Highly Vectorized SIKE for AVX-512

Hao Cheng, Georgios Fotiadis, Johann Großschädl and Peter Y. A. Ryan

DCS and SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
{hao.cheng,georgios.fotiadis,johann.groszschaedl,peter.ryan}@uni.lu

Abstract. It is generally accepted that a large-scale quantum computer would be capable to break any public-key cryptosystem used today, thereby posing a serious threat to the security of the Internet's public-key infrastructure. The US National Institute of Standards and Technology (NIST) addresses this threat with an open process for the standardization of quantum-safe key establishment and signature schemes, which is now in the final phase of the evaluation of candidates. SIKE (an abbreviation of Supersingular Isogeny Key Encapsulation) is one of the alternate candidates under evaluation and distinguishes itself from other candidates due to relatively short key lengths and relatively high computing costs. In this paper, we analyze how the latest generation of Intel's Advanced Vector Extensions (AVX), in particular AVX-512IFMA, can be used to minimize the latency (resp. maximize the throughput) of the SIKE key encapsulation mechanism when executed on Ice Lake CPUs based on the Sunny Cove microarchitecture. We present various techniques to parallelize and speed up the base/extension field arithmetic, point arithmetic, and isogeny computations performed by SIKE. All these parallel processing techniques are combined in AvxSike, a highly optimized implementation of SIKE using Intel AVX-512IFMA instructions. Our experiments indicate that AvxSike instantiated with the SIKEp503 parameter set is approximately 1.5 times faster than the to-date best AVX-512IFMA-based SIKE software from the literature. When executed on an Intel Core i3-1005G1 CPU, AvxSike outperforms the x64 assembly implementation of SIKE contained in Microsoft's SIDHv3.4 library by a factor of about 2.5 for key generation and decapsulation, while the encapsulation is even 3.2 times faster.

Keywords: Post-Quantum Cryptography · Isogeny-Based Cryptography · Software Optimization · Finite-Field Arithemtic · SIMD-Parallel Processing

1 Introduction

In 2016, the NIST became engaged in Post-Quantum Cryptography (PQC) and started an initiative to solicit, evaluate, and standardize quantum-safe public-key cryptographic algorithms [CJL+16]. In response to a call for proposals for post-quantum encryption (resp. key encapsulation) and digital signature algorithms, a total of 72 candidates were submitted by November 2017. 69 of those submissions met the minimum requirements for acceptability and entered the first round of the evaluation process. In early 2019, the NIST selected 26 of the submissions as candidates for the second round; amongst them were 17 public-key encryption/key-encapsulation algorithms and nine digital signature schemes. The algorithms for encryption (resp. key encapsulation) include nine that are based on hard computational problems in lattices, seven whose security rests on classical problems in coding theory, and one that claims its security from the presumed hardness of the supersingular isogeny walk problem on elliptic curves [Nat19]. About 18 months later, in July 2020, the number of candidates was further reduced to only seven, which entered the third and final round of NIST's evaluation process. Among the finalists are four encryption or key-encapsulation schemes; three of them are lattice-based and one

falls into the code-based category. In addition, the NIST also announced eight so-called "alternate candidates", which can still become part of the standard after the third round (i.e. some of the alternate candidates may be considered in a fourth round [Nat20]).

The Supersingular Isogeny Key Encapsulation (SIKE) protocol [JAC+20] is one of the alternate candidates for quantum-safe key encapsulation retained by the NIST. Its main attractions are relatively short secret and public keys, making it somewhat comparable with conventional ("pre-quantum") elliptic-curve key exchange protocols like ECDH and X25519 [HMV04]. Furthermore, since the low-level arithmetic of SIKE is basically long-integer arithmetic, implementers can (potentially) re-use existing hardware accelerators and software libraries for pre-quantum cryptosystems like RSA and ECC. SIKE is based on the Supersingular Isogeny Diffie-Hellman (SIDH) key exchange, which was proposed by Jao and De Feo in 2011 [JD11] as a post-quantum cryptosystem whose security rests on the difficulty of finding isogenies between supersingular curves. In short, SIKE applies a Fujisaki-Okamoto transformation [HHK17] on SIDH to obtain a Key-Encapsulation Mechanism (KEM) that is secure against Chosen Ciphertext Attacks (CCA). State-of-the-art parameter sets for SIKE use supersingular curves over quadratic extension fields of prime characteristic, where the length of the prime is between 434 and 751 bits. The main drawback of SIKE is high computing costs and long latency, caused mainly by the serial computation of these isogenies, which represents a serious bottleneck for practical applications. For example, the currently fastest software implementation of SIKE for the ARM Cortex-M4 platform [AAM21] is more than two orders of magnitude slower than the best lattice-based KEMs benchmarked in [KP21]. Therefore, optimization techniques to accelerate SIDH and SIKE are an important topic in PQC research.

An analysis of the academic literature on performance optimizations for SIDH and SIKE shows that past research can be broadly divided into two categories. Research in the first category is concerned with mathematical techniques and higher-level arithmetic optimizations to make the point arithmetic and isogeny computations more efficient, see e.g. [ABJK18, CLN16, COR20, FLOR18]. The second category covers research on software optimizations for the underlying field arithmetic, whereby the modular reduction received particular attention [BF20, BI21, SLLH18, TWL+20]. Most of the highly-tuned implementations published in the literature adopt the Montgomery modular reduction method [Mon85] since it is extremely efficient in software. The very first implementation of SIDH was introduced roughly 10 years ago [JD11] and uses the GMP library for the low-level arithmetic. Since then, many implementations of SIDH or SIKE with dedicated field-arithmetic functions written in Assembly language have been developed. Microsoft's PQCrypto-SIDH library, which is available on GitHub under MIT license, contains the to-date fastest x64 Assembler implementation of SIKE. This library features most of the improvements and optimizations that were presented in the literature to accelerate the SIKE protocol and make it more practical. However, despite a large body of research on fast software implementation, SIKE is still significantly slower than other post-quantum KEMs, in particular the lattice-based third-round NIST candidates.

The 64-bit Intel architecture (i.e. x64) serves as the main benchmarking platform to analyze and compare the efficiency of the NIST PQC candidates. Besides the x64 base instruction set, 64-bit Intel processors also support different kinds of vector instructions for a SIMD-parallel execution of workloads. Vector extensions for the Intel architecture have a history that stretches back some 25 years and began with the introduction of the MMX extensions for the 32-bit x86 architecture. Thereafter came numerous generations of Streaming SIMD Extensions (SSE), which support vectors of a length of 128 bits, and *Advanced Vector eXtensions (AVX)*. The most recent new member of the AVX family is AVX-512, which augments the execution environment of x64 by 32 registers of a length of 512 bits and various new instructions. These instructions can operate on e.g. sixteen 32-bit elements or eight 64-bit elements in a SIMD-parallel fashion. AVX-512 comprises

a set of core instructions called AVX-512F and multiple extensions that are optional and may be implemented independently. One of these optional extensions provides the so-called "Integer Fused Multiply and Add" (IFMA) instructions, which were designed to speed up big-integer arithmetic [GK16]. The IFMA extension is supported by all mobile and workstation/server processors code-named "Ice Lake" and their successors[1].

Contributions. In this paper, we study how the massive parallel processing capabilities of the latest generation of the AVX vector engines, in particular AVX-512IFMA, can be used to improve the efficiency of SIKE-based key encapsulation. Since AVX-512IFMA is a relatively recent extension of the AVX-512 architecture, it is still (widely) unexplored how its new instructions can be used to speed up SIKE. To our knowledge, there exists currently only one publication dealing with AVX-512 optimizations for SIKE, namely the ARITH 2019 paper of Kostic and Gueron [KG19], but their work focuses solely on the low-level field arithmetic, i.e. they did not explore avenues for parallel processing at the higher levels of SIKE. Hence, it is still unknown how AVX-512IFMA can be exploited to unleash the full potential of modern Intel processors for executing SIKE and what latency (resp. throughput) a carefully optimized implementation could achieve.

The present paper aims to fill this gap by introducing novel techniques to parallelize (and speed up) the field arithmetic, point arithmetic, and isogeny computations. At the lowest level, we present a carefully-optimized library for arithmetic operations in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ that uses a radix-2^{51} representation for the operands (i.e. 51 bits/limb) and adopts Montgomery's algorithm for modular reduction. We developed different variants of this arithmetic library, including one that minimizes the latency of two (resp. four) instances of an arithmetic operation, and one that maximizes the throughput of eight instances using the so-called limb-slicing technique[2]. At the medium level, we describe techniques for parallel point arithmetic operations on Montgomery curves, whereby we paid special attention to find viable trade-offs between the number of parallel instances of point and field operations, respectively. Finally, at the highest layer, we discuss various approaches for vectorized isogeny computation and key encapsulation. All these parallel processing techniques are combined in AvxSike, an optimized implementation of SIKE using Intel's AVX-512IFMA instructions. AvxSike supports all four (uncompressed) parameter sets given in [JAC+20] and comes with a low-latency version and a high-throughput version of SIKE, which we call AvxSike-LL and AvxSike-HT, respectively. Both versions are resistant against timing-based side-channel attacks in the sense that they do not contain any secret-dependent conditional statements or memory accesses. Our latency-optimized AvxSike instantiated with the SIKEp503 parameters is about 1.5 times faster than the AVX-512IFMA-based SIKE software presented in [KG19]. It also outperforms Microsoft's x64 Assembler implementation[3] of SIKE by a factor of about 2.5 for both key generation and decapsulation, and even 3.2 for encapsulation, when benchmarked on an Intel Core i3-1005G1 processor. Furthermore, our throughput-optimized AvxSike reaches an up to 4.6-fold higher throughput than Microsoft's SIKE library.

Availability of the Source Code. The source code of our AvxSike software is available online at https://gitlab.uni.lu/APSIA/AVXSIKE. This repository contains both the low-latency version AvxSike-LL and the high-throughput version AvxSike-HT.

[1] According to Intel there exist currently 13 "Ice Lake" processors for the mobile segment and 43 "Ice Lake" processors for the workstation/server segment, see https://ark.intel.com/content/www/us/en/ark/products/codename/74979/products-formerly-ice-lake.html.

[2] Limb-slicing uses a "reduced-radix" representation for the operands and is somewhat similar to the bit-slicing technique used in symmetric cryptography, i.e. it allows one to compute a batch of arithmetic operations in a SIMD-parallel way, which increases throughput at the expense of latency [CGT+21].

[3] We used version 3.4 of Microsoft's PQCrypto-SIDH library (i.e. SIDHv3.4), which is available on GitHub at https://github.com/Microsoft/PQCrypto-SIDH, as starting point for our work and the x64 assembly implementation of SIKE contained in this library as baseline for performance comparisons.

Outline. In Section 2, we review the SIKE key encapsulation mechanism and describe our target platform (focussing on the AVX-512IFMA vector instructions) as well as the experimental environment for collecting benchmarking results. Then, from Section 3 to Section 6, we introduce our AvxSike software layer by layer. Section 3 explains two different types of vectorized integer multiplication and Montgomery reduction. Various vectorized implementations of quadratic extension-field operations are described in detail in Section 4. Later, in Section 5, we focus on vectorized implementations of arithmetic operations on Montgomery curves. In Section 6, we present a low-latency version and a high-throughput version of AvxSike, our vectorized SIKE software. We compare the performance of AvxSike, Microsoft's SIDHv3.4 assembly library, and the IFMA-based SIKEp503 implementation of Kostic and Gueron in Section 7. Finally, in Section 8, we draw conclusions and discuss avenues for future work.

2 Preliminaries

We start with a summary of the mathematical background of isogenies of elliptic curves and proceed with a concise description of the SIKE mechanism [JAC+20]. Later, we give an overview of the AVX-512 instruction set architecture and introduce our experimental environment for performance measurements.

Let E and E' be two elliptic curves over a finite field $\mathbb{F}_q$ of prime characteristic p. An *isogeny* $\phi : E \to E'$ is a non-constant rational map defined over $\mathbb{F}_q$ that maps the identity element of E to the identity element of E', and we say E, E' are *isogenous* if and only if $\#E(\mathbb{F}_q) = \#E'(\mathbb{F}_q)$ [Tat66]. In isogeny-based cryptography, we are interested in *separable* isogenies [JD11]. Such isogenies have finite kernel and their degree is defined as $\deg \phi = \# \ker \phi$. In addition, given an elliptic curve E over $\mathbb{F}_q$ and a finite subgroup $G \subseteq E(\mathbb{F}_q)$, there exists a unique isogeny $\phi : E \to E' = E/G$, with $\ker(\phi) = G$ and $\deg(\phi) = \#G$. Isogeny-based cryptosystems generally use supersingular elliptic curves of smooth order since they facilitate the computation of isogenies of exponentially large degree by composing lower-degree isogenies that can be efficiently computed with Vélu's formulæ [Vél71]. Furthermore, every supersingular elliptic curve E defined over $\mathbb{F}_q$ can also be defined over $\mathbb{F}_{p^2}$, in which case $\#E(\mathbb{F}_{p^2}) = (p+1)^2$.

In the SIKE protocol, supersingular elliptic curves are represented using the Montgomery model [JAC+20]. A Montgomery curve in affine form is given by the equation $E_{(a,b)} : by^2 = x^3 + ax^2 + x$, where $a, b \in \mathbb{F}_{p^2}$ and $b(a^2 - 4) \neq 0$. It is often beneficial to work in the projective form, both in terms of curve points and curve coefficients. In this case, we write $E_{(A:B:C)} : BY^2Z = CX^3 + AX^2Z + CXZ^2$ with $a = A/C$, $b = B/C$, and $(x, y) = (X/Z, Y/Z)$. Montgomery curves are well known for efficient point arithmetic on their Kummer line, originally proposed in [Mon87], which entirely ignores the projective Y coordinate. In addition, they allow one to ignore the coefficient B in point operations and isogeny computations. Consequently, we will denote by $E_{(A:C)}$ a Montgomery curve with $B = 1$ and by $P = (X_P : Z_P)$ a point on the curve.

