Vectorized $(2, 2)$-Isogenies in the Theta Model Using AVX-512 and Applications to SQIsign

Marius A. Aardal[1], Diego F. Aranha[1], Patrick Longa[2] and Giacomo Pope[3,4]

[1] Aarhus University, Aarhus, Denmark
[2] Microsoft Research, Redmond, United States
[3] University of Bristol, Bristol, United Kingdom
[4] NCC Group, Cheltenham, United Kingdom

Abstract. Higher-dimensional isogenies have emerged as a powerful building block for isogeny-based cryptography, with recent protocols such as SQIsign relying on the efficient evaluation of $(2^n, 2^n)$-isogenies. Despite their attractive security and compactness properties, these schemes remain computationally expensive and improving their performance is an important step for their real-world use. In this work, we present a high-performance AVX-512 implementation of two-dimensional isogenies that effectively exploits their data-level parallelism. We develop a code generator that produces efficient AVX-512 implementations of arithmetic over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ for Montgomery-friendly primes of the form $p = c \cdot 2^e - 1$ with $e \approx \log_2(p)$, covering the family of primes used by several protocols based on two-dimensional isogenies, including SQIsign. Building on this arithmetic, we vectorize the $(2^n, 2^n)$-isogeny formulas by Dartois et al. (ASIACRYPT 2024), achieving significant speedups over the corresponding, highly-optimized scalar implementation that uses x86-64 assembly. As an application, we accelerate the Round 2 version of SQIsign verification and obtain speedups of 2.5× at NIST security level I and 3.2× and 3.3× at levels III and V, respectively. Our code generator also readily supports the larger primes of Round 3 SQIsign, for which we obtain verification speedups between 3.1× and 3.4×.

Keywords: Vectorized implementation · AVX-512 · Isogenies · Post-quantum · SQIsign

1 Introduction

Higher-dimensional isogenies have recently become an invaluable tool in isogeny-based cryptography. Kani's lemma [Kan97] enables the evaluation of an isogeny of non-smooth degree efficiently by embedding it into a higher-dimensional isogeny of smooth degree. The technique was first applied as part of the landmark attacks on SIKE [CD23, MMP+23, Rob23]. Following the attacks, the cryptographic community realized that higher-dimensional isogenies could also be used constructively to design simpler and more efficient protocols with cleaner security proofs. This observation led to a new generation of post-quantum cryptographic constructions requiring the computation of chains of $(2, 2)$-isogenies between products of two elliptic curves. Examples include the higher-dimensional variants of the original SQIsign signature scheme [DLRW24, BDD+24, NOC+24, DF24], the public-key encryption schemes FESTA [BMP23], QFESTA [NO24] and POKÉ [BM25], the key-encapsulation mechanism IS-CUBE [Mor23], the signature scheme PRISM [BBC+25], the

E-mail: maardal@cs.au.dk (Marius A. Aardal), dfaranha@cs.au.dk (Diego F. Aranha), plonga@microsoft.com (Patrick Longa), giacomopope@gmail.com (Giacomo Pope)

This work is licensed under a "CC BY 4.0" license.
Date of this document: 2026-10-31.

non-interactive key exchange MIKE [Rob24], as well as isogeny-group-action frameworks like SCALLOP-HD [CLP24], CORAL [BBRS26], KLaPoTi [BCE+25] and (qt-)Pegasis [DEF+25, DEIV26].

Arguably, the most notable representative in the new generation of isogeny-based protocols is the signature scheme SQIsign, which is a candidate in NIST's ongoing standardization process for additional digital signatures [Nat26]. In the Round 1 version of SQIsign [DKL+20, CSD+23], the costliest part of signing was sampling the response isogeny from the public-key curve to the challenge curve. To enable the verifier to evaluate the response isogeny efficiently, the scheme used the KLPT algorithm [KLPT14, DLLW23] to sample a smooth-degree response of degree $\approx p^{3.75}$. This made signing prohibitively slow, and required $p^2 - 1$ to be divisible by a large smooth odd factor, which is undesirable for fast field arithmetic.

