requiring 8 limbs for storage in SVE, can be achieved by performing an addition of 64-bit integers, which requires only a single limb. The 64-bit addition can then serve as a basis to derive all the required carries for the original addition, enabling the parallel execution of additions across the eight limbs, predicated on the obtained carries.

Section V presents two techniques for Montgomery reduction. The first technique can be applied to any prime number, while the second is specifically designed for prime numbers of the form $2^\ell F - 1$. In the first technique, we notice that serial calculations in the reduction can be replaced with additions of independent multiplication results, which can be computed efficiently using SIMD. In the second technique, it

is demonstrated that reductions on such prime numbers can be performed using just two large multiplications. Then, by leveraging concepts from [5], [12], the Karatsuba algorithm is utilized to speed up these large multiplication operations.

Lastly, experimental results for the proposed techniques are shown in Section VI. For addition, benchmark results show that the proposed algorithm is faster than the carry propagation addition implemented in AVX-512 by up to 2.5 times. The proposed SIMD addition is faster than the carry propagation addition implemented in x64 by up to 32%.

The proposed techniques for Montgomery reduction have shown significant speed improvements over previous implementations. Benchmark results demonstrate a 11% speedup for Montgomery multiplication and a 36% speedup for Montgomery squaring, compared to the state-of-art implementation in AVX-512 implementation by Cheng et al. [5]. Compared to x64 implementation, the improvement is 97% for Montgomery multiplication and 151% for Montgomery reduction.

Furthermore, the proposed techniques can accelerate the calculation of CSIDH by 11% compared to the work by Cheng et al. [5], and an 77% compared to x64 implementation.

In addition, a 30% increase in speed can be achieved compared to the latest implementation of CTIDH, by Benegas et al. [3] in SVE. Moreover, the proposed Montgomery reduction for the prime number $p_{503} = 2^{250}3^{159} - 1$, which is the standard prime used for SIKE, proves to be 26% faster than the reduction in Microsoft's standard PQCrypto-SIDH. This contributes to an overall 7% improvement in computation for SIKE.

II. Preliminaries

A. Large Integer Representation

The implementation of prime field arithmetic requires careful consideration of how to store large integers in memory. An approach is to use radix-2^ω, where a large integer is stored using several ω-bit integers. One ω bit integer is called a *limb*. If ω is equal to the machine word size, this is referred to as using the *native radix*, while if ω is smaller, it is called *reduced radix*. Throughout this paper, the symbol n is used to represent the number of limbs required to store a large integer.

Suppose a large integer requires 8 limbs to store and a single register can hold up to 8 limbs. It is possible to use one register to store all 8 limbs of the integer. Alternatively, it is possible to utilize two registers to pack two integers, storing four limbs of each integer in each register. The representation of the packing of x integers together is referred to as *x-way packing*.

The optimal values of x and ω can vary depending on the architecture and prime field being used, and selecting them is outside the scope of this investigation.

B. Single Instruction Multiple Data (SIMD)

SIMD instructions are a class of instructions that enable the manipulation of multiple data elements in a single instruction. They are found in modern CPUs, including AVX-512 on x64, Neon, and SVE on ARM. SIMD instructions offer different levels of support for various calculations and different vector length. For instance, AVX-512 has a vector length of 512 bits, Neon's vector length is 128 bits, while SVE ranges from 128 to 2048 bits.

TABLE I: Latency in clock cycles for instructions and Cycles per Instruction (CPI) for Intel Tigerlake's x64 and AVX-512, as well as Fujitsu A64FX's A64 and SVE

Operation	Tigerlake x64		Tigerlake AVX-512		A64FX A64	A64FX SVE
	Latency	CPI	Latency	CPI	Latency	Latency
Cache Access	3	0.5	4	0.5	5	11
Addition	1	0.25	1	0.5	1	4
Logic	1	0.25	1	0.5	1	4
Shift	1	0.5	1	1	2	4
Compare	1	0.25	3	1	1	4
Popcount	3	1	3	1	-	4
Multiplication, 64-bit	3	1	-	-	5	9
Multiplication, 52-bit	-	-	4	1	-	-
Table lookup/Cross-lane	-	-	3	1	-	6

A single vector has the capacity to store multiple integers, with each integer occupying a specific section of memory referred to as a *lane*. AVX-512 register can be interpreted as 64 8-bit lanes, 32 16-bit lanes, 16 32-bit lanes or eight 64-bit lanes. With most SIMD instructions, performing an operation using SIMD means applying that operation to all lanes of vectors simultaneously. For example, adding two vectors involves adding each lane of the two vectors independently. In this study, the terms SIMD_ADD and SIMD_SUB denote the process of carrying out parallel addition and subtraction operations using SIMD technology. For multiplication operations, when two one-limb operands yield a two-limb result, the terms SIMD_MUL_H and SIMD_MUL_L are employed. These denote the process of calculating the more and less significant limbs of the outcome, respectively. In contrast, there are instructions designed to move data across lanes, like the table lookup instruction TBL in SVE. These are referred to as cross-lane instructions and are generally more computationally expensive.

Intel CPUs from the 10th generation and later have extensively incorporated AVX-512. While Neon is available on numerous modern ARM CPUs, SVE with 512-bit vectors is now exclusively available on CPUs intended for high-performance computing, such as A64FX.

Table I presents latency in cycle times and throughput metrics as cycles per instruction (CPI) for both Tigerlake's x64 and AVX-512. It also includes latency figures for A64 and SVE on Fujitsu's A64FX [13], [14]. Latency denotes the time required to complete a single computation for the instructions, while throughput is defined as latency over the number of instructions that can be processed simultaneously. Throughput data is typically considered when executing several instructions concurrently, whereas latency data applies to tasks that cannot be carried out in parallel. Regrettably, Fujitsu has not supplied throughput information for each instruction, resulting in the listing of only latency data.

Avoiding instructions with high latency is desirable for any program. Table I demonstrates that cache access incurs a significant cost for all instruction sets listed, highlighting the need to minimize data movement between SIMD registers and general purpose registers. This is also true for cross-lane

instructions.

The table also highlights that all SVE instructions on A64FX exhibit high latency. Typically, when optimizing calculations, the number of additions and comparisons is not a major concern as they are not as expensive as other operations. However, in the case of SVE, their costs are significant and cannot be ignored.

C. Isogeny-based Cryptography: CSIDH, CTIDH, and SIKE

CSIDH [2] is an isogeny-based cryptography algorithm developed by Castryck et al. While the original implementation is not constant-time, which could lead to timing attacks, several attempts have been made to find a fast constant-time implementation [15], [16], [17], [18]. Unfortunately, according to the discussion in Cheng et al. [5], these implementations are significantly slower than the implementation which is not constant-time. CTIDH is a new algorithm developed by Banegas et al. [3] that improves constant-time performance, with over 60% speed-ups achieved by changing the key space of CSIDH.

Both CSIDH and CTIDH are defined over a finite field $\mathbb{F}_p$, where $p = (4 \cdot \prod l_i) - 1$ and l_i-s are small odd primes. Although both algorithms offer various primes to meet different security levels, most optimizations for CSIDH have been discussed for primes with 511 bits [5], [15], [16], [17], [18]. Most source codes for CSIDH and CTIDH are written in x64 assembly. Up to our knowledge, only implementation by Jalali et al. [16] is on ARM architecture. A constant-time implementation of CSIDH for ARM architecture was proposed in the same work, but it was computationally intensive. Currently, there is no implementation for CTIDH on ARM architecture.

SIKE [1] is another isogeny-based cryptosystem based on Supersingular Isogeny Diffie-Hellman (SIDH) key exchange [19]. It was one of the most well-known isogeny-based cryptosystem, and has been proposed as one of the four alternate candidates in the fourth round of NIST's PQC standardization process. However, in August 2022, it was reported that an algorithm [20] could attack the current implementation of SIKE, and subsequent research showed that minor changes do not make the protocol secure [21], [22]. While none of the algorithms proposed in this paper are exclusive to SIKE, but, as the implementations of SIKE were highly optimized, they are used as benchmarks of this work.

The protocol of SIKE is defined over the quadratic extension of the finite field $\mathbb{F}_{p^2}$, where $p = 2^{e_1}3^{e_2} - 1$ and $2^{e_1} \approx 3^{e_2}$. It defines four parameter sets, with prime bit lengths of 434, 503, 610, and 751. Microsoft has an implementation of SIKE called PQCrypto-SIDH [23], optimized for x64 and ARM architecture using handwritten assembly. The reference implementation used in this paper is SIDHv3.5.

III. Previous Approaches on Additions and Montgomery Reductions

A. Carry-Propagate Addition

One way to implement addition is through digit-by-digit carry propagation. The addition is calculated for each limb, starting from the least significant limb to the most significant limb, while considering the carry. Subtraction can be carried out using a similar approach.

General purpose instructions can calculate the addition between two large integer A and B efficiently. For A, B stored in radix 2^ω, denote the i-th least significant limb of A and B by A_i and B_i, i.e., $A = \sum_{i=0}^{n-1} 2^{i \cdot \omega} A_i$, and $B = \sum_{i=0}^{n-1} 2^{i \cdot \omega} B_i$. The carry propagation addition of A and B can be simply implemented with repeatedly calculating $D_i \leftarrow A_i + B_i + c_i \bmod 2^\omega$ and $c_{i+1} \leftarrow \lfloor (A_i + B_i + c_i)/2^\omega \rfloor$, starting with $c_0 = 0$.