2.1 Supersingular Isogeny Key Encapsulation (SIKE)

SIKE is a key encapsulation mechanism from the family of isogeny-based schemes. It was inspired by the SIDH protocol of Jao and De Feo [JD11] and is currently evaluated as an alternate candidate in the NIST PQC standardization process [Nat20].

Public Parameters. We fix two positive integers e_2 and e_3 such that $p = 2^{e_2}3^{e_3} - 1$ is prime. Primes of this form are Montgomery-friendly, meaning that they allow for some optimizations of the modular arithmetic, see e.g. [CLN16]. We define the two key spaces $\mathcal{K}_2 = \{0, \ldots, 2^{e_2} - 1\}$ and $\mathcal{K}_3 = \{0, \ldots, 3^{e_3} - 1\}$ for sampling secret keys. Further, we

also fix a starting supersingular elliptic curve $E_0 : y^2 = x^3 + 6x^2 + x$ over $\mathbb{F}_{p^2}$, where $\#E_0(\mathbb{F}_{p^2}) = (2^{e_2}3^{e_3})^2$, and two bases $\{P_2, Q_2\}$ and $\{P_3, Q_3\}$, which generate the torsion subgroups $E_0[2^{e_2}]$ and $E_0[3^{e_3}]$, respectively. The public parameters consist of the curve E_0 and the 3-tuples $\{x_{P_2}, x_{Q_2}, x_{PQ_2}\}$ and $\{x_{P_3}, x_{Q_3}, x_{PQ_3}\}$, where $x_{PQ_2} = x_{P_2} - x_{Q_2}$ and $x_{PQ_3} = x_{P_3} - x_{Q_3}$[4].

For each $\ell \in \{2, 3\}$, we denote by $(E', \phi_\ell) \leftarrow \texttt{isogeny}_\ell(E, x_{R_\ell})$ the computation of an isogeny $\phi_\ell : E \to E'$ of degree ℓ^{e_ℓ} and $\ker \phi_\ell = \langle x_{R_\ell} \rangle$, where $x_{R_\ell} = x_{P_\ell} + [\mathrm{sk}_\ell] x_{Q_\ell}$. Each secret key sk_ℓ is chosen randomly from $\mathcal{K}_\ell$ and the corresponding public key is obtained as $\mathrm{pk}_\ell = (\phi_\ell(x_{P_m}), \phi_\ell(x_{Q_m}), \phi_\ell(x_{PQ_m}))$, where $m \in \{2, 3\}$ such that $m \neq \ell$.

Algorithm 1: Public key encryption: SIPKE = (Gen, Enc, Dec)

function Gen()
- $\mathrm{sk}_3 \leftarrow_\$ \mathcal{K}_3$
- $x_{R_3} \leftarrow x_{P_3} + [\mathrm{sk}_3] x_{Q_3}$
- $(\phi_3, E_3) \leftarrow \texttt{isogeny}_3(E_0, x_{R_3})$
- $\mathrm{pk}_3 \leftarrow (\phi_3(x_{P_2}), \phi_3(x_{Q_2}), \phi_3(x_{PQ_2}))$
- **return** $(\mathrm{sk}_3, \mathrm{pk}_3)$

function Enc($\mathrm{pk}_3, m \in \mathcal{M}, \mathrm{sk}_2 \in \mathcal{K}_2$)
- $x_{R_2} \leftarrow x_{P_2} + [\mathrm{sk}_2] x_{Q_2}$
- $(\phi_2, E_2) \leftarrow \texttt{isogeny}_2(E_0, x_{R_2})$
- $c_1 \leftarrow (\phi_2(x_{P_3}), \phi_2(x_{Q_3}), \phi_2(x_{PQ_3}))$
- $x'_{R_2} \leftarrow \phi_3(x_{P_2}) + [\mathrm{sk}_2] \phi_3(x_{Q_2})$
- $(\phi'_2, E_{32}) \leftarrow \texttt{isogeny}_2(E_3, x'_{R_2})$
- $h \leftarrow \texttt{SHAKE256}(j(E_{32}))$
- $c_2 \leftarrow h \oplus m$
- **return** (c_1, c_2)

function Dec($\mathrm{sk}_3, (c_1, c_2)$)
- $x'_{R_3} \leftarrow \phi_2(x_{P_3}) + [\mathrm{sk}_3] \phi_2(x_{Q_3})$
- $(\phi'_3, E_{23}) \leftarrow \texttt{isogeny}_3(E_2, x'_{R_3})$
- $h \leftarrow \texttt{SHAKE256}(j(E_{23}))$
- $m \leftarrow h \oplus c_2$
- **return** m

Algorithm 2: Key encapsulation: SIKE = (KeyGen, Encaps, Decaps)

function KeyGen()
- $(\mathrm{sk}_3, \mathrm{pk}_3) \leftarrow \mathsf{Gen}()$
- $s \leftarrow_\$ \{0,1\}^n$
- **return** $(s, \mathrm{sk}_3, \mathrm{pk}_3)$

function Encaps(pk_3)
- $m \leftarrow_\$ \{0,1\}^n$
- $\mathrm{sk}_2 \leftarrow \texttt{SHAKE256}(m \parallel \mathrm{pk}_3)$
- $(c_1, c_2) \leftarrow \mathsf{Enc}(\mathrm{pk}_3, m, \mathrm{sk}_2)$
- $k \leftarrow \texttt{SHAKE256}(m \parallel (c_1, c_2))$
- **return** $(k, (c_1, c_2))$

function Decaps($s, \mathrm{sk}_3, \mathrm{pk}_3, (c_1, c_2)$)
- $m' \leftarrow \mathsf{Dec}(\mathrm{sk}_3, (c_1, c_2))$
- $\mathrm{sk}'_2 \leftarrow \texttt{SHAKE256}(m' \parallel \mathrm{pk}_3)$
- $x_{R_2} \leftarrow x_{P_2} + [\mathrm{sk}'_2] x_{Q_2}$
- $(\phi_2, E_2) \leftarrow \texttt{isogeny}_2(E_0, x_{R_2})$
- $c'_1 \leftarrow (\phi_2(x_{P_3}), \phi_2(x_{Q_3}), \phi_2(x_{PQ_3}))$
- **if** $c'_1 = c_1$ **then**
  - $k \leftarrow \texttt{SHAKE256}(m' \parallel (c_1, c_2))$
- **else**
  - $k \leftarrow \texttt{SHAKE256}(s \parallel (c_1, c_2))$
- **return** k

SIKE and SIPKE. The SIKE submission comes with four parameter sets that provide different levels of security and are named according to the size of the underlying prime p: SIKEp434, SIKEp503, SIKEp610, and SIKEp751. In all versions, the public parameters e_2 and e_3 are chosen so that $2^{e_2} \approx 3^{e_3}$. At the core of SIKE is the Supersingular Isogeny Public Key Encryption scheme (SIPKE) [DJP14], which offers the usual three functions (Gen, Enc, Dec) for key generation, encryption, and decryption (see Algorithm 1). Note that the ciphertext consists of two components $c_1 = (\phi_2(x_{P_3}), \phi_2(x_{Q_3}), \phi_2(x_{PQ_3}))$ and $c_2 = h \oplus m$, where h is the hash of $j(E_{32})$, the j-invariant of curve E_{32}. The first component is essential in the decryption process, particularly for the computation of the kernel generator $\phi_2(x_{P_3}) + [\mathrm{sk}_3]\phi_2(x_{Q_3})$, which defines the isogeny $\phi'_3 : E_2 \to E_{23}$. Decryption works because $E_{32} \cong E_{23}$ and hence $j(E_{32}) = j(E_{23})$. Like any other key encapsulation mechanism, SIKE consists of the usual three functions (KeyGen, Encaps, Decaps) for the key generation, encapsulation, and decapsulation, as described in Algorithm 2. In the original SIKE submission, the hash function used by both SIPKE and SIKE is actually an eXtendable Output Function (XOF), namely SHAKE256 [Nat15], which belongs to the SHA-3 family and has been approved by the NIST and other standardization bodies.

[4] The x-coordinates of $PQ_2 = P_2 - Q_2$ and $PQ_3 = P_3 - Q_3$ are used in the differential addition.

Security of SIKE. The security of SIKE relies on the SIDH problem, which is defined as follows: given the curves E_0, E_2, E_3 and points $\phi_2(P_3), \phi_2(Q_3), \phi_3(P_2), \phi_3(Q_2)$, determine the j-invariant of the curve $E_2/\langle \phi_2(P_3) + [\mathrm{sk}_3]\phi_3(Q_3)\rangle$ or $E_3/\langle \phi_3(P_2) + [\mathrm{sk}_2]\phi_3(Q_2)\rangle$. To date, the best classical algorithm for attacking SIKE is due to Galbraith [Gal99] and has a complexity of $O(\sqrt[4]{p})$, while the best quantum attack is Tani's claw finding algorithm [Tan09] with a complexity of $O(\sqrt[6]{p})$. In addition, Proposition 1 in the SIKE specification [JAC+20] proves that the SIPKE scheme described in Algorithm 1 is IND-CPA secure in the random oracle model, if the SIDH problem is hard and the SIKE mechanism is also proven to be IND-CCA secure. The parameter sets SIKEp434, SIKEp503, SIKEp610, and SIKEp751 correspond to NIST security level 1, 2, 3, and 5, respectively [JAC+20].

2.2 Optimized Isogeny Computations

The most demanding part of the SIKE protocol is the isogeny computations. Namely, in all three functions the SIKE suite consists of, an isogeny $\phi : E_0 \to E_{e_\ell}$ of degree ℓ^{e_ℓ} has to be computed, with kernel generated by a point $R_0 = P_0 + [\mathrm{sk}_\ell]Q_0 \in E[\ell^{e_\ell}]$, where $\ell \in \{2, 3\}$ and $\mathrm{sk}_\ell \in \{0, \ldots, \ell^{e_\ell} - 1\}$. Instead of directly computing this isogeny ϕ, the best practice is to break the computation in smaller parts where we iteratively compute e_ℓ isogenies of degree ℓ using the Vélu formulæ [Vél71] and compose them to obtain the desired ℓ^{e_ℓ}-isogeny. The straightforward approach is to carry out an iterative procedure for $0 \leq i < e_\ell$, whereby in each iteration we fix the kernel point $S_i = [\ell^{\ell_i - i - 1}]R_i$ to be of order ℓ and compute an isogeny $\phi_i : E_i \to E_{i+1} = E_i/\langle S_i\rangle$ of degree ℓ. Thereafter, we compute the image $R_{i+1} = \phi_i(R_i)$ in order to be able to obtain the kernel point on the new curve E_{i+1} for the next isogeny. The desired ℓ^{e_ℓ}-isogeny $\phi : E_0 \to E_\ell$ is represented as the composition $\phi = \phi_{e_\ell - 1} \circ \ldots \circ \phi_0$. This approach is *multiplication-oriented* since, in each iteration, a scalar multiplication $[\ell^{\ell_i - i - 1}]S_i$ has to be performed [JD11].

An alternative method, known as *isogeny-oriented*, was proposed by Jao and De Feo [JD11]. The aim in this method is to reduce the number of scalar multiplications at the cost of extra isogeny computations. This is done by computing an initial list of points $([\ell^j]R_0)_{j < e_\ell}$ on the starting curve and then, in each iteration for $0 \leq i < e_\ell$, update this list as the image of the points under the isogeny ϕ_i, i.e. $[\ell^j]R_{i+1} = \phi_i([\ell^j]R_i)$, for each $j = i, \ldots, e_\ell - 1$. In [DJP14], De Feo, Jao, and Plût showed that it is possible to speed up the isogeny-oriented approach by introducing the notion of *optimal strategies*, which allow for less scalar multiplications compared to the multiplication-based approach and less isogeny computations compared to the isogeny-based approach. These strategies are presented and implemented in the SIKE specification document [JAC+20] for the small primes $\ell \in \{2, 3\}$. Instead of point additions, the authors of the SIKE specification use only point doubling for the case $\ell = 2$ and point tripling for the case $\ell = 3$. Moreover, in the case $\ell = 2$, the authors compute iteratively isogenies of degree 4 instead of 2.

2.3 Intel Advanced Vector Extension AVX-512

AVX-512 is the latest generation of the Advanced Vector eXtensions (AVX) and enriches the x64 execution environment by thirty-two 512-bit registers (`zmm0`–`zmm31`) and various 512-bit instructions. AVX-512 consists of multiple extensions, whereby AVX-512F is the core extension with a 32-bit vector multiplier. Starting with Cannon Lake (Palm Cove microarchitecture), Intel integrated the so-called Integer Fused Multiply-Add extension (AVX-512IFMA, or simply IFMA) into AVX-512 [Int18], which was specifically designed to speed up public-key cryptographic software relying on large integer arithmetic (this also includes isogeny-based cryptosystems). Concretely, the two new IFMA instructions `vpmadd52luq` and `vpmadd52huq` multiply a pair of eight packed unsigned 52-bit integers (one located in each 64-bit element of two 512-bit vectors) to obtain eight intermediate products, each being 104 bits long. Then, either the lower 52 bits (`vpmadd52luq`) or the

upper 52 bits (`vpmadd52huq`) of these products are added to the eight packed unsigned 64-bit integers of a 512-bit destination register, which holds the final result. Compared to the `vpmuludq` and `vpmuldq` multiply instruction of AVX-512F, the IFMA extension does not only offer a wider multiplier of 52 bits, but also combines vector multiplication and vector addition into a single instruction.