The application of two-dimensional isogenies in the Round 2 version of SQIsign changed this picture dramatically [BDD+24, AAA+24]. The response isogeny could now be sampled as a uniformly random isogeny of degree roughly $p^{0.5}$, and be efficiently evaluated by embedding it into a $(2^n, 2^n)$-isogeny. Without KLPT, the scheme was also free to select a Montgomery-friendly prime of the form $p = c \cdot 2^e - 1$ with $e \approx \log_2(p)$, allowing much faster field arithmetic. Overall, this yielded a 12× speedup for signing and a 3× speedup for verification at NIST security level I.

The state-of-the-art formulas for computing $(2^n, 2^n)$-isogenies were introduced by [DMPR24]. Importantly, these formulas exhibit substantial data-level parallelism. Although recent advances have significantly improved the efficiency of isogeny-based schemes, their relatively high computational cost remains one of the most challenging obstacles to real-world deployment. It is therefore an important challenge to exploit the parallelism inherent in the $(2^n, 2^n)$-isogeny formulas to develop faster implementations using the Single Instruction Multiple Data (SIMD) capabilities of modern processors.

The x86 SIMD architecture has evolved over several generations, with the most recent version being AVX-512. AVX-512 provides 512-bit vector registers for computing eight 64-bit operations in parallel and has extensive support for integer arithmetic [Cor25b]. Notably, Intel's AVX10 roadmap appears to consolidate the long-term relevance of vectorized cryptographic software [Cor25a]. AVX10 is a converged vector ISA that incorporates the functionality of AVX-512 under a unified programming model for future processors. As a result, optimizations developed for AVX-512 are expected to remain applicable on future AVX10-enabled platforms, extending the practical impact of vectorized implementations.

Contributions. In this work, we investigate how effectively the parallelism inherent in $(2^n, 2^n)$-isogenies can be exploited using AVX-512. We make the following contributions.

1. We develop a code generator that produces efficient AVX-512 implementations of $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic for Montgomery-friendly primes of the form $p = c \cdot 2^e - 1$ with $e \approx \log_2(p)$. Our generator leverages and combines in an efficient way techniques such as lazy reduction, delayed carry propagation and non-canonical representation in the range $[0, 2p)$, together with the vectorization formats described in [CFGR22]. We provide functionality for evaluating two, four and eight independent field operations in parallel. All these techniques are carefully tailored to the shape of Montgomery-friendly primes to ensure top performance.

   The class of supported primes includes those used by SQIsign and, more generally, those used by the several schemes relying on two-dimensional isogenies, such as PRISM [BBC+25], Qlapoti [BCE+25], qt-Pegasis [DEIV26] and CORAL [BBRS26].

2. We present a high-performance AVX-512 implementation of the $(2^n, 2^n)$-isogeny formulas from [DMPR24]. We apply vectorization techniques that efficiently combine the field arithmetic and isogeny layers, trying to maximize parallelism while minimizing shuffle and data-movement overheads. The resulting library, which leverages

the field arithmetic produced by our code generator, provides a reusable SIMD backend for higher-dimensional isogeny computations that can be integrated into scalar implementations for a broad class of isogeny-based protocols with minimal modifications.

3. As a case study, we integrate our SIMD isogeny library into (Round 2) SQIsign verification and further identify and exploit additional sources of parallelism in the remaining verification routines. Relative to the highly-optimized implementation of SQIsign using hand-written assembly [AAA+24], we obtain a 2.5× speedup for SQIsign level I and speedups of 3.2× and 3.3× for levels III and V, respectively. We also adapt our implementation to Round 3 SQIsign, which uses larger primes and an x-only gluing isogeny, and obtain speedups between 3.1× and 3.4× across the three security levels (cf. Section 5.5).

Our implementation, including the field arithmetic generator in Python and our vectorized C code with AVX-512 intrinsics for both Round-2 and Round-3 SQIsign, is publicly available[1].