Unfortunately, the carry propagation algorithm performs poorly in SIMD. In order to obtain a value c_i for the $(i+1)$-th limb addition, all less significant limbs must be added first, creating a long dependency chain that cannot be parallelized. Additionally, the result c_{i+1} is stored in the i-th lane of the SIMD register, but it is needed in the $(i+1)$-th lane, requiring an expensive cross-lane operation unless each SIMD register only contains one limb. These issues are severe when implementing on A64FX, where all SVE instructions have high latency, as shown in Table I.

B. Carry-Select Addition [24]

The carry-select addition approach was introduced to circumvent the long dependency chain in carry-propagate addition. In carry-propagate addition, it is necessary to wait for the value of c_i before computing the sum $D_i = A_i + B_i + c_i$. Rather than waiting for the availability of $c_i \in \{0, 1\}$, carry-select addition computes the sum results for both potential values of c_i in advance. That is, the adders calculate $D_i^{(0)} = (A_i + B_i) \bmod 2^\omega$, $D_i^{(1)} = (A_i + B_i + 1) \bmod 2^\omega$, $c_{i+1}^{(0)} = \lfloor (A_i + B_i)/2^\omega \rfloor$, and $c_{i+1}^{(1)} = \lfloor (A_i + B_i + 1)/2^\omega \rfloor$ for $i > 1$. Since $D_0 = (A_0 + B_0) \bmod 2^\omega$ and $c_1 = \lfloor (A_0 + B_0)/2^\omega \rfloor$, it becomes feasible to use the c_i value from the less significant limb to select the value of D_i and c_{i+1} at the more significant limb, meaning $D_i = D_i^{(c_i)}$ and $c_{i+1} = c_{i+1}^{(c_i)}$.

With this approach, it becomes feasible to eliminate additions from the lengthy dependency chain through the precomputation of $D_i^{(0)}$, $D_i^{(1)}$, $c_{i+1}^{(0)}$, and $c_{i+1}^{(1)}$. However, this elimination needs the computation of results for both c_i values, effectively doubling the calculation effort. Furthermore, the long dependency chain continues to persist within the selection process.

C. Kogge-Stone Vector Addition

An approach presented in [25], based on the Kogge-Stone vector addition technique [26], demonstrates high efficiency in the AVX-512 architecture. This approach uses specific AVX-512 instructions to compute all c_i values concurrently, thereby mitigating the lengthy dependency chain found in the carry-select adder.

We will not discuss the specifics of this approach due to its reliance on unique AVX-512 instructions, which limits its applicability to other architectures. Moreover, these instruc-

tions yield c_i values on a register called mask registers[1]. This requires a transfer of these values to general-purpose registers. Although this transfer operation is relatively quick on AVX-512, it needs significantly more time-consuming on SVE.

Nevertheless, we use this approach as a benchmark for the AVX-512 architecture. We also attempt to implement this approach on SVE, using it as our benchmark in this respective architecture.

D. Montgomery Reduction [10]

Let $R = 2^{\omega n}$, and let R^{-1} be an integer that satisfies $R \cdot R^{-1} \equiv 1 \pmod{p}$. Define a *Montgomery reduction* of T when $T < pR$ as *REDC*(T), which calculates $TR^{-1} \bmod p$. Additionally, define a partial Montgomery reduction function, denoted as *redc*(T, k) for $0 < k \leq \omega n$, as a function which returns a positive integer such that *redc*$(T, k) \equiv T2^{-k} \pmod{p}$. Calculating modular reduction can be done by multiplying T with R^{-1} and performing the modulo operation. However, the modulo operation can be computationally expensive.

Algorithm ExistingGenericRedc: Generic Montgomery reduction

Input: $T < pR$, where $p < 2^{\omega n}, r = 2^{\omega}, R = 2^{\omega n}$, r^{-1}, R^{-1} are positive integers such that $rr^{-1} \equiv 1 \mod p$, $RR^{-1} \equiv 1 \mod p$, and $p' \leftarrow \frac{rr^{-1}-1}{p}$

Output: $REDC(T) = TR^{-1} \bmod p$

1 $T^{(0)} \leftarrow T$
2 **for** $i = 1$ **to** n **do**
3 $\quad Q \leftarrow T^{(i-1)}p' \bmod r$
4 $\quad T^{(i)} \leftarrow (T^{(i-1)} + Qp)/r$
5 **end**
6 **if** $T^{(n)} > p$ **then**
7 $\quad T^{(n)} \leftarrow T^{(n)} - p$
8 **return** $T^{(n)}$

Consider the Algorithm ExistingGenericRedc. It is shown in [10] that the variable $T^{(i)}$ is an integer and is a partial reduction $redc(T, \omega i)$. Hence, $T^{(n)} \equiv T2^{-\omega n} \equiv TR^{-1} \pmod{p}$. It is also shown in the paper that, by the final reduction at Lines 6-7, the value of $T^{(n)}$ at Line 8 is smaller than p, and hence $T^{(n)} = TR^{-1} \bmod p = REDC(T)$.

By utilizing the algorithm, computing the modulo p can be avoided. Instead, the reduction can be calculated by a series of modulo and division operations using $r = 2^{\omega}$. Since all the integers are stored in binary, modulo and division by r can be computed by discarding some of the binary words if radix-2^{ω} is used.

E. Montgomery Representation and Multiplication

In this paper, elements are taken from the prime field $\mathbb{F}_p = \{0, \ldots, p-1\}$. The multiplication of $A, B \in \mathbb{F}_p$ is computed as $A \cdot B \bmod p$, where the multiplication can be done efficiently, but the modulo operation afterwards can be time-consuming. Montgomery reduction in the previous subsection helps reducing the computation of the multiplication [10].

Recall from the previous subsection that $R = 2^{\omega n} > p$. Since p is an odd number, it is coprime to R. Therefore, for any $A, B \in \mathbb{F}_p$ with $A \neq B$, $AR \not\equiv BR \pmod{p}$. Hence, A can be represented with another representation $\widehat{A} = AR \bmod p$, known as the *Montgomery representation* of A.

The application of the Montgomery reduction to $\widehat{A}\widehat{B}$ enables the calculation of the Montgomery representation of AB. This is because *REDC*$(\widehat{A}\widehat{B}) = (AR)(BR)R^{-1} \bmod p = (AB)R \bmod p$. This approach allows us to perform the reduction operation following each multiplication rather than incurring the costly modulo operation after every multiplication. When numbers are represented using Montgomery representation, this approach can substantially reduce the cost of multiplication in prime field arithmetic since the reduction operation is much less expensive than the modulo operation. The multiplication of two numbers in the Montgomery representation is referred to as *Montgomery multiplication*.

F. More Efficient Montgomery Reduction for Primes in the Form $2^{\ell}F - 1$ [27]

A prime number p is considered to be λ-*Montgomery-friendly* if it satisfies the condition $p \equiv \pm 1 \bmod 2^{\lambda \cdot \omega}$, where λ and ω are positive integers. Many cryptographic systems, such as SQISign [28], CGL hash function [29], and SIKE [1], use prime numbers of the form $p = 2^{\ell}F - 1$ as moduli. For any $\lambda \leq \ell/\omega$, these numbers are in the form $p \equiv -1 \bmod 2^{\lambda \cdot \omega}$, and therefore, are λ-Montgomery-friendly. For the simplicity of our algorithm and explanation, we assume that $p \equiv -1 \bmod 2^{\lambda \cdot \omega}$ or $p + 1$ is divisible by $2^{\lambda \cdot \omega}$ in the remaining part of this subsection.

Algorithm ExistingSpecificRedc: Montgomery reduction with λ-Montgomery-friendly modulus [27]

Input: $p < 2^{\omega n}$ is a prime number such that $p \equiv -1 \pmod{2^{\lambda \cdot \omega}}$, $r = 2^{\omega}$, $R = 2^{\omega n}$, $T < pR$, $m \leq \lambda$, $\lambda_0 \leftarrow \lfloor n/m \rfloor$, $\lambda_0' \leftarrow n \bmod m$

1 , and $M \leftarrow (p+1)/2^{\lambda \cdot \omega}$. **Output:** $TR^{-1} \bmod p$
2 $T^{(0)} \leftarrow T$
3 **for** $i \leftarrow 1$ **to** λ_0 **do**
4 $\quad Q \leftarrow T^{(i-1)} \bmod 2^{m \cdot \omega}$
5 $\quad T^{(i)} \leftarrow \left\lfloor (T^{(i-1)} + 2^{\lambda \cdot \omega} Q \cdot M)/2^{m \cdot \omega} \right\rfloor$
6 **end**
7 $T^{(\lambda_0+1)} \leftarrow T^{(\lambda_0)}$
8 **if** $\lambda_0' \neq 0$ **then**
9 $\quad Q \leftarrow T^{(\lambda_0)} \bmod 2^{\lambda_0' \cdot \omega}$
10 $\quad T^{(\lambda_0+1)} \leftarrow \left\lfloor (T^{(\lambda_0)} + 2^{\lambda \cdot \omega} Q \cdot M)/2^{\lambda_0' \cdot \omega} \right\rfloor$
11 **end**
12 **if** $T^{(\lambda_0+1)} \geq p$ **then**
13 $\quad T^{(\lambda_0+1)} \leftarrow T^{(\lambda_0+1)} - p$
14 **return** $T^{(\lambda_0+1)}$

[1]While SVE refers to mask registers as predicate registers, we have opted to maintain consistency in terminology within this paper. Consequently, predicate registers in SVE will also be referred to as mask registers to ensure straightforward understanding.