Target Platform and Performance Measurements. AVX-512IFMA started to become commonly available with the Intel x64 processor family codenamed "Ice Lake" and its successors, e.g. "Tiger Lake" and "Rocket Lake." The "Ice Lake" family comprises 10th generation Intel Core mobile and 3rd generation Xeon scalable server processors based on the "Sunny Cove" microarchitecture. We developed our AvxSike software on (and optimized it for) a 10th generation Core, namely the i3-1005G1. On an "Ice Lake" Core CPU, both `vpmadd52luq` and `vpmadd52luq` have a throughput of one instruction/cycle and a latency of four cycles. AvxSike was written in C and uses compiler intrinsics to perform AVX-512 vector operations. We compiled the source code of AvxSike and the Microsoft SIDHv3.4 library with GCC version 9.3.0 and measured their execution times on our Core CPU, whereby turbo boost was disabled. However, we could not measure the execution time of the AVX-512IFMA implementation of Kostic and Gueron because the source code is not publicly available. Therefore, we resort to the timings reported in [KG19] and include these in our performance comparisons.

3 Prime-Field Arithmetic

In this section, we describe vectorized implementations of big-integer multiplication and Montgomery reduction at the $\mathbb{F}_p$-arithmetic layer, which are highly performance-critical operations of our AvxSike software. Note that AvxSike uses only IFMA instructions (i.e. `vpmadd52luq` and `vpmadd52huq`) for all vector-parallel multiplications, i.e. the basic AVX-512F multiply instructions like `vpmuludq` and `vpmuldq` are not executed at all. We adopt the term "$(y \times z)$-way parallelism" to describe an implementation that performs y prime-field (or integer-arithmetic) operations simultaneously, whereby each operation is executed in a z-way parallel fashion and, thus, uses z elements of a vector. For example, Algorithm 2 in [KG19] contains pseudo-code of a (1×8)-way integer multiplication for SIKEp503, which means it is a single multiplication of 512-bit integers that uses all eight 64-bit elements of an AVX-512 vector. Due to better instruction-level parallelism and the possibility of taking advantage of Karatsuba's method, the theoretically-optimal (8×1)-way approach is more efficient than other parallel processing techniques such as (4×2)-way, (2×4)-way, and (1×8)-way. Taking into account options for higher-level parallelism offered by the $\mathbb{F}_{p^2}$-arithmetic layer and the curve-arithmetic layer, we found that all prime-field operations can be executed in either an (8×1)-way or a (4×2)-way fashion, i.e. AvxSike always performs eight or four $\mathbb{F}_p$-operations simultaneously.

In this section (and also the four subsequent sections), we focus on SIKEp503 as case study to explain our vectorization techniques since it is the only parameter set that was considered in essentially every previous paper on fast SIKE software, including [KG19].

3.1 Radix-2^{51} Representation

Due to the 52-bit wide vector multiplier, most AVX-512IFMA implementations, such as [KG19, CFG+21], directly adopt the natural radix-2^{52} (i.e. 52 bits/limb) representation for the operands. However, in this work, we take advantage of a radix-2^{51} representation based on two main considerations. First, although a radix of 2^{52} is a reduced radix with respect to the 64-bit length of an element of an AVX-512 vector (there are still 12 bits of "headroom" for storing carry bits to delay carry propagation), it is saturating for the

52-bit multiplier because all limbs must be reduced to 52 bits before IFMA instructions can be executed on them. This is not ideal for operations like our (8×1)-way parallel version of Karatsuba multiplication [KO63], where the sums of two half-length additions are operands of the last half-length multiplication[5]. In such a situation, a representation based on radix 2^{52} would make it necessary to instantly propagate the carries produced by the two additions, which does not only require extra instructions, but also generates one more limb. In contrast, a radix of 2^{51} allows one to simply keep the carry bits and delay the carry propagation, thereby increasing the length of limbs to 52 bits. A second reason to favor a radix of 2^{51} is the efficient (4×2)-way carry propagation introduced in [OAL18], which we use in our (4×2)-way implementation of the prime-field arithmetic operations. This efficient carry propagation is "incomplete" in the sense that, after the propagation, two limbs are allowed to exceed the nominal limb-length by one bit. But in our case, when using a radix of 2^{52}, it is not possible to tolerate two over-length limbs (i.e. all limbs strictly have to fit into 52 bits, which costs some extra instructions). On the other hand, with a radix of 2^{51}, this problem does not arise since we can allow two limbs to have a length of 52 bits, while the other limbs are still 51 bits long. Finally, we remark that using a radix-2^{51} representation does not increase the number of limbs (in relation to a radix of 2^{52}) for any of the four parameter sets of SIKE.

(8×1)-way Limb Vector Set. The main data structure of our (8×1)-way prime-field operations is the (8×1)-way *limb vector set* composed of eight radix-2^{51} integers. Given eight integers $a, b, c, d, e, f, g, h \in \mathbb{F}_p$, an (8×1)-way limb vector set $\boldsymbol{U}$ for SIKEp503 is defined as:

$$\boldsymbol{U} = \langle a, b, c, d, e, f, g, h \rangle = \left\{ \begin{matrix} [a_0, b_0, c_0, d_0, e_0, f_0, g_0, h_0] \\ [a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1] \\ \vdots \\ [a_9, b_9, c_9, d_9, e_9, f_9, g_9, h_9] \end{matrix} \right\} = (U_0, U_1, \ldots, U_9), \qquad (1)$$

where each $U_i = [a_i, b_i, c_i, d_i, e_i, f_i, g_i, h_i]$ is called a *limb vector*.

(4×2)-way Limb Vector Set. Our (4×2)-way field-operations use (4×2)-way limb vector sets, which also based on the radix 2^{51}, but contain only four integers. In the case of SIKEp503, a (4×2)-way limb vector set $\boldsymbol{V} = \langle a, b, c, d \rangle$ has the following form:

$$\boldsymbol{V} = \langle a, b, c, d \rangle = \left\{ \begin{matrix} [a_0, a_5, b_0, b_5, c_0, c_5, d_0, d_5] \\ [a_1, a_6, b_1, b_6, c_1, c_6, d_1, d_6] \\ \vdots \\ [a_4, a_9, b_4, b_9, c_4, c_9, d_4, d_9] \end{matrix} \right\} = (V_0, V_1, \ldots, V_4). \qquad (2)$$

Each limb vector $V_i = [a_i, a_{i+5}, b_i, b_{i+5}, c_i, c_{i+5}, d_i, d_{i+5}]$ contains two 51-bit limbs from each integer, whereby the limbs are arranged in an interleaved pattern.

Reduction Modulo $2p$. Similar to SIDHv3.4, both our (8×1)-way and (4×2)-way implementation omit the final subtraction in the Montgomery reduction. All prime-field operations of AvxSike actually perform the reduction modulo $2p$ instead of p. To give a concrete example, the modular addition operation first computes $t \leftarrow a + b$ and then performs a subtraction $r \leftarrow t - 2p$. If $r < 0$ we add $2p$ to r, otherwise we add 0 (to have operand-independent execution time). The modular subtraction just directly computes $r \leftarrow a - b$ and then executes the same correction step as the modular addition.

[5] An $2n$-bit integer multiplication according to Karatsuba's algorithm is computed via the equation $r = a \cdot b = (a_0 + a_1 \cdot 2^n) \cdot (b_0 + b_1 \cdot 2^n) = a_0 b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1] \cdot 2^n + a_1 b_1 \cdot 2^{2n}$. To obtain $(a_0 + a_1)(b_0 + b_1)$, two n-bit additions have to be carried out before the n-bit multiplication.

3.2 Integer Multiplication

(8×1)-way Implementation. All (8×1)-way prime-field arithmetic functions operate on (8×1)-way limb vector sets and were developed based on the "limb-slicing" approach [CGT+21], which essentially duplicates a 1-way implementation to eight 64-bit elements using AVX-512 instructions. The multiplication consists of one level of Karatsuba with product scanning underneath because, according to our experiments, this combination is faster than other techniques (e.g. basic operand scanning and product scanning) for all four parameter sets of SIKE. Note that our implementation of the integer multiplication does not involve a carry propagation, which reduces sequential dependencies among the instructions (though carries are always propagated during Montgomery reduction).

(4×2)-way Implementation. Orisaka, Aranha, and López introduced in [OAL18] an efficient (4×2)-way AVX-512F implementation of Montgomery multiplication, which is composed of integer multiplication, Montgomery reduction, and carry propagation. This implementation operates on the (4×2)-way limb vector sets we defined in the previous subsection and uses the normal `vpmuldq` multiply instruction of AVX-512F but not the two IFMA instructions. The integer multiplication part of this implementation is based on the operand-scanning method. We developed our (4×2)-way integer multiplication following the approach of [OAL18] but replaced `vpmuldq` by IFMA instructions.

3.3 Montgomery Reduction

(8×1)-way Implementation. SIKE uses "SIDH-friendly" primes, which are a special form of Montgomery-friendly primes and allow one to speed up the modular reduction operation compared to general primes [CLN16]. The optimized Montgomery reduction for a SIDH-friendly prime $p = 2^{e_2}3^{e_3} - 1$ is given in [CLN16, Algorithm 1] and exploits the fact that the e_2 least significant bits of $p + 1$ are all 0 (i.e. $p + 1$ is nothing else than 3^{e_3} left-shifted by e_2 bits). In other words, $p + 1$ consists of many limbs that are 0 and this makes it possible to save a large number of instructions compared to a conventional Montgomery reduction. As already mentioned above, the reduction operation involves a carry propagation to get a final result represented by 51-bit limbs.

(4×2)-way Implementation. The modular reduction part of the implementation from [OAL18] is a (4×2)-way version of conventional Montgomery reduction. We optimized this implementation by applying the approach of [CLN16] and using IFMA instructions to obtain our (4×2)-way Montgomery reduction. In addition, a final carry propagation is always performed after the Montgomery reduction to get a final result whose limbs are sufficiently short. As already mentioned earlier, there is one limb vector (containing two limbs of each field-element) in this final result that is one bit longer than the other limb vectors, i.e. in our case 52 bits instead of 51 (see [OAL18, Algorithm 4] for details).

3.4 Results and Comparison

Table 1 shows the execution times of integer multiplication, Montgomery reduction, and Montgomery multiplication of different implementations. As mentioned in Section 1, we use the SIDHv3.4 x64 assembly library as baseline for comparisons. Since our software is vectorized, the "cycles/instance" is a useful metric for us, and the speed-up ratio relates a specific implementation with the baseline under this metric. The (8×1)-way and the (4×2)-way parallel integer multiplication is respectively 4.9 and 3.9 times faster than SIDHv3.4. Regarding our parallel Montgomery reduction, the (8×1)-way version has almost the same latency as the (4×2)-way implementation, which means it is twice as fast from the viewpoint of a single instance. This massive difference of speed-up factors

Table 1: Experimental results of $\mathbb{F}_p$-arithmetic operations for SIKEp503.

Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
Integer multiplication	SIDHv3.4	x64 asm	1-way	1	100	100	1.00×
	AvxSike	AVX-512	(8×1)-way	8	165	21	4.85×
	AvxSike	AVX-512	(4×2)-way	4	102	26	3.92×
Montgomery reduction	SIDHv3.4	x64 asm	1-way	1	75	75	1.00×
	AvxSike	AVX-512	(8×1)-way	8	144	18	4.17×
	AvxSike	AVX-512	(4×2)-way	4	140	35	2.14×
Montgomery multiplication	SIDHv3.4	x64 asm	1-way	1	201	201	1.00×
	[KG19]	AVX-512	hybrid	1	195	195	1.03×
	AvxSike	AVX-512	(8×1)-way	8	302	38	5.32×
	AvxSike	AVX-512	(4×2)-way	4	264	66	3.05×

(in relation to the integer multiplication) can be explained with sequential dependencies in the (4×2)-way Montgomery reduction and, as a consequence, lower instruction-level parallelism compared to the (8×1)-way parallel version. Finally, when looking at the results for the Montgomery multiplication, it is striking that the IFMA implementation of Kostic and Gueron [KG19] is merely a few cycles faster than SIDHv3.4. According to [KG19], their Montgomery multiplication combines a (1×8)-way integer multiplication using IFMA with an x64 assembly implementation of Montgomery reduction. Thus, it is necessary to convert between radix-2^{52} vectors and radix-2^{64} large integers, which is an obvious bottleneck of their software. Since the multiplication is vectorized, but not the reduction, we can say that their software follows a *hybrid* implementation approach.

4 Quadratic Extension-Field Arithmetic

We first define an "$(x \times y \times z)$-way" parallel $\mathbb{F}_{p^2}$-arithmetic implementation: it performs x $\mathbb{F}_{p^2}$-operations in parallel, whereby each of them executes y prime-field operations in parallel, and each of the y prime-field operation uses z 64-bit elements of a vector. In all algorithms of this section, the variable t denotes an integer having a similar length as an element of $\mathbb{F}_p$, whereas tt usually represents an integer of roughly twice the length of an $\mathbb{F}_p$-element; mod p stands for a Montgomery reduction modulo p, but as mentioned in Section 3.1, the result is in the range of $[0, 2p)$ and not always fully reduced; $\times$ denotes an *integer* multiplication and $+$ is an *integer* addition (except of Algorithm 5, where it is a *modular* addition). Note that, for reasons of brevity and to have succinct pseudo-code descriptions of algorithms, we do not distinguish between various (*integer* and *modular*) subtractions, which are uniformly denoted as $-$, but the result of a subtraction is always non-negative. We refer readers who are interested in the full details of the algorithms to the source code of Microsoft's SIDHv3.4 or our AvxSike.