Related work. Modern implementations exploiting AVX-512 to speed up isogeny-based cryptography started with Kostic and Gueron [KG19] and Cheng et al. [CFGR22]. In contrast to previous works like [OLHA18], these two works had access to the *Integer Fused Multiply-Add (IFMA)* extension, also known as AVX-512IFMA. This extension includes specialized instructions that compute eight parallel 52-bit multiplications. In particular, [CFGR22] developed parallelization strategies for the (quadratic extension) field arithmetic that exploit more effectively the inherent parallelism found in elliptic curve and isogeny computations. Their approach in turn minimized the number of radix conversions and enabled the use of the more efficient Karatsuba algorithm for multiplication over $\mathbb{F}_{p^2}$. Subsequent work extended the application of these parallelization strategies to the computation of pairings [CFGP25].

More recently and similarly to this work, Wang et al. [WWW+26] presented a vectorized implementation of SQIsign exploiting AVX-512IFMA. As we show in Section 3.5, our field arithmetic implementation, which adapts and optimizes the algorithms from [CFGR22] to primes of the form $p = c \cdot 2^e - 1$ with $e \approx \log_2(p)$, is significantly faster.

Finally, De Feo et al. [FJWY26] presented a vectorized implementation of SQIsign targeting Arm devices using NEON. While there are some similarities in the parallelization strategies, the NEON engine is quite different from AVX-512; it is a 128-bit vector instruction set, it does not support 64-bit vector long multiplication and, therefore, the field arithmetic implementation for SQIsign is limited to using SIMD instructions that execute two 32-bit long multiplications in parallel.

2 Preliminaries

Notation. We fix some notation for multi-precision modular arithmetic. Throughout, p is an odd prime and we identify each $\mathbb{F}_p$ element by its representative in $[0, p)$. We let ℓ denote the limb bit-length (so each limb lies in $[0, 2^\ell)$) and let $N = \lceil \log_2 p/\ell \rceil$ be the number of ℓ-bit limbs needed to store an $\mathbb{F}_p$ element. The i-th limb of an integer a is $a[i]$, so that $a = \sum_{i \geq 0} a[i] \cdot 2^{\ell \cdot i}$.

For level-two theta points in dimension two, we sometimes denote the four projective coordinates of the point $P = (x : y : z : t)$ as P^x, P^y, P^z and P^t (usually when explaining the creation of a new point from coordinates of other points).

The cost of a single addition, multiplication and squaring is referred to as $\mathbf{A}, \mathbf{M}$ and $\mathbf{S}$, respectively. When considering vectorized arithmetic to compute n simultaneous operations,

[1] https://github.com/dfaranha/sqisign-avx512

the cost of the same operations is denoted as $\mathbf{A}_n, \mathbf{M}_n$ and $\mathbf{S}_n$. Note that for cost counting, a subtraction is counted as an addition; for example, we cost the computation $a+b, a-b$ as $\mathbf{A}_2$.

2.1 Montgomery multiplication

Multi-precision modular multiplication can be implemented efficiently using Montgomery multiplication [Mon85], which proceeds in two stages. First, the operands $a, b \in \mathbb{F}_p$ are multiplied as integers, using multi-precision arithmetic, yielding a result $t \in [0, p^2)$ stored in $2N$ limbs. Then, Montgomery's reduction algorithm is applied to t (see Algorithm 1). It outputs $a \cdot b \cdot R^{-1} \bmod p$, where $R = 2^{\ell N}$ is the *Montgomery radix*. To use this reduction, we actually keep the inputs in "Montgomery form" $\tilde{a} = a \cdot R \bmod p$, so that reducing the product $\tilde{a} \cdot \tilde{b} = abR^2 \bmod p$ yields $a \cdot b \cdot R \bmod p = \widetilde{ab}$, again in Montgomery form.

A prime of the form $p = c \cdot 2^e - 1$ with $e \geq \ell$ is said to be *Montgomery-friendly*. These primes are used in isogeny-based cryptography and admit a cheaper reduction. Since $p' = 1 \bmod 2^\ell$, line 3 of Algorithm 1 can be simplified to $u = s[i]$. Moreover, $u \cdot p \cdot 2^{\ell \cdot i} = u \cdot c \cdot 2^{e+i\ell} - u \cdot 2^{i\ell}$. Subtracting $u \cdot 2^{i\ell} = s[i] \cdot 2^{i\ell}$ zeroes out the i-th limb of s with no borrow into higher limbs. Since these subtractions only affect the first N limbs of s, which are dropped in line 6, we can skip them. Hence, the full multiplication by p in line 4 is replaced by a short multiplication by c and a shift.