The Algorithm ExistingSpecificRedc suggests that for such values of p and $m \leq \lambda$, one can reduce the number of loop cycles in the Algorithm ExistingGenericRedc from n to $\lceil n/m \rceil$. However, each iteration in the Algorithm ExistingSpecificRedc has a more complex computation compared to the Algorithm ExistingGenericRedc. Specifically, while Line 4 of the Algorithm ExistingGenericRedc involves multiplying the n-limb integer p by the 1-limb integer Q, the corresponding operation in Line 4 of the Algorithm ExistingSpecificRedc multiplies the m-limb integer Q with the $(n-\lambda)$-limb integer M.

IV. Proposed SIMD Addition

We propose an addition and carry propagation for SIMD in Algorithm ProposedAdd. The algorithm uses some ideas from the carry-select adder [24], which is described in Section III-B. To eliminate the long dependency chain associated with carry selection, the algorithm simulates the addition of larger integers via the summation of smaller ones. As a result, all the carries are obtained in the general-purpose registers, thus bypassing the need for transferring data from mask registers to general-purpose ones as required by the approach discussed in Section III-C.

Algorithm ProposedAdd: Proposed SIMD addition

Input: integer A, B in radix-2^ω, where $A = \sum_{i=0}^{n-1} 2^{i\cdot\omega} A_i$, $B = \sum_{i=0}^{n-1} 2^{i\cdot\omega} B_i$, and $A_i, B_i < 2^\omega$

Output: $A + B$

/* $\mathsf{s}_i, \mathsf{t}_i$ and p_i are 8-bit integer. A_i, B_i, D_i and G_i are ω-bit integer. c_i, m_i are 1-bit integer. */

1 $\langle D_i \rangle_{i=0}^{n-1} \leftarrow \text{SIMD\_ADD}\left(\langle A_i \rangle_{i=0}^{n-1}, \langle B_i \rangle_{i=0}^{n-1}\right)$

2 Use SIMD to calculate t_i and p_i as in Table II in parallel.

3 $(\mathsf{s}_{n-1}, \ldots, \mathsf{s}_0) \leftarrow (\mathsf{t}_{n-1}, \ldots, \mathsf{t}_1) + (\mathsf{p}_{n-1}, \ldots, \mathsf{p}_1)$ (We use the carry-propagation addition method here, but with far fewer limbs than in the addition of A and B.)

4 $\langle \mathsf{s}_i \rangle_{i=0}^{n-1} \leftarrow \text{SIMD\_SUB}\left(\langle \mathsf{s}_i \rangle_{i=0}^{n-1}, \langle \mathsf{t}_i \rangle_{i=0}^{n-1}\right)$

5 $\langle c_i \rangle_{i=0}^{n-1} \leftarrow \text{SIMD\_SUB}\left(\langle \mathsf{s}_i \rangle_{i=0}^{n-1}, \langle \mathsf{p}_i \rangle_{i=0}^{n-1}\right)$

6 $\langle D_i \rangle_{i=0}^{n-1} \leftarrow \text{SIMD\_ADD}\left(\langle D_i \rangle_{i=0}^{n-1}, \langle c_i \rangle_{i=0}^{n-1}\right)$

7 **return** $D = \sum_{i=0}^{n-1} 2^{i\cdot\omega} D_i$

Consider the addition of A_i and B_i. The addition can be classified into one of the following three cases. We refer to the case corresponding to the i-th lane as $L_i \in \{\boldsymbol{N}, \boldsymbol{P}, \boldsymbol{G}\}$.

- Case $L_i = \boldsymbol{N}$: The addition would **N**ot generate a carry, which implies $c_{i+1} = 0$, regardless of c_i.
- Case $L_i = \boldsymbol{P}$: The addition would **P**ropagate carry from previous limb, which implies $c_{i+1} = c_i$.
- Case $L_i = \boldsymbol{G}$: The addition would **G**enerate a carry, which implies $c_{i+1} = 1$, regardless of c_i.

Even without knowing the operands A, B, the carries for all limbs can be determined by the value of L_i. For example, when $(L_2, L_1, L_0) = (\boldsymbol{N}, \boldsymbol{P}, \boldsymbol{G})$, the carries $(c_3, c_2, c_1, c_0) = (0, 1, 1, 0)$ because

- $L_0 = \boldsymbol{G}$ would give $c_1 = 1$,
- $L_1 = \boldsymbol{P}$ would propagate c_1 to c_2 and give $c_2 = c_1 = 1$,
- $L_2 = \boldsymbol{N}$ would give $c_3 = 0$.

Algorithm ProposedAdd employs this principle to transform a larger addition $A + B = (A_{n-1}, \ldots, A_0) + (B_{n-1}, \ldots, B_0)$ into a smaller one $\mathsf{t} + \mathsf{p} = (\mathsf{t}_{n-1}, \ldots, \mathsf{t}_0) + (\mathsf{p}_{n-1}, \ldots, \mathsf{p}_0)$. Assume that the sum of t_i and p_i corresponds to the case L'_i. The values of t_i and p_i are determined such that $L'_i = L_i$. This guarantees that the carry-overs from adding t_i and p_i align with those from adding A_i and B_i.

Table II illustrates the process of converting a large addition to a small one. Each line in the "Algorithm" column of Table II corresponds to an assembly instruction. The first two rows of the table are similar, except for the carry-checking approach and the choice of p_i.

We opted for 8-bit integers for p_i and t_i because it is efficient to transfer data across 8-bit lanes in AVX-512 and SVE. While it is feasible to convert each limb to a 1-bit addition, we have not discovered an efficient method for such a conversion in SVE. Since the addition in Line 5 needs a cross-lane operation, we made p_i independent of A_i and B_i. Consequently, p_i can be considered a constant and can be loaded from memory. We can show the correctness of the algorithm by the following lemmas and theorem.

Lemma 1. *The additions of t_i and p_i obtained from Table II falls into the same case as the additions of A_i and B_i.*

Proof. We demonstrate that the additions of t_i and p_i align with the same case as A_i and B_i, as shown in the third, fourth, fifth, and sixth columns of Table II. These columns represent the outcomes derived from each computation step for every instruction set and radix and for every $L_i \in \{\mathbf{N}, \mathbf{P}, \mathbf{G}\}$. □

Lemma 2. *The carry c_i obtained at Line 5 of Algorithm ProposedGenericRedc is same as the carry c_i in the addition of A_i and B_i*

Proof. By Lemma 1, the addition at Line 3 of the algorithm shares the same case as A_i and B_i. The carry-over resulting from adding t_{i-1} and p_{i-1} to the sum of t_i and p_i matches the carry-over from adding A_{i-1} and B_{i-1} to the sum of A_i and B_i. This carry-over is c_i. This implies that $\mathsf{s}_i = (\mathsf{t}_i + \mathsf{p}_i + c_i) \bmod 2^8$. The value of c_i can be computed by taking $\mathsf{s}_i - \mathsf{t}_i - \mathsf{p}_i$, as indicated at Lines 4-5 of the algorithm. □

Theorem 1. *Algorithm ProposedAdd calculates the addition results of A and B.*

Proof. By Lemma 2, the desirable value of c_i is obtained at Line 5 of the algorithm. Then, $D_i = (D_i + c_i) \mod 2^\omega = (A_i + B_i + c_i) \mod 2^\omega$, which is the desirable result, at the i-th limb at Line 6. □

The concepts of Algorithm ProposedAdd are demonstrated in the subsequent example:

Example 1. *Consider an example where $\omega = 16$, $n = 4$, $A_0 = 60000$, $B_0 = 5536$, $A_1 = 50000$, $B_1 = 15535$, $A_2 =$*

10000, $B_2 = 10000$, $A_3 = 20000$, *and* $B_3 = 20000$. *Assume that the numbers are represented using the native radix.*

The sum of A_0 *and* B_0 *invariably generates a carry. In contrast, the sum of* A_1 *and* B_1 *propagates a carry, while the sum of* A_2 *and* B_2 *does not generate a carry. Therefore,* (L_2, L_1, L_0) *is* $(\boldsymbol{N}, \boldsymbol{P}, \boldsymbol{G})$. *Applying the SIMD_ADD function from Line 1 of Algorithm ProposedAdd yields* $(D_3, D_2, D_1, D_0) = (40000, 20000, 65535, 0)$.

Using the native radix as a basis, the values of t_i *and* p_i *are computed according to the first row in Table II. This results in* $(G_3, G_2, G_1, G_0) = (5, 5, 16, 0)$, $(m_3, m_2, m_1, m_0) = (0, 0, 0, 1)$, *and* $(\mathsf{t}_3, \mathsf{t}_2, \mathsf{t}_1, \mathsf{t}_0) = (5, 5, 16, 17)$.