4.1 $\mathbb{F}_{p^2}$-Multiplication

1-way Parallelization at $\mathbb{F}_p$-level. In this implementation of a parallel $\mathbb{F}_{p^2}$-multiplication, each $\mathbb{F}_{p^2}$-multiplication instance performs only one integer-arithmetic or prime-field operation at a time. Formally, based on the above-defined $(x \times y \times z)$-way notation, we have $y = 1$. For such a $(x \times 1 \times z)$-way $\mathbb{F}_{p^2}$-multiplication, we use the same technique as the SIDHv3.4 library, namely Karatsuba's method. We explain this variant with a single $\mathbb{F}_{p^2}$-multiplication instance in Algorithm 3. Unlike to SIDHv3.4, which can perform the carry propagation through instructions like `ADC`, `ADCX`, and `ADOX`, our AVX-512 software has to carefully handle the carry propagation. Since carry propagation normally causes

Algorithm 3: $\mathbb{F}_{p^2}$-multiplication with 1-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 b_0 - a_1 b_1) + [(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1]i$.

1 $t_1 \leftarrow a_0 + a_1$
2 $t_2 \leftarrow b_0 + b_1$
3 $tt_1 \leftarrow a_0 \times b_0$
4 $tt_2 \leftarrow a_1 \times b_1$
5 $tt_3 \leftarrow t_1 \times t_2$
6 $tt_3 \leftarrow tt_3 - tt_1 - tt_2$
7 $tt_1 \leftarrow tt_1 - tt_2$
8 $r_0 \leftarrow tt_1 \bmod p$
9 $r_1 \leftarrow tt_3 \bmod p$
10 **return** $r = r_0 + r_1 i$

Algorithm 4: $\mathbb{F}_{p^2}$-multiplication with 2-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0)i$.

1 $tt_1 \leftarrow a_0 \times b_0$ $\quad ss_1 \leftarrow a_0 \times b_1$
2 $tt_2 \leftarrow a_1 \times b_0$ $\quad ss_2 \leftarrow a_1 \times b_1$
3 $tt_3 \leftarrow tt_1 - ss_2$ $\quad ss_3 \leftarrow tt_2 + ss_1$
4 $r_0 \leftarrow tt_3 \bmod p$ $\quad r_1 \leftarrow ss_3 \bmod p$
5 **return** $r = r_0 + r_1 i$

strong instruction dependencies, it always requires more clock cycles than a basic limb addition or subtraction. Algorithm 3 was designed to perform as few carry propagations as possible; the integer multiplication does not involve a propagation of carries and the subtractions at line 6 and 7 can "postpone" the carry propagations and integrate them into the subsequent Montgomery reductions. Using the (8×1)-way $\mathbb{F}_p$-arithmetic, we developed an $(8 \times 1 \times 1)$-way $\mathbb{F}_{p^2}$-multiplication via Algorithm 3, while the $(4 \times 1 \times 2)$-way implementation is based on the (4×2)-way $\mathbb{F}_p$-arithmetic.

2-way Parallelization at $\mathbb{F}_p$-level. In this variant (Algorithm 4), each $\mathbb{F}_{p^2}$-multiplication instance is internally parallelized in a 2-way fashion, namely each instance executes two prime-field (or integer-arithmetic) operations simultaneously, i.e. $y = 2$ according to the $(x \times y \times z)$-way notation from above. This variant uses the schoolbook method instead of Karatsuba's algorithm because it turned out that the latter is less efficient for 2-way parallelization at the $\mathbb{F}_p$-level. Compared to the $(x \times 1 \times z)$-way $\mathbb{F}_{p^2}$-multiplication, this 2-way variant has fewer additions, subtractions, and carry propagations, but needs one more multiplication at the $\mathbb{F}_p$-level. Using (8×1)-way and (4×2)-way $\mathbb{F}_p$ arithmetic, we developed a $(4 \times 2 \times 1)$-way and a $(2 \times 2 \times 2)$-way $\mathbb{F}_{p^2}$-multiplication, respectively. In terms of a parallel $\mathbb{F}_{p^2}$-multiplication performing four instances, there are currently the $(4 \times 2 \times 1)$-way and $(4 \times 1 \times 2)$-way options. Note that, although the former option does not use Karatsuba at the $\mathbb{F}_{p^2}$-layer, its underlying (8×1)-way integer multiplication is implemented with Karatsuba's algorithm. On the other hand, the $(4 \times 1 \times 2)$-way option uses Karatsuba at the $\mathbb{F}_{p^2}$-layer, whereas the (4×2)-way integer multiplication is simply a vectorized schoolbook multiplication.

Algorithm 5: $\mathbb{F}_{p^2}$-multiplication with 4-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0)i$.

1 $tt_1 \leftarrow a_0 \times b_0$ $\quad ss_1 \leftarrow a_0 \times b_1$ $\quad tt_2 \leftarrow a_1 \times b_0$ $\quad ss_2 \leftarrow a_1 \times b_1$
2 $t_1 \leftarrow tt_1 \bmod p$ $\quad s_1 \leftarrow ss_1 \bmod p$ $\quad t_2 \leftarrow tt_2 \bmod p$ $\quad s_2 \leftarrow ss_2 \bmod p$
3 $r_0 \leftarrow t_1 - s_2$ $\quad r_1 \leftarrow t_2 + s_1$
4 **return** $r = r_0 + r_1 i$

4-way Parallelization at $\mathbb{F}_p$-level. Algorithm 5 illustrates our 4-way $\mathbb{F}_{p^2}$-multiplication variant. Since the $\mathbb{F}_{p^2}$-multiplication using schoolbook involves four multiplications, it is possible to execute them all in parallel. Performing the Montgomery reductions before the operations at line 3 halves the length of the operands, which means the addition and subtraction at line 3 are single-length operations and can use the fast reduction modulo $2p$ we briefly described in Section 3.1. For a parallel $\mathbb{F}_{p^2}$-multiplication performing two instances, this 4-way variant could be used to develop a $(2 \times 4 \times 1)$-way implementation based on the optimal (8×1)-way prime-field operations.

4.2 $\mathbb{F}_{p^2}$-Squaring

A conventional squaring operation in $\mathbb{F}_{p^2}$ with the operand $a = a_0 + a_1 i$ is computed as $r = a^2 = (a_0^2 - a_1^2) + 2a_0a_1 i = r_0 + r_1 i$, which means two conventional integer squarings and an integer multiplication are required. A classic optimization, which is also used in SIDHv3.4, is to replace $a_0^2 - a_1^2$ by $(a_0 + a_1)(a_0 - a_1)$, i.e. two squaring operations are substituted by one multiplication. Since the conventional $\mathbb{F}_{p^2}$-squaring consists of three integer multiplication (or squaring) operations, and the optimized version still involves two integer multiplications, the 4-way variant for $\mathbb{F}_{p^2}$-squaring is not efficient. Thus, we only present a 1-way and a 2-way variant for $\mathbb{F}_{p^2}$-squaring, both of which take advantage of the optimization described above.

1-way Parallelization at $\mathbb{F}_p$-level. The $(x \times 1 \times z)$-way parallel $\mathbb{F}_{p^2}$ squaring is implemented according to Algorithm 6. Using (8×1)-way and (4×2)-way $\mathbb{F}_p$-arithmetic, we developed a $(8 \times 1 \times 1)$-way and a $(4 \times 1 \times 2)$-way version of $\mathbb{F}_{p^2}$-squaring, respectively.

Algorithm 6: $\mathbb{F}_{p^2}$-squaring with 1-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ element $a = a_0 + a_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 + a_1)(a_0 - a_1) + 2a_0a_1 i$.

1 $t_1 \leftarrow a_0 + a_1$	4 $tt_1 \leftarrow t_1 \times t_2$	7 $r_1 \leftarrow tt_2 \bmod p$
2 $t_2 \leftarrow a_0 - a_1$	5 $tt_2 \leftarrow t_3 \times a_1$	8 **return** $r = r_0 + r_1 i$
3 $t_3 \leftarrow a_0 + a_0$	6 $r_0 \leftarrow tt_1 \bmod p$	

2-way Parallelization at $\mathbb{F}_p$-level. The 2-way variant for squaring in $\mathbb{F}_{p^2}$ is specified in Algorithm 7, whereby a "perfect" parallelization is not possible at line 2. Based on this algorithm, we developed $(4 \times 2 \times 1)$-way and $(2 \times 2 \times 2)$-way $\mathbb{F}_{p^2}$-squaring with our two different implementations of the prime-field operations.

Algorithm 7: $\mathbb{F}_{p^2}$-squaring with 2-way parallelization at $\mathbb{F}_p$-level.

Input: $\mathbb{F}_{p^2}$ element $a = a_0 + a_1 i$.
Output: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 + a_1)(a_0 - a_1) + 2a_0a_1 i$.

1 $t_1 \leftarrow a_0 + a_1$	$s_1 \leftarrow a_0 + a_0$
2 $t_2 \leftarrow a_0 - a_1$	
3 $tt_1 \leftarrow t_1 \times t_2$	$ss_1 \leftarrow s_1 \times a_1$
4 $r_0 \leftarrow tt_1 \bmod p$	$r_1 \leftarrow ss_1 \bmod p$
5 **return** $r = r_0 + r_1 i$	

4.3 $\mathbb{F}_{p^2}$-Addition and Subtraction

A vectorized implementation of $\mathbb{F}_{p^2}$-addition/subtraction is fairly straightforward. The addition $r = a + b = (a_0 + a_1 i) + (b_0 + b_1 i) = (a_0 + b_0) + (a_1 + b_1)i = r_0 + r_1 i$ in $\mathbb{F}_{p^2}$ is

composed of two additions in $\mathbb{F}_p$. Our $(x \times 2 \times z)$-way implementation executes just the two additions in parallel, while the $(x \times 1 \times z)$-way version performs them sequentially one after the other. The $\mathbb{F}_{p^2}$-subtraction is vectorized in the same way.

4.4 Results and Comparison

The execution times of $\mathbb{F}_{p^2}$-multiplication and $\mathbb{F}_{p^2}$-squaring for SIKEp503 are shown in Table 2. We used our "Ice Lake" CPU to measure the execution times of AvxSike and the Microsoft SIDHv3.4 x64 assembly library, while the timings of Kostic and Gueron's IFMA implementation were taken from [KG19]. As explained in [KG19], they used the schoolbook method instead of Karatsuba's algorithm to develop their $\mathbb{F}_{p^2}$-multiplication and $\mathbb{F}_{p^2}$-squaring in order to mitigate the overhead caused by conversions between the radix-2^{52} vector representation and the radix-2^{64} big-integer representation. The results in Table 2 show that our vectorized implementations are more efficient than [KG19] and SIDHv3.4 when considering the "cycles/instance" metric. Furthermore, according to the measured timings, the $(4 \times 2 \times 1)$-way version is faster than the $(4 \times 1 \times 2)$-way version for both $\mathbb{F}_{p^2}$-multiplication and squaring, which means vectorization at the $\mathbb{F}_p$-layer has more impact on the performance than vectorization at the $\mathbb{F}_{p^2}$-layer. Furthermore, the $(2 \times 4 \times 1)$-way $\mathbb{F}_{p^2}$-multiplication requires fewer cycles than the $(2 \times 2 \times 2)$-way version because (8×1)-way integer multiplication is much more efficient than (4×2)-way, while the two vectorized reduction variants have similar latency (see Table 1). However, there is no $(2 \times 4 \times 1)$-way $\mathbb{F}_{p^2}$-squaring, which means we have to use the $(2 \times 4 \times 1)$-way $\mathbb{F}_{p^2}$-multiplication also for squaring. As a result, when implementing curve arithmetic with $(2 \times y \times z)$-way $\mathbb{F}_{p^2}$-operations, the $(2 \times 4 \times 1)$-way version has faster $\mathbb{F}_{p^2}$-multiplication (by 22 cycles) but slower $\mathbb{F}_{p^2}$-squaring (by 30 cycles) than the $(2 \times 2 \times 2)$-way version. In light of these results, we can not decide yet whether $(2 \times 4 \times 1)$-way or $(2 \times 2 \times 2)$-way is the more efficient option for the higher layers (see Section 5 for further discussions).

Table 2: Experimental results of $\mathbb{F}_{p^2}$-arithmetic implementations for SIKEp503.

Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
$\mathbb{F}_{p^2}$ Multiplication	SIDHv3.4	x64 asm	1-way	1	503	503	1.00×
	[KG19]	AVX-512	hybrid	1	282	282	1.78×
	AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	900	113	4.47×
	AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	570	143	3.53×
	AvxSike	AVX-512	$(4 \times 1 \times 2)$-way	4	684	171	2.94×
	AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	439	220	2.29×
	AvxSike	AVX-512	$(2 \times 4 \times 1)$-way	2	395	198	2.55×
$\mathbb{F}_{p^2}$ Squaring	SIDHv3.4	x64 asm	1-way	1	427	427	1.00×
	[KG19]	AVX-512	hybrid	1	287	287	1.49×
	AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	666	83	5.13×
	AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	380	95	4.49×
	AvxSike	AVX-512	$(4 \times 1 \times 2)$-way	4	576	144	2.97×
	AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	307	168	2.78×

5 Montgomery Elliptic Curve Arithmetic

In this section, we define a "$(w \times x \times y \times z)$-way" parallel Montgomery-curve arithmetic implementation: it performs w curve operations (e.g. point doublings, point triplings) in parallel, whereby each of them executes x $\mathbb{F}_{p^2}$-operations simultaneously, and each of the $\mathbb{F}_{p^2}$-operations performs y prime-field operations in parallel, and each of the prime-field

operations uses z 64-bit elements of a vector. Further, given an elliptic curve $E_{(A:C)}$ in Montgomery form, we define the three constants $A_{24}^{+} = A + 2C$, $A_{24}^{-} = A - 2C$, as well as $C_{24} = 4C$, which are used in the isogeny computations and the point arithmetic.