Algorithm 1 Montgomery reduction (word-by-word)

Input: t with $0 \leq t < Rp$, where $R = 2^{\ell N} > p$, $\gcd(p, R) = 1$, and $p' = -p^{-1} \bmod 2^\ell$.
Output: $s = tR^{-1} \bmod p$

1: $s \leftarrow t$
2: **for** $i = 0$ **to** $N - 1$ **do**
3: $\quad u \leftarrow s[i] \cdot p' \bmod 2^\ell$
4: $\quad s \leftarrow s + u \cdot p \cdot 2^{i\ell}$
5: **end for**
6: $s \leftarrow s/R$ // drop low N limbs
7: **if** $s \geq p$ **then**
8: $\quad s \leftarrow s - p$ // reduce from mod $2p$ to mod p
9: **end if**
10: **return** s

2.2 AVX-512

AVX-512 provides 512-bit vector registers, allowing arithmetic instructions to operate on eight 64-bit integers in parallel. In this case, most instructions apply the same operation independently to all eight lanes. Shuffle instructions move the 64-bit integers between the lanes of a vector, and blend instructions allow merging and shuffling the lanes of two vectors. In addition to the core instruction set, we make use of the AVX-512IFMA extension, which includes powerful 52-bit integer multiply-and-add instructions. In this paper, we present the vectorized algorithms in AVX-512 pseudocode for clarity, with the most common instructions presented in Table 1.

2.3 Computing $(2^n, 2^n)$-isogenies

We first recall the one-dimensional case, with supersingular elliptic curves over $\mathbb{F}_{p^2}$. An isogeny $\phi : E \to E'$ is a non-constant rational map between two elliptic curves, which is also a group homomorphism. The degree $d = \deg(\phi)$ is its degree as a rational map, and we say that ϕ is a d-isogeny. When d is coprime to p, ϕ is determined (up to isomorphism)

Table 1: Common AVX-512 instructions used in our code. Intrinsic names omit the `_mm512_` prefix.

Type	Pseudocode	Intrinsic	Explanation
Integer arith.	$\mathsf{VADD}(A, B)$	`add_epi64`	$a_i + b_i$
	$\mathsf{VSUB}(A, B)$	`sub_epi64`	$a_i - b_i$
	$\mathsf{VMULADDL}(A, B, C)$	`madd52lo_epu64`	$c_i + (a_i \cdot b_i)_{\text{lo}}$
	$\mathsf{VMULADDH}(A, B, C)$	`madd52hi_epu64`	$c_i + (a_i \cdot b_i)_{\text{hi}}$
Logical	$\mathsf{VAND}(A, B)$	`and_epi64`	$a_i \,\&\, b_i$
Shift	$\mathsf{VSL}(A, n)$	`slli_epi64`	$a_i \ll n$
	$\mathsf{VSRA}(A, n)$	`srai_epi64`	$a_i \gg_{\text{arith}} n$
Assignment	$\mathsf{VSET}(a_0, \ldots, a_7)$	`set_epi64`	New vector $(a_0, \ldots, a_7)$
	$\mathsf{VSETALL}(a)$	`set1_epi64`	New vector $(a, \ldots, a)$

by its kernel $\ker \phi$ of order d. From the kernel, the isogeny can be computed at the cost of $\tilde{O}(\sqrt{d})$ field operations using $\sqrt{\text{élu}}$ [BFLS20]. This is only efficient for small values of d. However, if d is smooth, e.g. $d = 2^n$, we can efficiently evaluate ϕ as a chain of 2-isogenies $\phi = \phi_n \circ \cdots \circ \phi_1$, each computable in $O(1)$ field operations, so that the total cost is $O(n)$.

The two-dimensional case is similar. Here the analogue of an elliptic curve is a principally polarized abelian surface, which is either a product $E_1 \times E_2$ of two elliptic curves or the Jacobian of a genus-2 curve. A (d, d)-isogeny $\Phi : A \to A'$ between principally polarized abelian surfaces has kernel generated by two linearly independent points $P, Q \in A$ of order d, which are isotropic, meaning the Weil pairing $e_d(P, Q) = 1$. Just as a 2^n-isogeny of elliptic curves decomposes into a chain of 2-isogenies, a $(2^n, 2^n)$-isogeny decomposes into a chain of $(2, 2)$-isogenies $\Phi = \Phi_n \circ \cdots \circ \Phi_1$.