Since $(\mathsf{p}_3, \mathsf{p}_2, \mathsf{p}_1, \mathsf{p}_0) = (239, 239, 239, 239)$, *the summation performed at Line 3 produces the outcome* $(\mathsf{s}_3, \mathsf{s}_2, \mathsf{s}_1, \mathsf{s}_0) = (196, 197, 0, 0)$. *After applying the SIMD_SUB function at Lines 4-5,* $(c_3, c_2, c_1, c_0) = (0, 1, 1, 0)$. *This represents the expected carry for this addition. The final result derived from Line 6 is* $(D_3, D_2, D_1, D_0) = (40000, 20001, 0, 0)$, *which aligns with the expectations.*

V. Proposed SIMD Montgomery Reduction

A. Optimization for General Prime Numbers

Algorithm ExistingGenericRedc can be directly implemented with SIMD. However, Line 3 of the algorithm incurs significant computation cost due to its reliance on a cross-lane broadcast operation and a costly multiplication instruction, as documented in Table I. Additionally, Line 4 in the algorithm is dependent on the completion of Line 3, and Line 3 must wait for the result of Line 4 in the previous iteration. These loop dependencies hinder the CPU pipeline's throughput, which is not a significant issue when using general instruction sets, where Line 4's computation time dominates. However, on SIMD, the computation time of Line 4 is much shorter, while Line 3's cost increases, making it a significant concern. To address this problem, we have devised Algorithm ProposedGenericRedc.

The function SIMD_MUL_nx1, found at Line 2 of Algorithm ProposedGenericRedc, computes the multiplication result of the n-limb integer M_i and the one-limb integer T_i for all $1 \le i \le n-2$, deploying SIMD instructions. Considering the n limbs of M_i as $M_{i,0}, \ldots, M_{i,n-1}$, the multiplication outcome can be calculated via the following steps: 1) First, calculate $\langle U_j \rangle_{j=0}^{n-1} = \text{SIMD\_MUL\_H}\left(\langle M_{i,j} \rangle_{j=0}^{n-1}, \langle Z_j \rangle_{j=0}^{n-1}\right)$ when $Z_j = T_i$ for all $0 \le j \le n-1$; 2) Calculate $\langle Y_j \rangle_{j=0}^{n-1} = \text{SIMD\_MUL\_L}\left(\langle M_{i,j} \rangle_{j=0}^{n-1}, \langle Z_j \rangle_{j=0}^{n-1}\right)$ when $Z_j = T_i$ for all $0 \le j \le n-1$; 3) Calculate $H_i = \sum_{j=0}^{n-1} U_j 2^{\omega(j+1)} + \sum_{j=0}^{n-1} Y_j 2^{\omega j}$ using the Algorithm ProposedAdd in Section IV.

To show the correctness of Algorithm ProposedGenericRedc, we consider the following lemmas and theorem:

Lemma 3. *REDC*$(T) \equiv T^{(n-2)} r^{-2} \pmod{p}$.

Algorithm ProposedGenericRedc: Proposed SIMD generic Montgomery reduction

Input: $T < pR$, $p < 2^{\omega n}$, $r = 2^{\omega}$, $R = 2^{\omega n}$, $n > 2$, $M_i \equiv r^{-n+i+1} \pmod{p}$ for $1 \le i \le n-2$, $p' \leftarrow \frac{rr^{-1}-1}{p}$

Output: $TR^{-1} \bmod p$

1 Let T_i be $\lfloor T/r^{i-1} \rfloor \bmod r$, i.e. $T = (T_{2n}, \ldots, T_1)$
2 $\langle H_i \rangle_{i=1}^{n-2} \leftarrow \text{SIMD\_MUL\_nx1}\left(\langle M_i \rangle_{i=1}^{n-2}, \langle T_i \rangle_{i=1}^{n-2}\right)$
3 $T^{(n-2)} \leftarrow \lfloor T/r^{n-2} \rfloor + \sum_{i=1}^{n-2} H_i$
4 **for** $i \leftarrow n-1$ **to** n **do**
5 $\quad Q \leftarrow T^{(i-1)} p' \bmod r$
6 $\quad T^{(i)} \leftarrow (T^{(i-1)} + Qp)/r$
7 **end**
8 **if** $T^{(n)} > p$ **then**
9 $\quad T^{(n)} \leftarrow T^{(n)} - p$
10 **if** $T^{(n)} > p$ **then**
11 $\quad T^{(n)} \leftarrow T^{(n)} - p$
12 **return** $T^{(n)}$

Proof. We can derive that:

$$\begin{aligned} \textit{REDC}(T) &\equiv TR^{-1} \\ &\equiv \left(\lfloor T/r^{n-2} \rfloor \cdot r^{n-2} + \sum_{i=1}^{n-2} T_i r^{i-1}\right) r^{-n} \\ &\equiv \left(\lfloor T/r^{n-2} \rfloor + \sum_{i=1}^{n-2} T_i r^{-n+i+1}\right) r^{-2} \\ &\equiv \left(\lfloor T/r^{n-2} \rfloor + \sum_{i=1}^{n-2} T_i M_i\right) r^{-2} \\ &\equiv \left(\lfloor T/r^{n-2} \rfloor + \sum_{i=1}^{n-2} H_i\right) r^{-2} \\ &\equiv T^{(n-2)} r^{-2} \pmod{p} \end{aligned} \tag{1}$$

□

Lemma 4. *The value of* $T^{(n)}$ *computed at Line 7 of Algorithm ProposedGenericRedc is such that* $T^{(n)} \equiv REDC(T) \pmod{p}$.

Proof. It is straightforward from the definitions that $T^{(n-2)}$ can be obtained at Line 3. Then, Lines 4-7 are used for calculating *redc* $\left(T^{(n-2)}, 2\omega\right)$. By Lemma 3, $T^{(n)} \equiv$ *redc* $\left(T^{(n-2)}, 2\omega\right) \equiv T^{(n-2)} r^{-2} \equiv REDC(T) \pmod{p}$. □

Lemma 5. *The value of* $T^{(n)}$ *computed at Line 7 of Algorithm ProposedGenericRedc is such that* $T^{(n)} < 3p$.

Proof. From Line 3 of Algorithm ProposedGenericRedc, $T^{(n-2)} = \lfloor T/r^{n-2} \rfloor + \sum_{i=1}^{n-2} H_i < r^2 p + (n-2)rp$. Then, $T^{(n-1)} = \frac{T^{(n-2)} + Qp}{r} < \frac{r^2 p + (n-2)rp + (r-1)p}{r} < 2rp$. In addition, $T^{(n)} = \frac{T^{(n-1)} + Qp}{r} < \frac{2rp + (r-1)p}{r} < 3p$. □

TABLE II: Algorithms for operands of 8-bit additions, which are capable of simulating ω-bit additions along with the outcomes

Radix	**Architecture**		**Operand 1** (t_i)			**Operand 2** (p_i)
		Algorithm	$L_i = \boldsymbol{N}$	$L_i = \boldsymbol{P}$	$L_i = \boldsymbol{G}$	Algorithm
Native Radix	AVX-512 SVE	$G_i \leftarrow \mathrm{popcnt}(D_i)$	$0 \le G_i < \omega$	$G_i = \omega$	$0 \le G_i < \omega$	$\mathsf{p}_i \leftarrow 255 - \omega$
		$m_i \leftarrow (D_i < A_i)$	$m_i = 0$	$m_i = 0$	$m_i = 1$	
		$\mathsf{t}_i \leftarrow \mathrm{mADD}(G_i, \omega + 1, m_i)$	$\mathsf{t}_i < \omega$	$\mathsf{t}_i = \omega$	$\mathsf{t}_i > \omega$	
Reduced Radix-2^k	AVX-512	$G_i \leftarrow \mathrm{popcnt}(D_i)$	$0 \le G_i < k$	$G_i = k$	$0 < G_i \le k$	$\mathsf{p}_i \leftarrow 255 - k$
		$m_i \leftarrow (D_i \ge 2^k)$	$m_i = 0$	$m_i = 0$	$m_i = 1$	
		$\mathsf{t}_i \leftarrow \mathrm{mADD}(G_i, k, m_i)$	$\mathsf{t}_i < k$	$\mathsf{t}_i = k$	$\mathsf{t}_i > k$	
Reduced Radix-2^k	SVE	$G_i \leftarrow \mathrm{SADD}(D_i, 2^\omega - 2^k - 1)$	$G_i < 2^\omega - 2$	$G_i = 2^\omega - 2$	$G_i = 2^\omega - 1$	$\mathsf{p}_i \leftarrow 254$
		$\mathsf{t}_i \leftarrow \mathrm{SSUB}(G_i, 2^\omega - 3)$	$\mathsf{t}_i = 0$	$\mathsf{t}_i = 1$	$\mathsf{t}_i = 2$	

Note: $\mathrm{popcnt}(D_i)$ =Number of 1 in binary representation of D_i.
For this function, we utilize the VPOPCNTQ instruction in AVX-512 and the CNT instruction in SVE.
$\mathrm{mADD}(G_i, \omega + 1, m_i) = G_i + \omega + 1$ ***if*** m_i ***else*** G_i
For this function, we utilize the VPADDQ instruction in AVX-512 and the ADD instruction in SVE.
$\mathrm{SADD}(x, y) = \min(x + y, 2^\omega - 1)$ For this function, we utilize the UQADD instruction in SVE.
$\mathrm{SSUB}(x, y) = \max(x - y, 0)$ For this function, we utilize the UQSUB instruction in SVE.