5.1 Three-Point Ladder

SIKE uses the *three-point ladder* algorithm from [FLOR18] as standard way to compute the kernel generator $R \leftarrow P + [k]Q$. For each bit of the scalar k, this algorithm performs a so-called *Montgomery ladder step* (`xDBLADD`), which essentially consists of a differential point addition and a point doubling; both operations are carried out using (projective) X and Z coordinates only, i.e. the Y-coordinate is not needed. The ladder step executes a fixed operation (resp. instruction) sequence, which means the three-point ladder has constant run-time. Various papers in the literature describe vectorized implementations of the Montgomery ladder-step, focusing particularly on reducing the latency of X25519 key exchange [Ber06]. For example, [Cho16, FL15, FLD19] discuss how to vectorize the Montgomery ladder-step in a 2-way fashion, while 4-way parallel implementations were presented in [CS09, HEY20, NS20]. The benchmarking results in [NS20, Table 2] and in [HEY20, Table 1] clearly indicate that the 4-way vectorized Montgomery ladder is more efficient than the 2-way variant on both AVX2 and AVX-512, and hence we decided to focus on the vectorization of the ladder-step for AvxSike-LL in 4-way fashion, or more precisely, by adopting $(1 \times 4 \times y \times z)$-way parallelization. In addition, Table 2 suggests that the $(4 \times 2 \times 1)$-way $\mathbb{F}_{p^2}$-operations are faster than the $(4 \times 1 \times 2)$-way versions, and so we finally chose the $(1 \times 4 \times 2 \times 1)$-way implementation for the ladder-step.

In [NS20], Nath and Sarkar analyze in detail four different methods to vectorize the Montgomery ladder step in 4–way fashion (one from [CS09], one from [HEY20], and two developed by themselves) and compared in [NS20, Table 1] the operations that all these methods have to perform. The methods presented in [CS09] and in [NS20] require three 4-way vectorized field-multiplications (resp. field-squarings), plus a special multiplication by a small constant in each step of the ladder. This special multiplication computes the product of a field-element and the coefficient of Curve25519, which is much faster than a conventional field multiplication. However, for some scalar multiplications of SIKE, the coefficient of the Montgomery curve is not a small constant but an element of $\mathbb{F}_{p^2}$. As a consequence, the vectorization of [CS09] and [NS20] will, in the case of SIKE, require four vectorized $\mathbb{F}_{p^2}$-multiplication (or $\mathbb{F}_{p^2}$-squaring) operations. On the other hand, the vectorization technique presented by Hisil, Egrice, and Yassi in [HEY20] needs only two vectorized field multiplications and one vectorized field squaring in each ladder step (no special multiplication by a small curve-constant is carried out). Thus, the vectorization proposed in [HEY20] is the better choice for a low-latency SIKE implementation.

Based on the 4-way vectorization from [HEY20] and our $(4 \times 2 \times 1)$-way $\mathbb{F}_{p^2}$-operations, we developed a $(1 \times 4 \times 2 \times 1)$-way parallel ladder step for the latency-optimized AvxSike-LL. In addition, to speed up the SIPKE encryption operation (Algorithm 15) of AvxSike-LL, we implemented a $(2 \times 4 \times 1 \times 1)$-way ladder step for two simultaneous scalar multiplications, which uses the efficient $(8 \times 1 \times 1)$-way $\mathbb{F}_{p^2}$-operations. For the throughput-oriented AvxSike-HT software, we developed a $(8 \times 1 \times 1 \times 1)$-way parallel ladder step according to the batched X25519 implementation described in [CGT+21].

5.2 Point Doubling and Tripling

Given a point $P = (X_P : Z_P)$ on the Montgomery curve $E_{(A:C)}$, we define the double $[2]P = (X_{[2]P} : Z_{[2]P})$ as:

$$X_{[2]P} = C_{24}(X_P^2 - Z_P^2)^2 \quad \text{and} \quad Z_{[2]P} = 4X_P Z_P \left(4A_{24}^{+} X_P Z_P + C_{24}(X_P - Z_P)^2\right).$$

Algorithm 8 shows our 2-way parallel implementation of the doubling operation. For the tripling, we took advantage of a new formula that uses the value C_{24} instead of A^-_{24} as in the original `xTPL` function from the SIKE specification (Algorithm 6 in [JAC+20]) since the latter is less efficient for 2-way vectorization. Given $P = (X_P : Z_P)$ on the curve $E_{(A:C)}$, we compute the point $[3]P = (X_{[3]P} : Z_{[3]P})$ via the formulae:

$$X_{[3]P} = X_P \left[C_{24} \left(X_P^2 - Z_P^2 \right)^2 - 4Z_P^2 \left(4A^+_{24} X_P Z_P + C_{24}(X_P - Z_P)^2 \right) \right]^2$$
$$Z_{[3]P} = Z_P \left[C_{24} \left(X_P^2 - Z_P^2 \right)^2 - 4X_P^2 \left(4A^+_{24} X_P Z_P + C_{24}(X_P - Z_P)^2 \right) \right]^2 .$$

The 2-way implementation of our point-tripling function is described in Algorithm 9.

Algorithm 8: XZ-coordinate point doubling with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P = (X_P : Z_P)$, $(A^+_{24} : C_{24})$.
Output: $Q = [2]P = (X_Q : Z_Q)$.

1	$t_0 \leftarrow X_P + Z_P$	$s_0 \leftarrow X_P - Z_P$
2	$t_1 \leftarrow t_0^2$	$s_1 \leftarrow s_0^2$
3	$t_0 \leftarrow t_1 + s_1$	$s_0 \leftarrow t_1 - s_1$
4	$t_2 \leftarrow C_{24} \times s_1$	$s_2 \leftarrow A^+_{24} \times s_0$
5	$t_3 \leftarrow t_2 + s_2$	
6	$X_Q \leftarrow t_2 \times t_1$	$Z_Q \leftarrow t_3 \times s_0$
7	**return** $Q = (X_Q : Z_Q)$	

Algorithm 9: XZ-coordinate point tripling with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P = (X_P : Z_P)$, $(A^+_{24} : C_{24})$.
Output: $Q = [3]P = (X_Q : Z_Q)$.

1	$t_0 \leftarrow X_P + Z_P$	$s_0 \leftarrow X_P - Z_P$
2	$t_1 \leftarrow t_0^2$	$s_1 \leftarrow s_0^2$
3	$t_0 \leftarrow X_P + X_P$	$s_0 \leftarrow t_1 - s_1$
4	$t_2 \leftarrow t_1 + t_1$	$s_2 \leftarrow s_0 + s_0$
5	$t_3 \leftarrow C_{24} \times s_1$	$s_3 \leftarrow A^+_{24} \times s_0$
6	$t_0 \leftarrow t_3 \times t_1$	$s_0 \leftarrow t_0 \times t_0$
7	$t_1 \leftarrow t_2 + t_2$	$s_1 \leftarrow s_0 + s_2$
8	$t_3 \leftarrow t_3 + s_3$	$s_3 \leftarrow t_1 - s_1$
9	$t_4 \leftarrow s_3 \times t_3$	$s_4 \leftarrow s_0 \times t_3$
10	$t_4 \leftarrow t_0 - t_4$	$s_4 \leftarrow t_0 - s_4$
11	$t_4 \leftarrow t_4^2$	$s_4 \leftarrow s_4^2$
12	$X_Q \leftarrow X_P \times t_4$	$Z_Q \leftarrow Z_P \times s_4$
13	**return** $Q = (X_Q : Z_Q)$	

5.3 Isogeny Generation

Recall that in SIKE the 2^{e_2}-isogeny and the 3^{e_3}-isogeny are computed as a composition of 4-isogenies and 3-isogenies, respectively. Given a point $R_4 = (X_4 : Z_4)$ of order 4 on $E_{(A:C)}$, a 4-isogeny $\phi_4 : E_{(A:C)} \to E_{(A':C')}$ is constructed with $\ker \phi_4 = \langle R_4 \rangle$. Algorithm 10 describes our 2-way implementation of such a 4-isogeny computation. This algorithm outputs the two parameters $A^+_{24} = A' + 2C'$ and $C_{24} = 4C'$ that define the target curve $E_{(A':C')}$ (where $A^+_{24} = 4X_4^4$, $C_{24} = 4Z_4^4$) and the values $K_0 = 4Z_4^2$, $K_1 = X_4 - Z_4$, and $K_2 = X_4 + Z_4$, which are used when evaluating the 4-isogeny ϕ_4 at a point. When the point $R_3 = (X_3 : Z_3)$ on $E_{(A:C)}$ is of order 3, a 3-isogeny $\phi_3 : E_{(A:C)} \to E_{(A':C')}$ has to be constructed with $\ker \phi_3 = \langle R_3 \rangle$. Algorithm 11 shows our 2-way implementation of the function to compute an isogeny of degree 3. The 3-isogeny generation algorithm outputs the values $A^+_{24} = A' + 2C'$, $C_{24} = 4C'$[6] that define the target curve $E_{(A':C')}$, namely:

$$A^+_{24} = (3X_3^2 - 2X_3Z_3 - Z_3^2)(3X_3 + Z_3)^2 \quad \text{and} \quad C_{24} = -16X_3Z_3^3.$$

The algorithm also outputs the constants $K_1 = X_3 - Z_3$ and $K_2 = X_3 + Z_3$, which will be used in the evaluation of a 3-isogeny at a point.

[6] The 3-isogeny generation algorithm in the SIKE specification (Algorithm 15 in [JAC+20]) originally outputs A^-_{24}. Our formula outputs C_{24} instead of A^-_{24} since our point tripling takes C_{24} as input.

Algorithm 10: 4-isogeny computation with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P_4 = (X_4 : Z_4)$.
Output: $(A_{24}^+ : C_{24})$, $(K_0 : K_1 : K_2)$.

1	$K_2 \leftarrow X_4 + Z_4$	$K_1 \leftarrow X_4 - Z_4$
2	$t_0 \leftarrow Z_4^2$	$s_0 \leftarrow X_4^2$
3	$t_0 \leftarrow t_0 + t_0$	$s_0 \leftarrow s_0 + s_0$
4	$C_{24} \leftarrow t_0^2$	$A_{24}^+ \leftarrow s_0^2$
5	$K_0 \leftarrow t_0 + t_0$	
6	**return** $(A_{24}^+ : C_{24}), (K_0 : K_1 : K_2)$	

Algorithm 11: 3-isogeny computation with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P_3 = (X_3 : Z_3)$.
Output: $(A_{24}^+ : C_{24})$, $(K_1 : K_2)$.

1	$K_2 \leftarrow X_3 + Z_3$	$K_1 \leftarrow X_3 - Z_3$
2	$t_0 \leftarrow K_2^2$	$s_0 \leftarrow K_1^2$
3	$t_1 \leftarrow X_3 + X_3$	$s_1 \leftarrow s_0 - t_0$
4	$t_2 \leftarrow t_1^2$	$s_2 \leftarrow Z_3^2$
5	$t_1 \leftarrow t_2 + t_2$	$s_1 \leftarrow s_1 + s_1$
6	$t_3 \leftarrow t_1 - s_1$	$s_3 \leftarrow s_2 + s_2$
7	$t_4 \leftarrow t_3 + s_0$	$s_4 \leftarrow t_2 - t_0$
8	$C_{24} \leftarrow s_1 \times s_3$	$A_{24}^+ \leftarrow t_4 \times s_4$
9	**return** $(A_{24}^+ : C_{24}), (K_1 : K_2)$	

5.4 Isogeny Evaluation

Let $\phi_4 : E_{(A:C)} \rightarrow E_{(A':C')}$ be a 4-isogeny with kernel $\ker \phi_4 = \langle (X_4 : Z_4) \rangle$ and let the point $P = (X_P : Z_P)$ be on curve $E_{(A:C)}$. Then, the point $P' = \phi_4(P) = (X_{P'} : Z_{P'})$ is derived after the evaluation of the 4-isogeny ϕ_4 at P and defined as:

$$X_{P'} = 16\left((X_4X_P - Z_4Z_P)^2 + Z_4^2(X_P^2 - Z_P^2)\right)(X_4X_P - Z_4Z_P)^2$$
$$Z_{P'} = 16\left((X_4Z_P - Z_4X_P)^2 - Z_4^2(X_P^2 - Z_P^2)\right)(X_4Z_P - Z_4X_P)^2$$

Our 2-way implementation for the 4-isogeny evaluation is specified in Algorithm 12. In the case of 3-isogeny, let $\phi_3 : E_{(A:C)} \rightarrow E_{(A':C')}$ with kernel $\ker \phi_3 = \langle (X_3 : Z_3) \rangle$ and $P = (X_P : Z_P)$ be a point on curve $E_{(A:C)}$. Then, the image of P under the 3-isogeny ϕ_3 is a point $P' = \phi_3(P) = (X_{P'} : Z_{P'})$ such that:

$$X_{P'} = 4X_P\left(X_3X_P - Z_3Z_P\right)^2 \quad \text{and} \quad Z_{P'} = 4Z_P\left(X_3Z_P - Z_3X_P\right)^2. \tag{3}$$

The 2-way implementation of the 3-isogeny evaluation is presented in Algorithm 13.

Algorithm 12: 4-isogeny evaluation with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P = (X_P : Z_P)$, $(K_0 : K_1 : K_2)$.
Output: $P' = \phi_4(P) = (X_{P'} : Z_{P'})$.