Fast formulas for computing chains of $(2, 2)$-isogenies were introduced in [DMPR24]. They represent points on the abelian surfaces in *theta coordinates*, which can be seen as a generalization of x-only coordinates. In theta coordinates (of level two), a point P is represented projectively as a tuple $(x : y : z : t)$ of four $\mathbb{F}_{p^2}$ coordinates. The abelian surface A is represented by a special theta point θ_A which we call the theta null point. A single $(2, 2)$-step is one of three types according to the domain and codomain: a *gluing* step from a product of elliptic curves to a Jacobian, a *generic* step between Jacobians, and a *splitting* step from a Jacobian back to a product. When the $(2^n, 2^n)$-isogeny $\Phi : E_1 \times E_2 \to E_3 \times E_4$ is sampled randomly between products of elliptic curves, as in SQIsign, the middle steps are heuristically all generic except with negligible probability [AAA+24]. Therefore, the formulas of [DMPR24] assume a chain consisting of an initial gluing step, $n - 2$ generic steps and a final splitting step. Unlike when working on the Jacobian model however, only the gluing isogeny needs special treatment and the remaining $n - 1$ isogenies all use the same formula[2], with the extra work of splitting being a final symplectic transform to efficiently recover the Montgomery models of the codomain from the level two theta null point.

3 Automatically Generated Field Arithmetic

In this section, we present a code generator producing AVX-512 vectorized field arithmetic for the special class of Montgomery-friendly primes $p = c \cdot 2^e - 1$ with $e \approx \log_2(p)$ (made precise in Section 3.1). It first generates the $\mathbb{F}_p$ arithmetic and then builds the quadratic field extension $\mathbb{F}_{p^2} = \mathbb{F}_p[X]/\langle X^2 + 1\rangle$ on top. In Section 3.1 and Section 3.2, we describe the generic vectorization formats and algorithms for the $\mathbb{F}_p$ and the $\mathbb{F}_{p^2}$ layers,

[2]One must be careful in the final two steps of the isogeny chain to avoid zero coordinates which appear when computing isogenies to an elliptic product; but this is easily and deterministically handled.

Table 2: Prime parameters and limb layouts at each security level for Round-2 SQIsign.

			64-bit	AVX-512		
Level	Prime	Bits	#Limbs	#Limbs N	Limb bits ℓ	Top limb bits ℓ_{top}
I	$5 \cdot 2^{248} - 1$	251	4	5	51	47
III	$65 \cdot 2^{376} - 1$	383	6	8	49	40
V	$27 \cdot 2^{500} - 1$	505	8	10	51	46

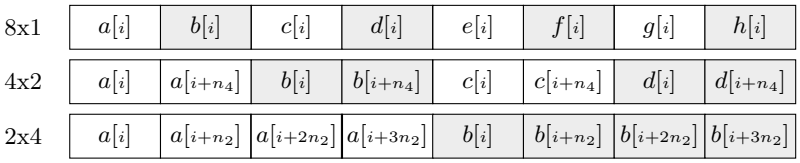

Figure 1: Contents of the i-th vector under the three vectorized $\mathbb{F}_p$ formats $(k \times z)$. We write $a[j]$ for the j-th limb of $a \in \mathbb{F}_p$. If z does not divide the number of limbs N, the last slots are not used.

respectively. Then, in Section 3.3, we describe the prime-specific strategies that the code generator applies to improve performance. We also provide functions for converting between vectorization formats, which we discuss in Section 3.4. Finally, in Section 3.5, we present benchmarks for each layer.