Theorem 2. *The value of $T^{(n)}$ computed at Line 12 of Algorithm ProposedGenericRedc is such that $T^{(n)} = TR^{-1} \bmod p$.*

Proof. It is known from Lemma 4 and Lines 8-11 that $T^{(n)} \equiv TR^{-1} \pmod p$. Also, because $T^{(n)}$ is less than $3p$ at Line 7, the value is less than p by the final reduction at Lines 8-11. Hence, $T^{(n)} = TR^{-1} \mod p$. □

When $T < pR - (n-2)r^{n-1}p$, it is worth noting that similar reasoning can be employed to demonstrate that the value of $T^{(n)}$ obtained at Line 7 is less than $2p$. Therefore, the instruction at Lines 10-11 can be skipped. This condition holds for all the reductions in SIKE, CSIDH, and CTIDH, and hence, there is no need to execute the instruction at Lines 10-11 while implementing these cryptographic algorithms.

Algorithm ProposedGenericRedc requires additional memory reads, but this is not a problem as SIMD is efficient at memory reads and the load/write units in CPUs are not frequently used when computing Montgomery reduction.

Although M_i is considered a pre-calculated constant, no information about p is required to calculate it. Instead, it can be computed as $M_i = \mathit{REDC}(r^{i-1})$ using Algorithm ExistingGenericRedc. Therefore, the condition to use Algorithm ProposedGenericRedc is the same as that for Algorithm ExistingGenericRedc, and it can be applied to field arithmetic where the prime is variable, such as in RSA.

The concepts of Algorithm ProposedGenericRedc are illustrated in the subsequent example.

Example 2. *Set $\omega = 4$, $r = 2^4 = 16$, $n = 4$, $R = 2^{16} = 65536$, and $p = 62207$. Given that $(M_1, M_2) = (3888, 243)$, and p' equals to 1.*

Suppose $T = 100000000$, which leads to $(T_8, \ldots, T_1) = (0, 5, 15, 5, 14, 1, 0, 0)$ as per Line 1 of Algorithm ProposedGenericRedc. By Line 2, $H_1 = H_2 = 0$, and Line 3 gives $T^{(2)} = T/256 + 0 = 390625$.

In the loop's first iteration from Lines 4-7, $Q = 390625 \bmod 16 = 1$ and $T^{(3)} = (390625 + 1 \times 62207)/16 = 28302$. By the second iteration, $Q = 28302 \bmod 16 = 14$ and $T^{(4)} = (28302 + 14 \times 62207)/16 = 56200$.

As $T^{(4)} = 56200$ is already less than p, the final reductions at Lines 8-11 are not needed. Therefore, the result of this Montgomery reduction is 56200.

B. Optimization for Primes in the Form $2^\ell F - 1$

We observe that the prime number p used in several cryptographic systems such as SIKE [1] or SQISign [28] can be written in the form of $p = 2^\ell F - 1$ when F and 2^ℓ are large integers. In this subsection, we aim to give an improvement over Algorithm ExistingSpecificRedc for the case when $2^\ell \approx F$. In particular, assume that $\ell < \lceil n/2 \rceil \omega < \ell + \omega$.

We are using the prime number $p_{503} = 2^{250}3^{159} - 1$ to demonstrate our idea, which can also be applied to other prime numbers. The prime number is often used in the SIKE cryptographic system [1]. When $\omega = 64$, p_{503} can be used as a 3-Montgomery-friendly modulus with $n = 8$ limbs for. In Algorithm ExistingSpecificRedc, the PQCrypto-SIDH selected $m = 2$ for p_{503} because it is the largest divisor of n that is not greater than $\lambda = 3$. This means that four multiplications between Q (two limbs) and M (four limbs) will be performed at lines 4-7 of Algorithm ExistingSpecificRedc.

We aim to use the Karatsuba algorithm [30] to speed up the multiplications. The Karatsuba algorithm allows us to reduce the complexity of multiplying two multiple-limb numbers, making it an attractive option for speeding up multiplications. It is discussed in [5] that the algorithm can be efficiently implemented in SIMD. However, the Karatsuba algorithm would be less effective if operands do not have the same number of limbs. Therefore, the Karatsuba algorithm cannot be used in the situation stated in the previous paragraph as the size of Q and F are not the same.

Our goal is to modify Algorithm ExistingSpecificRedc by increasing the value of Q to a larger number such as one with $\lceil n/2 \rceil \omega$ bits. For example, when dealing with the prime number p_{503}, instead of using a 128-bit value for Q in Algorithm ExistingSpecificRedc, it is more preferable to use a 256-bit value for Q. It is not possible to achieve this by just simply replacing the value $2^{m\omega}$ in Lines 6-7 of Algorithm ExistingSpecificRedc with a larger number. Merely

replacing the component at the algorithm would result in an incorrect reduction output.

Algorithm ProposedSpecificRedc: Proposed Montgomery reduction with λ-Montgomery-friendly modulus

Input: $T < pR, p = 2^\ell F - 1$ such that $\ell < \lceil n/2 \rceil \omega < \ell + \omega, R = 2^{\omega n} > p$
Output: $TR^{-1} \bmod p$

1 $T^{(0)} \leftarrow T$
2 $t^{(1)} \leftarrow [T^{(0)} F \bmod 2^\omega] \cdot 2^\ell$
3 $Q^{(1)} \leftarrow (T^{(0)} + t^{(1)}) \bmod 2^{\lceil n/2 \rceil \omega}$
4 $T^{(1)} \leftarrow \lfloor (T^{(0)} + 2^\ell \cdot \text{Karatsuba}(Q^{(1)}, F))/2^{\lceil n/2 \rceil \omega} \rfloor$
5 **if** n *is odd number* **then**
6 $\quad Q^{(2)} \leftarrow T^{(1)} \bmod 2^{\lfloor n/2 \rfloor \omega}$
7 $\quad T^{(2)} \leftarrow \lfloor (T^{(1)} + 2^\ell \cdot \text{Karatsuba}(Q^{(2)}, F))/2^{\lfloor n/2 \rfloor \omega} \rfloor$
8 **else**
9 $\quad t^{(2)} \leftarrow [T^{(1)} F \bmod 2^\omega] \cdot 2^\ell$
10 $\quad Q^{(2)} \leftarrow (T^{(1)} + t) \bmod 2^{\lceil n/2 \rceil \omega}$
11 $\quad T^{(2)} \leftarrow \lfloor (T^{(1)} + 2^\ell \cdot \text{Karatsuba}(Q^{(2)}, F))/2^{\lceil n/2 \rceil \omega} \rfloor$
12 **end**
13 **if** $T^{(2)} > p$ **then**
14 $\quad T^{(2)} \leftarrow T^{(2)} - p$
15 **return** $T^{(2)}$

In order to yield a valid result for the reduction process, the variables $t^{(1)}$ and $t^{(2)}$ are introduced at the second and ninth line of Algorithm ProposedSpecificRedc, respectively. Through the subsequent lemmas, we establish that by applying these corrections, Algorithm ProposedSpecificRedc always provides a correct output. First, let us consider the variable $t^{(1)}$ at Line 2 and the variable $t^{(2)}$ at Line 9 of Algorithm ProposedSpecificRedc.

Lemma 6. *Let $r = 2^{\lceil n/2 \rceil \omega}$. For $i \in \{1, 2\}$, $t^{(i)} \equiv T^{(i-1)} F 2^\ell \pmod r$.*

Proof. This lemma can be proved as follows:

$$\begin{aligned} t^{(i)} \bmod r &= (T^{(i-1)} F \mod 2^\omega) \cdot 2^\ell \mod 2^{\lceil n/2 \rceil \omega} \\ &= [(T^{(i-1)} F \mod 2^\omega) \mod 2^{\lceil n/2 \rceil \omega - \ell}] \cdot 2^\ell \\ &= (T^{(i-1)} F \mod 2^{\lceil n/2 \rceil \omega - \ell}) \cdot 2^\ell \\ &= T^{(i-1)} F 2^\ell \mod 2^{\lceil n/2 \rceil \omega}. \end{aligned}$$

Thus, $t^{(i)} \equiv T^{(i-1)} F 2^\ell \pmod r$ □

In the following lemma, let us consider the variable $Q^{(1)}$ at Line 3 and the variable $Q^{(2)}$ at Line 10 of Algorithm ProposedSpecificRedc.