1	$t_0 \leftarrow X_P + Z_P$	$s_0 \leftarrow X_P - Z_P$
2	$t_1 \leftarrow K_1 \times t_0$	$s_1 \leftarrow t_0 \times s_0$
3	$t_2 \leftarrow K_2 \times s_0$	$s_2 \leftarrow K_0 \times s_1$
4	$t_0 \leftarrow t_2 + t_1$	$s_0 \leftarrow t_2 - t_1$
5	$t_0 \leftarrow t_0^2$	$s_0 \leftarrow s_0^2$
6	$t_1 \leftarrow t_0 + s_2$	$s_1 \leftarrow s_0 - s_2$
7	$X_{P'} \leftarrow t_1 \times t_0$	$Z_{P'} \leftarrow s_1 \times s_0$
8	**return** $(X_{P'} : Z_{P'})$	

Algorithm 13: 3-isogeny evaluation with 2-way parallelization at $\mathbb{F}_{p^2}$-level.

Input: $P = (X_P : Z_P)$, $(K_1 : K_2)$.
Output: $P' = \phi_3(P) = (X_{P'} : Z_{P'})$.

1	$t_0 \leftarrow X_P + Z_P$	$s_0 \leftarrow X_P - Z_P$
2	$t_0 \leftarrow K_1 \times t_0$	$s_0 \leftarrow K_2 \times s_0$
3	$t_1 \leftarrow t_0 + s_0$	$s_1 \leftarrow t_0 - s_0$
4	$t_1 \leftarrow t_1^2$	$s_1 \leftarrow s_1^2$
5	$X_{P'} \leftarrow X_P \times t_1$	$Z_{P'} \leftarrow Z_P \times s_1$
6	**return** $(X_{P'} : Z_{P'})$	

5.5 Results and Comparison

Because of dependencies among the involved operations, our AvxSike-LL only requires a $(1 \times x \times y \times z)$-way implementation of the point tripling and 3-isogeny generation. On the other hand, for the doubling operation and 4-isogeny generation, we can besides the

$(1 \times x \times y \times z)$-way implementation also use a $(2 \times x \times y \times z)$-way implementation in an optimized version of SIPKE encryption (see Algorithm 15). Table 3 indicates that the $(1 \times 2 \times 2 \times 2)$-way point-operations outperform their $(1 \times 2 \times 4 \times 1)$-way counterparts by a few clock cycles. In addition, the 4-isogeny computation (Algorithm 10) just uses $\mathbb{F}_{p^2}$-squaring but not $\mathbb{F}_{p^2}$-multiplication and, therefore, the $(1 \times 2 \times 4 \times 1)$-way version is clearly not efficient in this case. As a result, we chose $(1 \times 2 \times 2 \times 2)$-way parallelism to implement all the $(1 \times x \times y \times z)$-way operations for the curve arithmetic. According to Table 4, the difference (in terms of cycles/instance) between the $(8 \times 1 \times 1 \times 1)$-way and the $(4 \times 2 \times 1 \times 1)$-way parallelism is rather small. Both of these parallelization options are much more efficient than the $(2 \times 2 \times 2 \times 1)$-way and $(1 \times 2 \times 2 \times 2)$-way versions.

Table 3: Experimental results of point-operation implementations for SIKEp503.

Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
Ladder step (`xDBLADD`)	SIDHv3.4	x64 asm	1-way	1	5056	5056	1.00×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	9417	1177	4.30×
	AvxSike	AVX-512	$(2 \times 4 \times 1 \times 1)$-way	2	2880	1440	3.51×
	AvxSike	AVX-512	$(1 \times 4 \times 2 \times 1)$-way	1	1757	1757	2.88×
Point doubling (`xDBL`)	SIDHv3.4	x64 asm	1-way	1	2873	2873	1.00×
	[KG19]	AVX-512	hybrid	1	1782	1782	1.61×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	5052	632	4.55×
	AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	1660	830	3.46×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1273	1273	2.26×
	AvxSike	AVX-512	$(1 \times 2 \times 4 \times 1)$-way	1	1319	1319	2.18×
Point tripling (`xTPL`)	SIDHv3.4	x64 asm	1-way	1	5794	5794	1.00×
	[KG19]	AVX-512	hybrid	1	3527	3527	1.64×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	10063	1258	4.61×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	2730	2730	2.12×
	AvxSike	AVX-512	$(1 \times 2 \times 4 \times 1)$-way	1	2745	2745	2.11×

Table 4: Experimental results of isogeny-operation implementations for SIKEp503.

Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
4-isogeny generation (`get_4_isog`)	SIDHv3.4	x64 asm	1-way	1	1729	1729	1.00×
	[KG19]	AVX-512	hybrid	1	1379	1379	1.25×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	3113	389	4.44×
	AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	843	422	4.10×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	673	673	2.57×
4-isogeny evaluation (`eval_4_isog`)	SIDHv3.4	x64 asm	1-way	1	3852	3852	1.00×
	[KG19]	AVX-512	hybrid	1	2292	2292	1.68×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	6925	866	4.45×
	AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	3569	892	4.32×
	AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	2289	1145	3.36×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1858	1858	2.07×
3-isogeny generation (`get_3_isog`)	SIDHv3.4	x64 asm	1-way	1	2783	2783	1.00×
	[KG19]	AVX-512	hybrid	1	2011	2011	1.38×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	4508	564	4.93×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1350	1350	2.06×
3-isogeny evaluation (`eval_3_isog`)	SIDHv3.4	x64 asm	1-way	1	2893	2893	1.00×
	[KG19]	AVX-512	hybrid	1	1628	1628	1.78×
	AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	5130	641	4.51×
	AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	2651	663	4.36×
	AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	1521	761	3.80×
	AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1235	1235	2.34×

6 Higher Layers

6.1 Low-Latency Implementation

All functions in Algorithm 1 and 2 contain two types of costly operations, namely the computation of the kernel generator $R \leftarrow P + [k]Q$ and the generation/evaluation of the ℓ^{e_ℓ}-isogeny for $\ell \in \{2, 3\}$. The former is performed using the three-point ladder, whose efficient four-way vectorization has been explained in Section 5.1. On the other hand, as discussed in Section 2.2, SIKE takes advantage of the optimal strategies from [DJP14] to reduce the execution time of the ℓ^{e_ℓ}-isogeny generation and evaluation (these algorithms for $\ell \in \{2, 3\}$ are analyzed in detail in the SIKE specification, see [JAC+20, Algorithm 19 and 20]).

Algorithm 14: Vectorized `isogeny`$_2$ in SIKEp503 using the optimal strategies.

Input: Curve E_A, public parameter e_2, point (X_R, Z_R), and a *strategy* $(s_1, \ldots, s_{e_2/2-1})$.
Output: Curve E_B such that $\phi_2 : E_A \to E_B$ with $\deg \phi_2 = 2^{e_2}$, $\ker \phi_2 = \langle (X_R : Z_R) \rangle$.

$pts \leftarrow [\,]$, $i \leftarrow 0$, $E_B \leftarrow E_A$, $k \leftarrow 1$
for j **from** 1 **to** $e_2/2 - 1$ **by** 1 **do**
  while $i < e_2/2 - j$ **do**
    `push`(X_R, Z_R, i) to pts `// Append it to the end of pts`
    $e \leftarrow s_k$
    $(X_R, Z_R) \leftarrow$ `xDBLe_1x2x2x2w`$(X_R, Z_R, E_B, 2e)$ `//` $[2^{2e}]R$
    $i \leftarrow i + e$, $k \leftarrow k + 1$
  $E_B, \phi \leftarrow$ `get_4_isog_1x2x2x2w`(X_R, Z_R) `// 4-isogeny generation`
  $pts \leftarrow$ `eval_4_isog_parallel`(ϕ, pts) `// Parallel 4-isogeny evaluation`
  `pop`(X_S, Z_S, i_S) from pts `// Remove it from the end of pts`
  $X_R \leftarrow X_S$, $Z_R \leftarrow Z_S$, $i \leftarrow i_S$
$E_B, \phi \leftarrow$ `get_4_isog_1x2x2x2w`(X_R, Z_R) `// 4-isogeny generation`
return E_B

Vectorized ℓ^{e_ℓ}-isogeny Computation and Evaluation. We pick the 2^{e_2}-isogeny case as example to demonstrate our vectorized implementation (note that the 3^{e_3}-isogeny can be vectorized in a very similar fashion). It is common practice that, when e_2 is even, the 2^{e_2}-isogeny is computed as the composition of $e_2/2$ isogenies of degree 4, while an extra isogeny of degree 2 needs to be computed when e_2 is odd. The 2^{e_2}-isogeny generation for even e_2 is described in Algorithm 14. Because of dependencies among the operations in each iteration, we choose the $(1 \times 2 \times 2 \times 2)$-way parallel point doubling[7] to compute the kernel $[2^{2e}]R$ of the 4-isogeny, and the $(1 \times 2 \times 2 \times 2)$-way 4-isogeny generation to obtain the coefficients of the target curve. The vector $(s_1, \ldots, s_{e_2/2-1})$ denotes the tree traversal strategy used for fast isogeny computations. These strategies are described in [JAC+20, Appendix D] for the four parameter sets of our SIKE implementation. On the other hand, for the 4-isogeny evaluation at different points in the *pts* queue, we decided to develop a dedicated `eval_4_isog_parallel` function to achieve a fast simultaneous isogeny evaluation, which uses the more efficient $(8 \times 1 \times 1 \times 1)$-way and $(4 \times 2 \times 1 \times 1)$-way parallel implementations. This function checks at first the number of points in the *pts* queue and then uses the different vectorized implementations of 4-isogeny evaluation (i.e `eval_4_isog`) with corresponding points in *pts* to handle the computation[8]:

[7] $(X_Q, Z_Q) \leftarrow$`xDBLe`(X_P, Z_P, E, n) denotes computing $Q \leftarrow [2^n]P$ on curve E by using n times the point-doubling operation `xDBL`.

[8] Note that the number of points in the *pts* queue (i.e. $\#pts$) is public information. Hence, using the different vectorized 4-isogeny evaluation implementations does not leak any secrets.

$\#pts = 1 : (1 \times 2 \times 2 \times 2)$-way $\quad \#pts = 5 : (4 \times 2 \times 1 \times 1)$-way $+ (1 \times 2 \times 2 \times 2)$-way
$\#pts = 2 : (2 \times 2 \times 2 \times 1)$-way $\quad \#pts = 6 : (4 \times 2 \times 1 \times 1)$-way $+ (2 \times 2 \times 2 \times 1)$-way
$\#pts = 3 : (4 \times 2 \times 1 \times 1)$-way $\quad \#pts = 7 : (8 \times 1 \times 1 \times 1)$-way
$\#pts = 4 : (4 \times 2 \times 1 \times 1)$-way $\quad \#pts = 8 : (8 \times 1 \times 1 \times 1)$-way

Optimized SIKE Encapsulation. Even at the highest layer of SIKE, there are options to parallelize some internal operations. For example, we managed to further optimize the Encaps operation by parallelizing two scalar multiplications (computed with the same scalar sk_2) and parallelizing two 2^{e_2}-isogeny generation and evaluation operations, which are denoted as $\texttt{isogeny}_2$ in our optimized Enc function for SIPKE that is described in Algorithm 15. As stated in Section 5.1, the kernel generator in AvxSike-LL is obtained with help of the $(1 \times 4 \times 2 \times 1)$-way three-point ladder. However, Algorithm 15 computes in line 2 two kernel generators simultaneously, and therefore it is possible to use a more efficient ladder, namely the $(2 \times 4 \times 1 \times 1)$-way vectorized version. More importantly, in line 3, the algorithm performs two 2^{e_2}-isogeny generation and evaluation operations in parallel, which makes it possible to use the $(2 \times 2 \times 2 \times 1)$-way implementation for the point doubling and 4-isogeny generation instead of the $(1 \times 2 \times 2 \times 2)$-way version. This will lead to a significant difference in performance because the underlying $\mathbb{F}_p$-arithmetic implementation changes from (4×2)-way to (8×1)-way. According to our results, the optimized SIPKE encryption in Algorithm 15 improves the speed of a SIKE encapsulation by around 27% compared to the straightforward version of Enc (i.e. Algorithm 1).

Algorithm 15: The optimized SIPKE encryption operation.

1 **function** $\mathsf{EncOpt}(\text{pk}_3, m \in \mathcal{M}, \text{sk}_2 \in \mathcal{K}_2)$
2 $x_{R_2} \leftarrow x_{P_2} + [\text{sk}_2]x_{Q_2}$ $\quad x'_{R_2} \leftarrow \phi_3(x_{P_2}) + [\text{sk}_2]\phi_3(x_{Q_2})$
3 $(\phi_2, E_2) \leftarrow \texttt{isogeny}_2(E_0, x_{R_2})$ $\quad (\phi'_2, E_{32}) \leftarrow \texttt{isogeny}_2(E_3, x'_{R_2})$
4 $c_1 \leftarrow (\phi_2(x_{P_3}), \phi_2(x_{Q_3}), \phi_2(x_{PQ_3}))$
5 $h \leftarrow \texttt{SHAKE256}(j(E_{32}))$
6 $c_2 \leftarrow h \oplus m$
7 **return** (c_1, c_2)

6.2 High-Throughput Implementation

Since constant-time SIKE executes a fixed operation sequence, a batched implementation using AVX-512 is fairly easy to develop. We modified the SIDHv3.4 x64 implementation by using our $(8 \times 1 \times 1 \times 1)$-way curve arithmetic, $(8 \times 1 \times 1)$-way $\mathbb{F}_{p^2}$-operations, and (8×1)-way $\mathbb{F}_p$-operations at the different layers. We also developed a $(8 \times 1 \times 1 \times 1)$-way version of some other subroutines of AvxSike-HT, most notably the computation of the j-invariant (`j_inv`) and the curve coefficient (`get_A`), as well as a 3-way simultaneous inversion (`inv_3_way`), which takes advantage of the (8×1)-way $\mathbb{F}_p$-inversion.