3.1 Prime Field

We represent an $\mathbb{F}_p$ element as a sequence of N limbs stored in AVX-512 vectors. Since the AVX-512IFMA integer multiplier operates on 52-bit operands, we use a limb bit-length ℓ of at most 52-bits. We use ℓ_{top} to denote the bit-length of the top limb, which may be shorter. The limb length ℓ is chosen per prime by the code generator, as we discuss in Section 3.3. For example, the parameters chosen for the SQIsign (Round 2) primes are listed in Table 2. As a simplifying assumption, the code generator requires $e \geq (N-1)\ell$ and that $N \geq 4$. The first assumption ensures that all but the top limb of $p+1$ are zero, while the second one allows us to ignore special cases that only arise for primes too small to be cryptographically relevant. While both restrictions are suitable for the range of parameters we target, we note that they could be easily lifted by handling the additional cases in the code generator.

We implement 2-way, 4-way and 8-way vectorization on the $\mathbb{F}_p$ layer, for computing 2, 4 or 8 independent $\mathbb{F}_p$ operations in parallel. Each k-way implementation requires its own format for representing the $\mathbb{F}_p$ elements. To describe how these formats utilize the 8 lanes, we adopt the notion of "$(k \times z)$-way parallelism" from [CFGR22]. An implementation has $(k \times z)$-way parallelism if it computes k independent $\mathbb{F}_p$ operations, where each operation uses $z = 8/k$ lanes. We store the k elements using $n_k = \lceil N/z \rceil$ vectors, with the limbs arranged as in Fig. 1. For example, with the (4×2) format, we split each element across two adjacent lanes. For an even number of limbs, $N = 2n_4$ and the first vector holds limbs 0 and n_4, the second vector holds limbs 1 and $n_4 + 1$ and so on. If the number of limbs is odd so that $N = 2n_4 - 1$, the last vector only stores limb $n_4 - 1$, the other slot is not used.

We proceed to discuss the field operations for each vectorization format. We focus primarily on multiplication, as it is the field operation that dominates the running time of isogeny-based schemes. We adapt and optimize the (8×1)-way operations from [CFGR22] and the (4×2)-way operations from [OLHA18] to the special class of Montgomery-friendly primes we are considering. To explore $\mathbb{F}_p$ vectorization further, we also introduce algorithms for (2×4)-way operations. All the field operations have constant-time implementations.

3.1.1 Additions, subtractions and carry propagation

Conceptually, the addition algorithm is the same for all three formats. Given two operands $\boldsymbol{A}$ and $\boldsymbol{B} \in [0, p)$ in the $(k \times z)$ format, holding $a[1], \ldots, a[k]$ and $b[1], \ldots, b[k]$, it returns $a[1] + b[1], \ldots, a[k] + b[k]$ in the same format. First, we compute $(a[i] + b[i]) - p \in [-p, p)$ as integers. The addition is computed vector by vector as $R_j \leftarrow \mathsf{VADD}(A_j, B_j)$. For the subtraction with p, we use the fact that $p + 1 = c \cdot 2^e$ has all but its top limb equal to zero, so subtracting p touches only two limbs: we subtract $p + 1$ from the top limb and add 1 to the lowest. Finally, we propagate carries, sequentially adding the overflow of each limb to the next. If the result is negative, we conditionally add p back (exploiting the special form of p as before), followed by another carry propagation. This ensures that the result is reduced modulo p.

Note that for some suitable primes we can improve the cost of multiplications by working in the range $[0, 2p)$ instead of $[0, p)$. In this case, we subtract and add $2p$ in the procedure explained above (see Section 3.3).

Subtraction is implemented similarly to addition. We also provide a combined add–subtract operation in which a k-bit flag selects, per element, whether each pair is added or subtracted. This is used in the Hadamard operation for $(2, 2)$-isogeny computation.

The difference between the formats mainly lies in how carries are propagated. In the (8×1) format, this can be achieved by one *ripple* from the first vector to the last. In the (4×2) format, the first ripple only propagates the carry to limb $(n_4 - 1)$. The carry out of this limb must be shifted to the next lane and added back to the first vector. A subsequent ripple finally propagates the carry to the top limb. In the (2×4) format, carry propagation becomes even more expensive. Up to four ripples and three cross-lane shifts are needed, depending on which lane the top limb sits in. The (2×4)-way carry-propagation algorithm as well as an example are presented in Fig. 2. For simplicity, we always propagate ripples through to the last vector. This wastes at most three steps in the last ripple, whose cost is negligible in practice.