Lemma 7. *Let $r = 2^{\lceil n/2 \rceil \omega}$. For $i \in \{1, 2\}$, consider $Q^{(i)}$ calculated at Line 3 and Line 10 of Algorithm ProposedSpecificRedc. $T^{(i-1)} + pQ^{(i)}$ is divisible by r.*

Proof. From Lemma 6:

$$\begin{aligned} Q^{(i)} &\equiv T^{(i-1)} + t^{(i)} \\ &\equiv T^{(i-1)} + 2^\ell F T^{(i-1)} \\ &\equiv T^{(i-1)} (2^\ell F + 1) \\ &\equiv T^{(i-1)} (p + 2) \pmod r. \end{aligned}$$

Because $\lceil n/2 \rceil \omega < \ell + \omega \leq 2\ell$, we notice that $2^{2\ell}$ is divisible by $r = 2^{\lceil n/2 \rceil \omega}$. Hence,

$$\begin{aligned} T^{(i-1)} + pQ^{(i)} &\equiv T^{(i-1)} + p(p+2) T^{(i-1)} \\ &\equiv (p^2 + 2p + 1) T^{(i-1)} \\ &\equiv (p+1)^2 T^{(i-1)} \\ &\equiv 2^{2\ell} F^2 T^{(i-1)} \\ &\equiv 0 \pmod r. \end{aligned}$$

□

Lemma 8. *The value $T^{(1)}$ calculated at Line 4 is $redc(T, \lceil n/2 \rceil \omega)$. The value $T^{(2)}$ calculated at Line 7 and Line 11 is $redc(T, n\omega)$.*

Proof. Let $r = 2^{\lceil n/2 \rceil \omega}$. From Lemma 7:

$$\begin{aligned} \left\lfloor \frac{T^{(0)} + 2^\ell Q^{(1)} \cdot F}{r} \right\rfloor &\equiv \left\lfloor \frac{T^{(0)} + Q^{(1)}(p+1)}{r} \right\rfloor \\ &\equiv \left\lfloor \frac{T^{(0)} + Q^{(1)} p}{r} + \frac{Q^{(1)}}{r} \right\rfloor \\ &\equiv \frac{T^{(0)} + Q^{(1)} p}{r} + \left\lfloor \frac{Q^{(1)}}{r} \right\rfloor \\ &\equiv \frac{T^{(0)} + Q^{(1)} p}{r} \\ &\equiv T^{(0)} r^{-1} \pmod p \end{aligned} \tag{2}$$

Hence, $T^{(1)}$ is $redc(T^{(0)}, \lceil n/2 \rceil \omega)$.

Next, let us consider the case when n is odd. In that case, Lines 6-7 are used to calculate $T^{(2)}$. Let $r' = 2^{\lfloor n/2 \rfloor \omega}$. Then:

$$\begin{aligned} \left\lfloor \frac{T^{(1)} + 2^\ell Q^{(2)} \cdot F}{r'} \right\rfloor &\equiv \left\lfloor \frac{T^{(1)} + Q^{(2)}(p+1)}{r'} \right\rfloor \\ &\equiv \left\lfloor \frac{T^{(1)} + Q^{(2)} p}{r'} + \frac{Q^{(2)}}{r'} \right\rfloor \\ &\equiv \frac{T^{(1)} + Q^{(2)} p}{r'} + \left\lfloor \frac{Q^{(2)}}{r'} \right\rfloor \\ &\equiv \frac{T^{(1)} + Q^{(2)} p}{r'} \\ &\equiv T^{(1)} (r')^{-1} \pmod p. \end{aligned} \tag{3}$$

Hence, $T^{(2)}$ is $redc(T^{(1)}, \lfloor n/2 \rfloor \omega)$ and, since $\lfloor n/2 \rfloor \omega + \lceil n/2 \rceil \omega = n\omega$, $T^{(2)}$ is $redc(T^{(0)}, n\omega)$.

When n is even, the same argument as in (2) can be used to show that $T^{(2)}$ is $redc(T^{(1)}, \lceil n/2 \rceil \omega)$ and, since $\lceil n/2 \rceil \omega + \lceil n/2 \rceil \omega = n\omega$, $T^{(2)}$ is $redc(T^{(0)}, n\omega)$. □

We are now ready to prove the correctness of Algorithm ProposedSpecificRedc in the following theorem.

Theorem 3. *The value $T^{(2)}$ returned at Line 15 of Algorithm ProposedSpecificRedc is $TR^{-1} \bmod p$.*

Proof. By Lemma 8, it is left to show that $T^{(2)}$ at Line 15 is smaller than p. By the same argument as in (2), $T^{(1)} = \frac{T^{(0)}+pQ^{(1)}}{2^{\lceil n/2 \rceil \omega}} < \frac{pR+p\cdot 2^{\lceil n/2 \rceil \omega}}{2^{\lceil n/2 \rceil \omega}} = p2^{\lfloor n/2 \rfloor \omega} + p$. Then, by the same argument as (2) and (3), at Line 13, $T^{(2)} = \frac{T^{(1)}+p\cdot Q^{(2)}}{2^{\lfloor n/2 \rfloor \omega}} < \frac{p\cdot 2^{\lfloor n/2 \rfloor \omega}+p+p(2^{\lfloor n/2 \rfloor \omega}-1)}{2^{\lfloor n/2 \rfloor \omega}} = 2p$. By the final reduction at Lines 13-14, the result at Line 15 is smaller than p. □

Calculating the value of $t^{(i)}$ at Lines 2 and 9 is not resource-intensive. The computation only necessitates the least significant limb, which is derived from the product of the least significant limb of $T^{(0)}$ and that of F. Moreover, the least significant limb of $Q^{(i)} \cdot F$ at Lines 4 and 11 is also equal to $T^{(i-1)} \cdot F \bmod 2^{\omega}$. Consequently, the result of $t^{(i)}$ can be reused at those points. This means that the introduction of $t^{(i)}$ into Algorithm ProposedSpecificRedc incurs no additional multiplication cost.

The bottleneck of Algorithm ProposedSpecificRedc lies in the multiplications at Lines 4, 7, and 11. The Karatsuba method is employed to perform these multiplications. To demonstrate the efficiency of the proposed algorithm compared to Algorithm ExistingSpecificRedc, we consider the multiplication of 512-bit integers. As previously discussed, the previous method requires four multiplications between 128-bit integers (2 limbs) and 256-bit integers (4 limbs), with each multiplication costing eight multiplication instructions. Therefore, a total of 32 multiplication instructions are needed. In contrast, the proposed method enables the reduction to be performed by two multiplications between 256-bit integers (4 limbs). If one-level Karatsuba method is used for these multiplications, each 256-bit (4 limbs) integer multiplication can be completed using 12 instructions, resulting in a total of 24 instructions. This reduces the number of SIMD multiplication instructions from 32 to 24, resulting in greater efficiency.

Primes of the form $2^{\ell}F - 1$ are commonly employed in isogeny-based cryptography. However, currently, only a few algorithms such as SIKE and SIDH use primes where $\ell < \lceil n/2 \rceil \omega < \ell + \omega$. One other algorithm that uses such primes is Curve448, an elliptic curve cryptography algorithm which utilizes $p_{448} = 2^{448} - 2^{224} - 1$. Unfortunately, a unique property of p_{448} enables calculation of $\mathbb{F}_{p^{448}}$ without the need for Montgomery reduction. Despite the current circumstances, the employment of p_{448} in elliptic curve cryptography signals the likelihood of more elliptic-curve based protocols using primes that can be represented as $2^{\ell}F - 1$ in the future.

The concepts of Algorithm ProposedSpecificRedc are illustrated in the subsequent example. The inputs of the following example is same as in Example 2.

Example 3. *Set $\omega = 4$, $n = 4$, $R = 2^{16} = 65536$, $p = 62207 = 2^8 \cdot 243 - 1$, and $T = 100000000$. The value of p allows us to deduce that $\ell = 8$ and $F = 243$.*

At Line 3 of Algorithm ProposedSpecificRedc, the calculation is made that $t^{(1)} = [(100000000 \times 243) \mod 16] \times 2^8 = 0$. Subsequent calculations yield $Q^{(1)} = 100000000 \bmod 256 = 0$ and $T^{(1)} = \lfloor 100000000/2^8 \rfloor = 390625$. Given that n is an even number, Lines 9-11 of the algorithm are used in this example. This results in $t^{(2)} = [(390625 \times 243) \bmod 16] \times 2^8 = 768$. Consequently, the results obtained are $Q^{(2)} = (390625 + 768) \bmod 256 = 225$, and $T^{(2)} = \lfloor (390625 + 256 \times 243 \times 225)/256 \rfloor = 56200$, which accord with the output in Example 2.

TABLE III: Number of cycles of proposed SVE implementation for CTIDH-511 on A64FX

Operation	A64 (cycles)	SVE (cycles)	Speedup
Addition	16.07	13.72	1.17x
Montgomery multiplication	406.98	258.96	1.57x
Scalar multiplication	39.98	18.68	2.14x
CTIDH Action	316,308,640	242,948,411	1.30x

VI. Case Study and Experimental Results

This section will cover the implementation of prime field arithmetic for various cryptography algorithms and architectures. The benchmarks for ARM architecture are conducted on Wisteria/BDEC-01 (Odyssey) with A64FX@2.20GHz, while those for x64 architecture are run on Intel i7-1165G7 clocked at 2701 MHz. The benchmark for field arithmetic involves unrolling loops 80 times due to the calculation time being too short. The results are the average of 800,000 repetitions for field operations and 1,000 repetitions for the entire cryptography algorithm. We utilize the compiler's intrinsic to execute the calculation for AVX-512, whereas assembly code is employed for the tests on A64, x64, and SVE architectures.

A. CTIDH with SVE

Our implementation of CTIDH utilizes SVE with native radix. While it is not ideal to assume the vector length of SVE, we have assumed a 512-bit implementation for optimal performance[2].