Parallel SHAKE256. As explained in Section 2, the hash function used by SIKE is an XOF, namely `SHAKE256`, which employs the Keccak permutation. For AvxSike-HT, we developed a batched `SHAKE256` implementation based on AVX-512 instructions that can process eight inputs independently and in parallel. The eXtended Keccak Code Package (XKCP) contains various Keccak-related software artifacts, including highly-optimized implementations of the Keccak permutation for two generations of AVX, namely AVX2 and AVX-512[9]. Sinha Roy showed in [Sin19] that the AVX2-based Keccak source code from XKCP can serve as a starting point to build a batched `SHAKE256` implementation capable to process four inputs simultaneously, each using a 64-bit slot of a 256-bit AVX2

[9] https://github.com/XKCP/XKCP/tree/master/lib/low/KeccakP-1600-times8/AVX512

vector. We followed Sinha Roy's idea and used the AVX-512 implementation of Keccak from the XKCP to batch `SHAKE256` with AVX-512 instructions.

7 Experimental Results

Table 5 shows the cycle counts for the different SIKEp503 implementations. The timings for SIDHv3.4 and AvxSike are measured on our target Ice Lake CPU, while the results for the implementation reported in [KG19] are taken from the paper. As the first three implementations in Table 5 are designed to reduce the latency, we can compare them in terms of speed, while AvxSike-HT aims at increasing throughput and, thus, it makes sense to compare it with SIDHv3.4 in terms of throughput. Regarding key generation and decapsulation, AvxSike-LL is about 2.5 times faster than SIDHv3.4 and outperforms the implementation of [KG19] by a factor of 1.5. Furthermore, due to our optimizations for encapsulation (see Section 6.1), AvxSike-LL reaches a 3.2-fold higher encapsulation speed compared to SIDHv3.4, which can be beneficial for e.g. server-side TLS processing since, when SIKE is integrated into TLS, the server has to perform encapsulations. TLS servers could profit even more from our high-throughput AvxSike-HT implementation because it outperforms SIDHv3.4 throughput-wise by a factor of almost 4.6.

Table 5: Execution times (in cycles) of implementations of SIKEp503 on an Intel Core i3-1005G1 processor. The cycle-counts for SIDHv3.4, Kostic-Gueron's work [KG19], and AvxSike-LL are for the execution of one instance of an operation and "Speed-up" is the speed-up factor compared to SIDHv3.4. The cycle-counts for AvxSike-HT are for the execution of *eight* instances of an operation and "Throughput" is the throughput gain compared to SIDHv3.4 when it executes eight instances.

Operation	SIDHv3.4 (1 instance)	Kostic [KG19] (1 instance)		AvxSike-LL (1 instance)		AvxSike-HT (8 instances)	
	Cycles	Cycles	Speed-up	Cycles	Speed-up	Cycles	Throughput
KeyGen	8,078,669	4,842,909	1.67×	3,215,375	2.51×	14,179,026	4.56×
Encaps	13,188,788	7,923,514	1.66×	4,111,650	3.21×	22,992,807	4.59×
Decaps	14,026,750	8,513,409	1.65×	5,715,005	2.45×	24,619,263	4.56×

An x64 implementation of SIKE executed on an Ice Lake Core has to use one single $(64 \times 64 \rightarrow 128)$-bit multiplier sequentially, whereas AVX-512IFMA is able to perform eight parallel $(52 \times 52 + 64 \rightarrow 64)$-bit multiply-add operations. But this does not mean that IFMA instructions can lead to an (almost) eight-fold performance gain, not even in theory. Though the IFMA engine can carry out eight element-wise multiplications simultaneously, various other architectural and micro-architectural features and effects have to be considered, e.g. different multiplier widths (52 vs. 64 bits), different carry chains and other sequential dependencies, different instruction latencies and throughputs, as well as differences in the register space and occupation[10]. For all these reasons, the theoretical speed-up factor of an IFMA implementation compared to an x64 implementation like SIDHv3.4 is far from eight and very difficult to estimate. Kostic and Gueron focused in [KG19] on optimizing the $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ layer of SIKE, especially the multiplication of field elements, using AVX-512IFMA and achieved a reduction of the overall execution time by a factor of roughly 1.7 (these results still represent the speed record for SIKE on an Intel CPU). On the other hand, AvxSike takes advantage of sophisticated vectorization of the higher layers of SIKE, in addition to notably more efficient vectorized prime-field

[10] The SIDHv3.4 library uses a full-radix representation (64 bits/limb) for field elements, which enables a 100% occupation of the registers (except for the register with the highest limb), while our 51 bits/limb representation implies that around 20% of each 64-bit element of an AVX-512 register is empty.

Table 6: Execution times (in cycles) of implementations of SIKEp434, SIKEp610, and SIKEp751 on an Intel Core i3-1005G1 processor. The cycle-counts for SIDHv3.4 and AvxSike-LL are for the execution of one instance of an operation and "Speed-up" is the speed-up factor compared to SIDHv3.4. The cycle-counts for AvxSike-HT are for the execution of *eight* instances of an operation and "Throughput" is the throughput gain compared to SIDHv3.4 when it executes eight instances.

Scheme	Operation	SIDHv3.4 (1 instance)	AvxSike-LL (1 instance)		AvxSike-HT (8 instances)	
		Cycles	Cycles	Speed-up	Cycles	Throughput
SIKEp434	KeyGen	5,976,700	2,474,187	2.42×	10,442,609	4.58×
	Encaps	9,690,764	3,062,491	3.16×	16,801,041	4.61×
	Decaps	10,357,218	4,341,099	2.39×	18,053,398	4.59×
SIKEp610	KeyGen	14,096,085	6,918,618	2.04×	32,172,538	3.51×
	Encaps	25,875,968	10,001,282	2.59×	58,747,976	3.52×
	Decaps	26,040,095	13,124,052	1.98×	59,103,361	3.52×
SIKEp751	KeyGen	23,843,419	10,212,410	2.33×	46,662,723	4.09×
	Encaps	38,446,643	12,804,923	3.00×	74,885,499	4.11×
	Decaps	41,368,995	17,834,974	2.32×	80,684,214	4.10×

arithmetic. Thanks to careful optimizations at the higher layers, AvxSike-LL reaches 1.5 times faster execution times for both key generation and decapsulation compared to [KG19] (see Table 5). Furthermore, the encapsulation is almost two times faster.

The benchmarking results of Microsoft's SIDHv3.4 and AvxSike for the other three parameter sets are given in Table 6. When analyzing the cycle counts achieved by the low-latency version AvxSike-LL, it is apparent that the speed-up factors (in relation to SIDHv3.4) for SIKEp434 and SIKEp751 are similar to the speed-up for SIKEp503, though a bit smaller. This reduced performance gain can be explained by the number of limbs needed for the elements of a 434-bit and 751-bit prime field, respectively. Namely, due to the radix-2^{51} representation, the number of limbs is *odd* in both the 434-bit case (nine limbs) and the 751-bit case (15 limbs), whereas it is *even* for the 503-bit field. An odd number of limbs is not ideal for the structure of (4×2)-way limb vector set (since there are four elements unused in the last limb vector) and causes an "underutilization" of the parallelism in many of the executed AVX-512 vector instructions. Furthermore, also the register allocation (i.e. how many operands and results can be kept in registers) impacts the overall performance, which, in turn, depends on the length of the field elements. In general, the larger the order of the underlying prime field, the more difficult it becomes to keep operands in the register file and the more register spills will happen in both the low-latency and the high-throughput version, respectively. While the speed-up factors for SIKEp434 and SIKEp751 are only marginally smaller than that for SIKEp503, it turns out that the gap between SIKEp610 and SIKEp503 is bigger. Even though the elements of a 610-bit field can be represented by an even number of limbs ($51 \times 12 = 612$), there are only two bits of "headroom" in this representation, which is not ideal with respect to lazy reduction. Concretely, when using the SIKEp610 parameters, a number of additional modular reductions have to be carried out to prevent overflows, which impacts both the low-latency version and the high-throughput version of AvxSike.

8 Conclusions

Vector processing engines like Intel's AVX offer a great potential to reduce the execution time (or increase the throughput) of public-key cryptosystems, and this is also the case

for post-quantum KEMs such as SIKE. The AVX-512IFMA instructions deserve special attention because they allow one to execute eight multiplications of 52-bit operands in parallel, followed by a parallel addition of the upper or lower halves of the products to eight 64-bit operands. By developing sophisticated vector processing techniques for field arithmetic, point arithmetic, and isogeny computations, all of which are integrated into our AvxSike software, we were able to significantly improve both the latency and the throughput of SIKE on modern Intel processors. For example, AvxSike-LL instantiated with the SIKEp503 parameters is about 1.5 times faster than the AVX-512IFMA-based SIKE implementation described in [KG19] and outperforms Microsoft's highly-optimized SIDHv3.4 library by a factor of roughly 2.5 for key generation and decapsulation, while the speed-up factor for the encapsulation reaches even 3.2. In summary, AvxSike does not only set new software speed and throughput records for SIKE on Intel CPUs, but also narrows the gap between SIKE and lattice-based post-quantum KEMs, mainly because the IFMA instructions are more beneficial for the multiplication in prime fields than the multiplication in the polynomial rings of e.g. NTRU, Kyber, or Saber.

We envision that follow-up work in two directions can yield interesting results. The first direction concerns the integration of AvxSike into an existing SSL/TLS protocol stack like OpenSSL to evaluate the impact of the latency-optimized implementation on the side of the client and the impact of the throughput-optimized implementation on the server side. In particular, it would be interesting to figure out to what extent the speed and throughput improvements of AvxSike propagate up to the protocol layer. A second research direction could target the question of how beneficial the presented vectorization techniques can be for compressed SIKE. For example, the public-key compression process described in [JAC+20] requires the execution of some very expensive operations, such as pairing and discrete logarithm computations over $\mathbb{F}_{p^2}$. Since these operations constitute the main bottleneck in the compression algorithm, it can be expected that utilizing the parallel processing power of AVX-512IFMA has the potential to significantly accelerate the overall execution of compressed SIKE.

Acknowledgements. We are grateful to all reviewers for their valuable comments and suggestions. We would also like to thank Patrick Longa and Luca De Feo for their helpful insights. This work was supported by the Luxembourg National Research Fund (FNR) CORE project EquiVox (C19/IS/13643617/EquiVox/Ryan).

References

[AAM21] Mila Anastasova, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Fast strategies for the implementation of SIKE round 3 on ARM Cortex-M4. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(10):4129–4141, October 2021.

[ABJK18] Reza Azarderakhsh, Elena Bakos Lang, David Jao, and Brian Koziel. EdSIDH: Supersingular isogeny Diffie-Hellman key exchange on Edwards curves. In Anupam Chattopadhyay, Chester Rebeiro, and Yuval Yarom, editors, *Security, Privacy, and Applied Cryptography Engineering — SPACE 2018*, volume 11348 of *Lecture Notes in Computer Science*, pages 125–141. Springer Verlag, 2018.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography — PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Verlag, 2006.

[BF20] Joppe W. Bos and Simon J. Friedberger. Faster modular arithmetic for isogeny-based crypto on embedded devices. *Journal of Cryptographic Engineering*, 10(2):97–109, June 2020.

[BI21] Cyril Bouvier and Laurent Imbert. An alternative approach for SIDH arithmetic. In Juan A. Garay, editor, *Public-Key Cryptography — PKC 2021*, volume 12710 of *Lecture Notes in Computer Science*, pages 27–44. Springer Verlag, 2021.

[CFG+21] Hao Cheng, Georgios Fotiadis, Johann Großschädl, Peter Y. A. Ryan, and Peter B. Rønne. Batching CSIDH group actions using AVX-512. *Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):618–649, August 2021.

[CGT+21] Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, and Peter Y. A. Ryan. High-throughput elliptic curve cryptography using AVX2 vector instructions. In Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography — SAC 2020*, volume 12804 of *Lecture Notes in Computer Science*, pages 698–719. Springer Verlag, 2021.

[Cho16] Tung Chou. Sandy2x: New Curve25519 speed records. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography — SAC 2015*, volume 9566 of *Lecture Notes in Computer Science*, pages 145–160. Springer Verlag, 2016.

[CJL+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. National Institute of Standards and Technology (NIST) Interagency Report 8105, available for download at https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf, 2016.

[CLN16] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology — CRYPTO 2016*, volume 9814 of *Lecture Notes in Computer Science*, pages 572–601. Springer Verlag, 2016.

[COR20] Daniel Cervantes-Vázquez, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. Parallel strategies for SIDH: Towards computing SIDH twice as fast. Cryptology ePrint Archive, Report 2020/383, 2020.

[CS09] Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the Cell Broadband Engine. In Bart Preneel, editor, *Progress in Cryptology — AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*. Springer Verlag, 2009.

[DJP14] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology*, 8(3):209–247, 2014.

[FL15] Armando Faz-Hernández and Julio López. Fast implementation of Curve25519 using AVX2. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology — LATINCRYPT 2015*, volume 9230 of *Lecture Notes in Computer Science*, pages 329–345. Springer Verlag, 2015.

[FLD19] Armando Faz-Hernández, Julio López, and Ricardo Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software*, 45(3):1–35, July 2019.

[FLOR18] Armando Faz-Hernández, Julio C. López-Hernández, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IEEE Transactions on Computers*, 67(11):1622–1636, November 2018.

[Gal99] Steven D. Galbraith. Constructing isogenies between elliptic curves over finite fields. *LMS Journal of Computation and Mathematics*, 2:118–138, 1999.

[GK16] Shay Gueron and Vlad Krasnov. Accelerating big integer arithmetic using Intel IFMA extensions. In Paolo Montuschi, Michael J. Schulte, Javier Hormigo, Stuart F. Oberman, and Nathalie Revol, editors, *Proceedings of the 23rd IEEE Symposium on Computer Arithmetic (ARITH 2016)*, pages 32–38. IEEE Computer Society, 2016.