3.1.2 Multiplication

Recall from Section 2.1 that Montgomery-friendly primes admit a faster reduction: since $-p^{-1} \equiv 1 \bmod 2^\ell$ we simply have $u = s[i]$, and since $p + 1 = c \cdot 2^e$, the full multiplication by p reduces to the short multiplication by c and a shift. The primes we consider are the nicest case of this family. Because $e \geq (N-1)\ell$, we may write $p + 1 = \delta \cdot 2^{(N-1)\ell}$ where δ fits in a single ℓ_{top}-bit limb. The per-iteration multiplication by p becomes a multiplication by the single limb δ and a shift by $N - 1$ limbs.

The consequence is that the word-by-word reduction is very fast for these primes. In iteration i, eliminating limb i adds $s[i] \cdot \delta$ at index $i + (N - 1)$. For $i \geq 1$, this index is at least N, which is above the range of limbs the loop still has to process, so the elimination never feeds a later iteration. The exception is the first iteration, whose elimination lands in limb $N - 1$ and thus contributes to the last limb to be processed. Apart from this, the iterations are mutually independent, depending only on their own input limb, not on the result of a previous elimination. This allows substantial instruction-level parallelism, where a single carry propagation is sufficient at the end.

The vectorization of Montgomery multiplication is quite different for each format, so we discuss each separately. We do not have a specialized algorithm for squaring, as our attempts gave a negligible advantage in comparison to simply using multiplication.

In all the formats, we make use of the fact that the limb bit-length ℓ is at most 52 bits, leaving $64 - 52 = 12$ bits of headroom in each lane. This often allows us to delay carry propagation to the very end, or until a value needs to re-enter the 52-bit AVX-512 multiplier. In practice, we always have plenty of headroom. For example, schoolbook integer multiplication accumulates at most N 52-bit values in a lane, so an overflow occurs

Algorithm 2 (2×4)-way carry propagation

Input: An array $\boldsymbol{A}$ of n_2 vectors and the lane of the top limb $t \in \{0, 1, 2, 3\}$

1: $L \leftarrow \mathsf{VSETALL}(2^\ell - 1)$ // limb mask
2: $S \leftarrow L$; $S[t] := S[t{+}4] := 2^{64} - 1$
3: **for** $i = 0$ **to** t **do**
4: **for** $j = 0$ **to** $n_2 - 2$ **do** // ripple
5: $C \leftarrow \mathsf{VSRA}(A_j, \ell)$
6: $A_j \leftarrow \mathsf{VAND}(A_j, L)$
7: $A_{j+1} \leftarrow \mathsf{VADD}(A_{j+1}, C)$
8: **end for**
9: $C \leftarrow \mathsf{VSRA}(A_{n_2-1}, \ell)$
10: $A_{n_2-1} \leftarrow \mathsf{VAND}(A_{n_2-1}, S)$
11: **if** $i < t$ **then** // cross-lane shift
12: $C \leftarrow \mathsf{shiftLane}(C)$ // shuffle
13: $A_0 \leftarrow \mathsf{VADD}(A_0, C)$
14: **end if**
15: **end for**

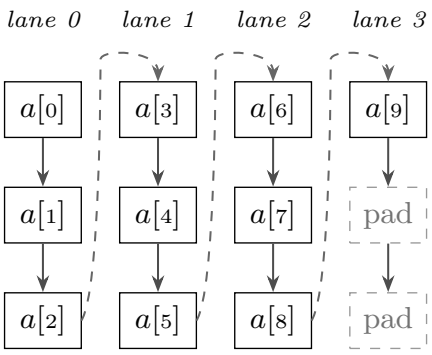

Figure 2: Left: Carry propagation in the 2×4 format. The three additional vectors are: C for the carry, the limb mask L (keeping the low ℓ bits of each lane), and the mask S, which is the same mask as L except that it leaves the top limb of each element unchanged to preserve signs. shiftLane is shorthand for a cross-lane shuffle instruction. The same carry logic is used in multiplication, with more vectors. Right: The propagation for the SQIsign level-V prime ($N = 10$, $t = 3$). Solid arrows are the in-lane ripples; dashed arrows are the cross-lane shifts.

only for $N > 2^{12}$, which would make p orders of magnitude larger than the primes used in isogeny-based cryptography.