Although Algorithm ProposedAdd has significantly improved the performance of addition on SVE for $\mathbb{F}_p$ addition and subtraction, it is not significantly faster than A64 implementation. Therefore, we perform $\mathbb{F}_p$ addition and subtraction using A64. For $\mathbb{F}_p$ multiplication, we utilize Algorithms ProposedAdd and ProposedGenericRedc. The multiplication is carried out using an operand-scan technique, and the reduction step is implemented with Algorithm ProposedGenericRedc. We use radix-2^{56} for Lines 3-5 in Algorithm ProposedGenericRedc instead of native radix, allowing us to perform the reduction with only one partial reduction in Lines 8-11, rather than two. This conversion of radix can be accomplished with a single table lookup instructions, since both radices is a multiple of 8.

Since p_{511} does not satisfy the condition for lazy correction, we still need to correct the result after reduction. However, a complete correction requires the carries for $T^{(n)}$ and $T^{(n)} - p$

[2]Our CTIDH implementation is uploaded at https://github.com/splight793/ARMv8-CTIDH.

to be both propagated, which we want to avoid since carry propagation is still costly even with Algorithm ProposedAdd. We therefore only check for the most significant word of $T^{(n)}$ to determine whether to use $T^{(n)} - p$, and we postpone further correction until subtraction or equality testing becomes necessary. Given that subtraction and equality testing occur less frequently than multiplication, we believe that the tradeoff is worthwhile.

We evaluated the proposed method against the state-of-the-art CTIDH implementation proposed by Benegas et al. [3]. However, since there is no existing SVE assembly code for their implementation, we adapted the assembly code for CSIDH by Jalali et al. [16] using the concept outlined in the CTIDH paper, and employed that code as our benchmark. We also implement the code of our algorithms based on that assembly code.

As shown in Table III, in SVE, the proposed SIMD addition is 18% faster than the standard A64 addition. We attribute this mainly to the benefit of SVE memory read-write capabilities. While A64 addition is a memory-bound function, the proposed SIMD addition is a compute-bound function. However, we observed a slight decrease in performance when using the proposed addition for CTIDH operations. This could be due to the fact that the remaining CTIDH operations are primarily compute-bound functions, and using A64 addition enables a better distribution of work between the compute execution unit and memory access execution unit.

Algorithm ProposedGenericRedc outperforms Jalali et al.'s Montgomery multiplication implementation by 57%, and our scalar multiplication is faster than the current state-of-the-art implementation on A64 by 114%. The speedup of our algorithm for CTIDH action is 30%. Nonetheless, if our SIMD multiplication is utilized with a prime of 510 bits or less, the resulting speedup would be even greater.

B. CSIDH with AVX-512

This section will present the results obtained by utilizing AVX-512 with Algorithm ProposedAdd and Algorithm ProposedGenericRedc[3]. We will compare the proposed methods' running time with two existing implementations, namely:

1) The state-of-the-art implementation of constant-time CSIDH on x64 by Cervantes-Vázquez et al. [18], which employs the code from Castryck et al. [2] for $\mathbb{F}_p$ arithmetic, and is based on the constant-time OAYT-style CSIDH implementation.
2) The state-of-the-art implementation of constant-time CSIDH in SIMD by Cheng et al. [5], which is a constant-time OAYT-style CSIDH implementation on AVX-512.

It is worth mentioning that Cheng et al. [5] have also optimized $\mathbb{F}_{p_{511}}$ squaring in their work, whereas Cervantes-Vázquez et al. utilized the code for multiplication to perform squaring.

Cheng et al. [5] utilized a two-packed radix-2^{43} representation to implement $\mathbb{F}_{p_{511}}$ arithmetic. This implies that an $\mathbb{F}_{p_{511}}$ element is represented with 12 limbs, and three vector registers are utilized to store the two operands. The storage order of each limb is defined as follows:

$$\boldsymbol{V} = \langle a, b \rangle = \left\{ \begin{matrix} [a_0, a_3, a_6, a_9\ , b_0, b_3, b_6, b_9\ ] \\ [a_1, a_2, a_7, a_{10}, b_1, b_4, b_7, b_{10}] \\ [a_2, a_5, a_8, a_{11}, b_2, b_5, b_8, b_{11}] \end{matrix} \right\} \quad (4)$$

Our code for Algorithm ProposedGenericRedc is based on Cheng et al.'s code [5]. In addition, we have performed some extra calculations to evaluate the performance of Algorithm ProposedAdd. Table IV presents the benchmark results obtained from these calculations.

The top two rows of the table present the runtime performance of Algorithm ProposedAdd compared to the carry propagate addition implemented in x64 and AVX-512 when calculating one instance. The results demonstrate that the proposed algorithm's runtime on AVX-512 is significantly shorter than x64 and naive AVX-512 carry propagation.

The third row shows that naive AVX-512 carry propagation is still slower than x64 for two-packed calculation. We encountered some difficulties while implementing Algorithm ProposedAdd with the input defined in (4). We found that the following storage order is better for SIMD implementation.

$$\boldsymbol{V^*} = \langle a, b \rangle = \left\{ \begin{matrix} [a_0, b_0, a_3, b_3, a_6, b_6, a_9\ , b_9\ ] \\ [a_1, b_1, a_4, b_4, a_7, b_7, a_{10}, b_{10}] \\ [a_2, b_2, a_5, b_5, a_8, b_8, a_{11}, b_{11}] \end{matrix} \right\} \quad (5)$$

With this storage order, carries can be efficiently propagated using 128-bit SSE instructions without any expensive cross-lane operations. Row 3 has shown that it is very efficient to implement with input defined as (5) using SSE. Although the experiment in Row 3 has an input loaded from cache, it is also efficient for SSE to calculate this carry propagation with input stored in AVX-512 registers. AVX-512 supports an instruction VEXTRACTI64x2, which allows CPU to read any 128-bit aligned lanes in an AVX-512 registers to SSE registers. VEXTRACTI64x2 is a cross-lane instruction and have a CPI of 1 and latency of 3. Since these reads are independent to each other, SSE can handle input defined as (5) stored in AVX-512 register efficiently.

Performance results for Montgomery multiplication are presented in the fourth row of the table, indicating an improvement of around 10% over Cheng et al.'s implementation [5]. While the proposed algorithm reduces the number of 52-bit multiplications by 6%, we attribute the larger improvement in computation time because the proposed algorithm gives a more efficient CPU pipeline. The proposed algorithm can also enhance the performance of Montgomery squaring. Compared to the state-of-the-art algorithm, our implementation is 36% faster in running time. This larger difference again confirms that the proposed Algorithm ProposedGenericRedc have improved CPU pipeline.

Row 6 of the table demonstrates that the proposed Montgomery multiplication and squaring contribute to an improvement in the CSIDH algorithm's performance. Our implementa-

[3]Our CSIDH implementation is uploaded at https://github.com/splight793/AVX-CSIDH.

TABLE IV: Performance results of proposed AVX-512 implementation for CSIDH-511 on TigerLake

Operation	x64 Number of cycles	Previous AVX-512 Implementations Number of cycles	Speedup	Our AVX-512 Implementation Number of cycles	Speedup
Native Radix Addition 8 limbs	6.052 [18]	11.593	0.52x	4.589	1.32x
Radix-2^{43} Addition 8 limbs	6.838	7.672	0.89x	5.280	1.30x
Radix-2^{43} Carry Propagation 12 limbs, 2-packed	18.617*	23.493*[5]	0.79x	10.627**	1.75x
Montgomery Multiplication 2-packed	258.140 [18]	144.890 [5]	1.78x	131.066	1.97x
Montgomery Squaring 2-packed	262.103 [18]	141.696 [5]	1.85x	104.404	2.51x
CSIDH Action	132,051,574 [18]	83,099,556 [5]	1.59x	74,491,118	1.77x

* Input defined as (4)
**Input defined as (5). Naive carry propagation using SSE.

TABLE V: Performance results of proposed Reduction for SIKE in comparison with SIDHv3.5 on A64FX

Operation	SIDHv3.5 Time (ns)	new Reduction Time (ns)	Speedup
Reduction	196.84	156.48	1.26x
Keygen	36,749,182	35,204,915	1.04x
Encapsulation	60,642,713	56,449,034	1.07x
Decapsulation	65,017,001	60,901,455	1.07x

tion is 11% faster than Cheng et al.'s implementation and 77% faster than Cervantes-Vázquez et al.'s x64 implementation.

C. SIKE on ARM64

In this section, we present our benchmark results for Algorithm ProposedSpecificRedc using the *SIKEp503* protocol parameter[4]. This parameter set is widely recognized for its use in both SIKE and SIDH. This section takes into account the code from Microsoft's SIDHv3.5 [23], a recognized highly optimized implementation of an isogeny-based cryptosystem. Modifications are performed to replace SIDHv3.5's addition and reduction operations with the newly proposed algorithms. The comparative results showcasing the latency difference before and after these alterations in SIDHv3.5 are presented in Table V.

The proposed reduction method is 26% faster than the reduction method of SIDHv3.5, which is close to the theoretical improvement where Comba has 33% more multiplications than the one-level Karatsuba method. Regarding the overall performance of SIKE, our implementation is 7% faster than SIDHv3.5. As we optimized only the reduction method and left many calculations of SIKE unchanged (e.g., multiplication), this result aligns with expectation.