[HEY20] Hüseyin Hişil, Berkan Eğrice, and Mert Yassi. Fast 4 way vectorized ladder for the complete set of Montgomery curves. Cryptology ePrint Archive, Report 2020/388, 2020.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography — TCC 2017*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371. Springer Verlag, 2017.

[HMV04] Darrel R. Hankerson, Alfred J. Menezes, and Scott A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

[Int18] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual. Available for download at https://software.intel.com/content/dam/develop/public/us/en/documents/64-ia-32-architectures-optimization-manual.pdf, 2018.

[JAC+20] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchinson, Amir Jalali, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular isogeny key encapsulation. Specification, available for download at https://sike.org/files/SIDH-spec.pdf, 2020.

[JD11] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography — PQCrypto 2011*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer Verlag, 2011.

[KG19] Dusan Kostic and Shay Gueron. Using the new VPMADD instructions for the new post quantum key encapsulation mechanism SIKE. In Naofumi Takagi, Sylvie Boldo, and Martin Langhammer, editors, *Proceedings of the 26th IEEE Symposium on Computer Arithmetic (ARITH 2019)*, pages 215–218. IEEE, 2019.

[KO63] Anatoly A. Karatsuba and Yurii P. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics - Doklady*, 7(7):595–596, January 1963.

[KP21] Matthias J. Kannwischer and Richard Petri. pqm4: NISTPQC round 3 results on the Cortex-M4. Presentation given at the 3rd NIST PQC Standardization Conference, slides available for download at https://csrc.nist.gov/CSRC/media/Presentations/pqm4-nistpqc-round-3-results-on-the-cortex-m4/images-media/session-3-kannwischer-pqm4.pdf, 2021.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, January 1987.

[Nat15] National Institute of Standards and Technology (NIST). SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS Publication 202, available for download at https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf, August 2015.

[Nat19] National Institute of Standards and Technology (NIST). NIST reveals 26 algorithms advancing to the post-quantum crypto 'semifinals'. Press release, available online at https://www.nist.gov/news-events/news/2019/01/nist-reveals-26-algorithms-advancing-post-quantum-crypto-semifinals, 2019.

[Nat20] National Institute of Standards and Technology (NIST). NIST's post-quantum cryptography program enters 'selection round'. Press release, available online at https://www.nist.gov/news-events/news/2020/07/nists-post-quantum-cryptography-program-enters-selection-round, 2020.

[NS20] Kaushik Nath and Palash Sarkar. Efficient 4-way vectorizations of the Montgomery ladder. Cryptology ePrint Archive, Report 2020/378, 2020.

[OAL18] Gabriell Orisaka, Diego F. Aranha, and Julio López. Finite field arithmetic using AVX-512 for isogeny-based cryptography. In *Proceedings of the 18th Brazilian Symposium on Information and Computational Systems Security (SBSEG 2018)*, pages 49–56. Brazilian Computing Society, 2018.

[Sin19] Sujoy Sinha Roy. SaberX4: High-throughput software implementation of Saber key encapsulation mechanism. In *Proceedings of the 37th IEEE International Conference on Computer Design (ICCD 2019)*, pages 321–324. IEEE, 2019.

[SLLH18] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. SIDH on ARM: Faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):1–20, September 2018.

[Tan09] Seiichiro Tani. Claw finding algorithms using quantum walk. *Theoretical Computer Science*, 410(50):5285–5297, November 2009.

[Tat66] John Tate. Endomorphisms of abelian varieties over finite fields. *Inventiones Mathematicae*, 2(2):134–144, April 1966.

[TWL+20] Jing Tian, Piaoyang Wang, Zhe Liu, Jun Lin, Zhongfeng Wang, and Johann Großschädl. Efficient software implementation of the SIKE protocol using a new data representation. Cryptology ePrint Archive, Report 2020/660, 2020.

[Vél71] Jacques Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences de Paris, Série A*, 273(4):238–241, July 1971.

A Benchmarks for SIKEp434, SIKEp610, and SIKEp751

This section presents benchmarking results (measured on our Ice Lake CPU) of various operations at different layers of the SIDHv3.4 assembly implementation (serving as the baseline) and AvxSike for the parameter sets SIKEp434, SIKEp610, and SIKEp751.

Table 7: Benchmarking results of $\mathbb{F}_p$-arithmetic operations.

Scheme	Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
SIKEp434	Integer multiplication	SIDHv3.4	x64 asm	1-way	1	82	82	1.00×
		AvxSike	AVX-512	(8×1)-way	8	130	16	5.05×
		AvxSike	AVX-512	(4×2)-way	4	91	23	3.60×
	Montgomery reduction	SIDHv3.4	x64 asm	1-way	1	60	60	1.00×
		AvxSike	AVX-512	(8×1)-way	8	112	14	4.29×
		AvxSike	AVX-512	(4×2)-way	4	128	32	1.88×
	Montgomery multiplication	SIDHv3.4	x64 asm	1-way	1	169	169	1.00×
		AvxSike	AVX-512	(8×1)-way	8	276	35	4.90×
		AvxSike	AVX-512	(4×2)-way	4	231	58	2.93×
SIKEp610	Integer multiplication	SIDHv3.4	x64 asm	1-way	1	131	131	1.00×
		AvxSike	AVX-512	(8×1)-way	8	255	32	4.11×
		AvxSike	AVX-512	(4×2)-way	4	143	36	3.66×
	Montgomery reduction	SIDHv3.4	x64 asm	1-way	1	117	117	1.00×
		AvxSike	AVX-512	(8×1)-way	8	201	25	4.66×
		AvxSike	AVX-512	(4×2)-way	4	188	47	2.49×
	Montgomery multiplication	SIDHv3.4	x64 asm	1-way	1	297	297	1.00×
		AvxSike	AVX-512	(8×1)-way	8	412	52	5.77×
		AvxSike	AVX-512	(4×2)-way	4	342	86	3.47×
SIKEp751	Integer multiplication	SIDHv3.4	x64 asm	1-way	1	202	202	1.00×
		AvxSike	AVX-512	(8×1)-way	8	365	46	4.43×
		AvxSike	AVX-512	(4×2)-way	4	241	60	3.35×
	Montgomery reduction	SIDHv3.4	x64 asm	1-way	1	149	149	1.00×
		AvxSike	AVX-512	(8×1)-way	8	296	37	4.03×
		AvxSike	AVX-512	(4×2)-way	4	342	86	1.74×
	Montgomery multiplication	SIDHv3.4	x64 asm	1-way	1	425	425	1.00×
		AvxSike	AVX-512	(8×1)-way	8	692	87	4.91×
		AvxSike	AVX-512	(4×2)-way	4	570	143	2.98×

Table 8: Benchmarking results of $\mathbb{F}_{p^2}$-arithmetic implementations.

Scheme	Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
SIKEp434	$\mathbb{F}_{p^2}$ Multiplication	SIDHv3.4	x64 asm	1-way	1	408	408	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	763	95	4.28×
		AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	493	123	3.31×
		AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	398	199	2.05×
	$\mathbb{F}_{p^2}$ Squaring	SIDHv3.4	x64 asm	1-way	1	356	356	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	614	77	4.64×
		AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	368	92	3.87×
		AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	281	141	2.53×
SIKEp610	$\mathbb{F}_{p^2}$ Multiplication	SIDHv3.4	x64 asm	1-way	1	735	735	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	1894	237	3.10×
		AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	1099	275	2.68×
		AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	812	406	1.81×
	$\mathbb{F}_{p^2}$ Squaring	SIDHv3.4	x64 asm	1-way	1	583	583	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	1184	148	3.94×
		AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	714	179	3.27×
		AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	559	280	2.09×
SIKEp751	$\mathbb{F}_{p^2}$ Multiplication	SIDHv3.4	x64 asm	1-way	1	954	954	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	1997	250	3.82×
		AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	1277	319	2.99×
		AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	940	470	2.03×
	$\mathbb{F}_{p^2}$ Squaring	SIDHv3.4	x64 asm	1-way	1	762	762	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1)$-way	8	1433	179	4.25×
		AvxSike	AVX-512	$(4 \times 2 \times 1)$-way	4	768	192	3.97×
		AvxSike	AVX-512	$(2 \times 2 \times 2)$-way	2	628	314	2.43×

Table 9: Benchmarking results of implementations of point operations.

Scheme	Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
SIKEp434	Ladder step (xDBLADD)	SIDHv3.4	x64 asm	1-way	1	4413	4413	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	8289	1036	4.26×
		AvxSike	AVX-512	$(2 \times 4 \times 1 \times 1)$-way	2	2462	1231	3.58×
		AvxSike	AVX-512	$(1 \times 4 \times 2 \times 1)$-way	1	1518	1518	2.91×
	Point doubling (xDBL)	SIDHv3.4	x64 asm	1-way	1	2432	2432	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	4359	545	4.46×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	1444	722	3.37×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1153	1153	2.11×
	Point tripling (xTPL)	SIDHv3.4	x64 asm	1-way	1	4867	4867	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	8805	1101	4.42×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	2485	2485	1.96×
SIKEp610	Ladder step (xDBLADD)	SIDHv3.4	x64 asm	1-way	1	7292	7292	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	17556	2195	3.32×
		AvxSike	AVX-512	$(2 \times 4 \times 1 \times 1)$-way	2	5277	2639	2.76×
		AvxSike	AVX-512	$(1 \times 4 \times 2 \times 1)$-way	1	3225	3225	2.26×
	Point doubling (xDBL)	SIDHv3.4	x64 asm	1-way	1	4102	4102	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	9436	1180	3.48×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	3007	1504	2.73×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	2427	2427	1.69×
	Point tripling (xTPL)	SIDHv3.4	x64 asm	1-way	1	8228	8228	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	18598	2325	3.54×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	4946	4946	1.66×
SIKEp751	Ladder step (xDBLADD)	SIDHv3.4	x64 asm	1-way	1	9703	9703	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	19438	2430	3.99×
		AvxSike	AVX-512	$(2 \times 4 \times 1 \times 1)$-way	2	6037	3019	3.21×
		AvxSike	AVX-512	$(1 \times 4 \times 2 \times 1)$-way	1	3540	3540	2.74×
	Point doubling (xDBL)	SIDHv3.4	x64 asm	1-way	1	5388	5388	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	10613	1327	4.06×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	3404	1702	3.17×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	2679	2679	2.01×
	Point tripling (xTPL)	SIDHv3.4	x64 asm	1-way	1	10633	10633	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	20939	2617	4.06×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	5587	5587	1.90×

Table 10: Benchmarking results of implementations of isogeny operations.

Scheme	Operation	Reference	Impl.	Vectorization	#Inst.	Cycles	Cyc./inst.	Speed-up
SIKEp434	4-isogeny generation (`get_4_isog`)	SIDHv3.4	x64 asm	1-way	1	1518	1518	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	2828	354	4.29×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	773	387	3.92×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	629	629	2.41×
	4-isogeny evaluation (`eval_4_isog`)	SIDHv3.4	x64 asm	1-way	1	3299	3299	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	6002	750	4.40×
		AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	3207	802	4.11×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	1936	968	3.41×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1562	1562	2.11×
	3-isogeny generation (`get_3_isog`)	SIDHv3.4	x64 asm	1-way	1	2117	2117	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	3863	483	4.38×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1248	1248	1.70×
	3-isogeny evaluation (`eval_3_isog`)	SIDHv3.4	x64 asm	1-way	1	2400	2400	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	4375	547	4.39×
		AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	2200	550	4.36×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	1376	688	3.49×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1114	1114	2.15×
SIKEp610	4-isogeny generation (`get_4_isog`)	SIDHv3.4	x64 asm	1-way	1	2427	2427	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	5280	660	3.68×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	1626	813	2.99×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1194	1194	2.03×
	4-isogeny evaluation (`eval_4_isog`)	SIDHv3.4	x64 asm	1-way	1	5467	5467	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	13091	1636	3.34×
		AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	6855	1714	3.19×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	4227	2114	2.59×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	3318	3318	1.65×
	3-isogeny generation (`get_3_isog`)	SIDHv3.4	x64 asm	1-way	1	3470	3470	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	7658	957	3.63×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	2407	2407	1.44×
	3-isogeny evaluation (`eval_3_isog`)	SIDHv3.4	x64 asm	1-way	1	3981	3981	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	9517	1190	3.35×
		AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	4863	1216	3.27×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	2973	1487	2.68×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	2300	2300	1.73×
SIKEp751	4-isogeny generation (`get_4_isog`)	SIDHv3.4	x64 asm	1-way	1	3177	3177	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	6311	789	4.03×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	1730	865	3.67×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	1367	1367	2.32×
	4-isogeny evaluation (`eval_4_isog`)	SIDHv3.4	x64 asm	1-way	1	7281	7281	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	14367	1796	4.05×
		AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	7589	1897	3.84×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	4617	2309	3.15×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	3657	3657	1.99×
	3-isogeny generation (`get_3_isog`)	SIDHv3.4	x64 asm	1-way	1	4522	4522	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	8954	1119	4.04×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	2700	2700	1.67×
	3-isogeny evaluation (`eval_3_isog`)	SIDHv3.4	x64 asm	1-way	1	5308	5308	1.00×
		AvxSike	AVX-512	$(8 \times 1 \times 1 \times 1)$-way	8	10558	1320	4.02×
		AvxSike	AVX-512	$(4 \times 2 \times 1 \times 1)$-way	4	5416	1354	3.92×
		AvxSike	AVX-512	$(2 \times 2 \times 2 \times 1)$-way	2	3282	1641	3.23×
		AvxSike	AVX-512	$(1 \times 2 \times 2 \times 2)$-way	1	2633	2633	2.02×