D. Comparison with the Kogge-Stone Vector Addition

We have not included a comparison between the Kogge-Stone vector addition [25] and the proposed addition algorithm in Tables III-V, primarily because the code provided in [25] pertains to additions with carry, whereas we focus on the addition without carry in these tables. Instead, we compare our proposed addition algorithm with previous works in this section.

We modify the addition algorithm detailed in [25] for execution with SVE instructions. The average cycle count for the addition with carry operation stands at 55.28 cycles. However, our algorithm operates at a reduced count of 42.90 cycles. This performance difference reveals that our algorithm delivers a 29% speed improvement in comparison to the state-of-the-art addition algorithm in SVE. A reason which gives the proposed algorithm a better efficiency is the fact that the algorithm fixes the value of second operation (p_i in Table I) to a constant.

We have additionally carried out experiments employing AVX-512 instructions. The average cycle count observed for our algorithm stands at 6.41 cycles, in comparison to the Kogge-Stone vector addition which has an average of 4.16 cycles. The fact that our algorithm lags behind the state-of-the-art algorithm can be attributed to the relatively insignificant cost of transferring from mask registers to general-purpose ones in AVX-512. Therefore, our approach does not contribute a significant improvement to the execution time.

E. Discussions

Our proposed addition and reduction algorithms have demonstrated their ability to enhance the performance of state-of-the-art SIMD algorithms in diverse scenarios. They have also shown the potential to boost the performance of post-quantum cryptographic protocols which rely on these operations. When compared to computations performed without SIMD (x64 or A64), the speed enhancements derived from our algorithm range from 1.17x to 2.51x.

One could conjecture an improvement closer to 8x, given that SIMD allows for 8 simultaneous additions, subtractions, and multiplications. However, as illustrated in Table I, the latency of SIMD instructions typically exceeds that of x64 or A64 instructions. The throughput (CPI) of AVX-512 is merely 2-4 times greater than that of x64. During our experimentation, we also observed that the improvement in SVE is even less

[4]Our SIDH and SIKE implementations are uploaded at https://github.com/splight793/PQCrypto-SIDH.

noticeable, indicating that the expected speed up is unlikely to be more than 2x. Additionally, SIMD requires the identical execution of eight parallel operations. Due to dependencies between operations, arranging them to fully harness the potential of SIMD is often impracticable. Considering these limitations, we believe that an improvement in the range of 1.17x to 2.51x is appreciable.

VII. Conclusions

This research demonstrates that SVE can significantly enhance prime field arithmetic and cryptosystems such as CTIDH. However, we have identified that the existing algorithms for large addition and Montgomery reduction are not efficient for SVE on A64FX, resulting in too many pipeline stalls. To address this issue, we propose new algorithms for addition and Montgomery reduction, which lead to a 30% speedup for CTIDH. These algorithms are not limited to SVE and have shown usefulness for AVX-512 as well. Additionally, we provide an algorithm for SIKE to improve its Montgomery reduction. Furthermore, we emphasize that addition should be implemented using general-purpose instructions or shorter SIMD to avoid the expensive cross-lane operation.

Acknowledgement

This work is partially supported by JSPS Grant-in-Aid for Transformative Research Areas A grant number JP21H05845, and also by JST SICORP Grant Number JPMJSC2208, Japan. The authors would like to thank anonymous reviewers and Taku Onodera for several useful comments and ideas which significantly improved this paper.

References

[1] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, D. Urbanik, G. Pereira, K. Karabina, and A. Hutchinson, "SIKE," National Institute of Standards and Technology, Tech. Rep., 2022, available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[2] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes, "CSIDH: An efficient post-quantum commutative group action," in *ASIACRYPT 2018, Part III*, ser. LNCS, T. Peyrin and S. Galbraith, Eds., vol. 11274. Springer, Heidelberg, Dec. 2018, pp. 395–427.

[3] G. Banegas, D. J. Bernstein, F. Campos, T. Chou, T. Lange, M. Meyer, B. Smith, and J. Sotáková, "CTIDH: faster constant-time CSIDH," *IACR TCHES*, vol. 2021, no. 4, pp. 351–387, 2021, https://tches.iacr.org/index.php/TCHES/article/view/9069.

[4] D. Kostic and S. Gueron, "Using the new vpmadd instructions for the new post quantum key encapsulation mechanism sike," in *ARITH 2019*, 2019, pp. 215–218.

[5] H. Cheng, G. Fotiadis, J. Großschädl, P. Y. A. Ryan, and P. B. Rønne, "Batching CSIDH group actions using AVX-512," *IACR TCHES*, vol. 2021, no. 4, pp. 618–649, 2021, https://tches.iacr.org/index.php/TCHES/article/view/9077.

[6] A. Jalali, R. Azarderakhsh, M. M. Kermani, M. Campagna, and D. Jao, "ARMv8 SIKE: Optimized supersingular isogeny key encapsulation on ARMv8 processors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, pp. 4209–4218, 11 2019.

[7] M. Anastasova, R. Azarderakhsh, and M. M. Kermani, "Time-optimal design of finite field arithmetic for SIKE on Cortex-M4," in *WISA 2022*, 2023, pp. 265–276.

[8] P. Ren, R. Suda, and V. Suppakitpaisarn, "Throughput-optimized implementation of isogeny-based cryptography on vectorized ARM SVE processor," in *CANDAR 2022*, 2022, pp. 165–171.

[9] T. Edamatsu and D. Takahashi, "Efficient large integer multiplication with ARM SVE instructions," in *AsiaHPC 2023*, 2023, pp. 9–17.

[10] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[11] H. Seo, Z. Liu, Y. Nogami, J. Choi, and H. Kim, "Hybrid montgomery reduction," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 3, pp. 1–13, 2016.

[12] T. Edamatsu and D. Takahashi, "Accelerating large integer multiplication using Intel AVX-512IFMA," in *ICA3PP 2019*, 2020, pp. 60–74.

[13] A. Fog, "Instruction tables: List of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs (2012)," 2022. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf

[14] *A64FX Microarchitecture Manual*, Fujitsu, 2022, revision 1.8.1.

[15] A. Hutchinson, J. T. LeGrow, B. Koziel, and R. Azarderakhsh, "Further optimizations of CSIDH: A systematic approach to efficient strategies, permutations, and bound vectors," in *ACNS 20, Part I*, ser. LNCS, M. Conti, J. Zhou, E. Casalicchio, and A. Spognardi, Eds., vol. 12146. Springer, Heidelberg, Oct. 2020, pp. 481–501.

[16] A. Jalali, R. Azarderakhsh, M. M. Kermani, and D. Jao, "Towards optimized and constant-time CSIDH on embedded devices," in *COSADE 2019*, ser. LNCS, I. Polian and M. Stöttinger, Eds., vol. 11421. Springer, Heidelberg, Apr. 2019, pp. 215–231.

[17] H. Onuki, Y. Aikawa, T. Yamazaki, and T. Takagi, "A faster constant-time algorithm of CSIDH keeping two points," Cryptology ePrint Archive, Report 2019/353, 2019, https://eprint.iacr.org/2019/353.

[18] D. Cervantes-Vázquez, M. Chenu, J.-J. Chi-Domínguez, L. De Feo, F. Rodríguez-Henríquez, and B. Smith, "Stronger and faster side-channel protections for CSIDH," in *LATINCRYPT 2019*, ser. LNCS, P. Schwabe and N. Thériault, Eds., vol. 11774. Springer, Heidelberg, Oct. 2019, pp. 173–193.

[19] D. Jao and L. De Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, B.-Y. Yang, Ed. Springer, Heidelberg, Nov. / Dec. 2011, pp. 19–34.

[20] W. Castryck and T. Decru, "An efficient key recovery attack on SIDH (preliminary version)," Cryptology ePrint Archive, Report 2022/975, 2022, https://eprint.iacr.org/2022/975.

[21] L. Maino and C. Martindale, "An attack on SIDH with arbitrary starting curve," Cryptology ePrint Archive, Report 2022/1026, 2022, https://eprint.iacr.org/2022/1026.

[22] D. Robert, "Breaking SIDH in polynomial time," Cryptology ePrint Archive, Report 2022/1038, 2022, https://eprint.iacr.org/2022/1038.

[23] Microsoft, "PQCrypto-SIDH," 2020. [Online]. Available: https://github.com/Microsoft/PQCrypto-SIDH

[24] O. J. Bedrij, "Carry-select adder," *IRE Transactions on Electronic Computers*, no. 3, pp. 340–346, 1962.

[25] A. Yee, "Integer addition and carryout," http://www.numberworld.org/y-cruncher/internals/addition.html#ks_add, Feb. 2019.

[26] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 786–793, 1973.

[27] A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez, "A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1622–1636, 2017.

[28] L. De Feo, A. Leroux, and B. Wesolowski, "New algorithms for the deuring correspondence: SQISign twice as fast," Cryptology ePrint Archive, Report 2022/234, 2022, https://eprint.iacr.org/2022/234.

[29] J. Doliskani, G. C. C. F. Pereira, and P. S. L. M. Barreto, "Faster cryptographic hash function from supersingular isogeny graphs," Cryptology ePrint Archive, Report 2017/1202, 2017, https://eprint.iacr.org/2017/1202.

[30] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.