(8×1) **multiplication.** The (8×1) implementation repeats the Montgomery multiplication independently in each of the eight lanes.

Integer multiplication. As in [CFGR22], the integer multiplication is computed using one layer of Karatsuba, where the three half-length multiplications are computed using product scanning. One of these is the product of $a_{\text{lo}} + a_{\text{hi}}$ and $b_{\text{lo}} + b_{\text{hi}}$, whose limbs may be $\ell + 1$ bits. To make use of the 52-bit AVX-multiplier, we always select $\ell \leq 51$.

Montgomery reduction. We begin by propagating carries in the integer multiplication output. As explained in Section 2.1, the multiplication by p is replaced by a short multiplication by c. Accordingly, we optimized the carry propagation by exploiting the fact that such multiplication does not touch the upper limbs and hence propagating their carries can be delayed to the very end of the computation. Finally, we carry out the word-by-word reduction in the straightforward way. We experimented with replacing the word-by-word loop by a one-shot computation, using that $-p^{-1} \equiv \delta \cdot 2^{(N-1)\ell} + 1 \bmod 2^{N\ell}$. However, this ended up being less efficient, as it required one extra multiplication when computing $u = t \cdot (-p^{-1}) \equiv t + t[0] \cdot \delta \cdot 2^{(N-1)\ell} \bmod 2^{N\ell}$.

(4×2) **multiplication.** The (4×2) format splits the limbs of an $\mathbb{F}_p$ element across two lanes, which requires handling cross-lane dependencies in the reduction.

Integer multiplication. The algorithm from [OLHA18, Algorithm 2] computes the integer multiplication using product scanning in the two lanes simultaneously. Writing each $\mathbb{F}_p$ element as $a = a_{\text{lo}} + a_{\text{hi}} \cdot 2^{n_4 \ell}$, it computes $a_{\text{lo}} \cdot b$ in the left lane and $a_{\text{hi}} \cdot b$ in the right. The two limbs of a stored in a vector are multiplied by the same limb of b, which is assigned to both lanes by a shuffle. Accumulating over all limb pairs, the result is stored in $n_4 + N$ vectors, where the k-th vector holds $\sum_{i+j=k} a[i] \cdot b[j]$ in the left lane and $\sum_{i+j=k} a[i + n_4] \cdot b[j]$ in the right.

Montgomery reduction. The algorithm [OLHA18, Algorithm 3] interleaves the carry propagation and the word-by-word reduction. For smaller primes, this makes the iterations less independent. At iteration i, it propagates carries from vector i to vector $i+1$, eliminates the left lane of vector i, and folds the right lane into vector $i+(N-1-n_4)$. For $i \leq n_4+1$, the folded value will be input to a later multiplication. Iteration i's fold must therefore complete before the later iteration can multiply. As an example, for the SQIsign level I prime, $N-1-n_4=1$, so that each iteration depends on the previous one, creating a serial dependency chain. To remove the multiplications from the critical path, we implement a reduction algorithm that proceeds in two stages. In the first stage, it computes a carry propagation and right lane to left lane propagation loop. After this pass, the first N limbs of the left lane hold the inputs to the multiplications. In the second stage, we compute the multiplications. For the first iterations, we add the result of the product $t[i] \cdot \delta$ to the left lane, so they do not interfere with later iterations. The only exception is once again at iteration $i=0$, which interferes with the last iteration $i=N-1$, and therefore needs to be handled separately. The algorithm is presented in Algorithm 3. Compared to the algorithm from [OLHA18] (adapted to use the cheaper multiplication by δ), our algorithm is 9% faster for SQIsign level I. However, for higher security levels, there is a bigger gap between dependent iterations, and both algorithms perform the same.

Algorithm 3 (4×2)-way Montgomery reduction (without final reduction mod p)

Input: An array $\boldsymbol{T}$ of $N+n_4$ vectors output by the integer multiplication
Output: Reduced output in $T_N, \ldots, T_{N+n_4-1}$

1: $L \leftarrow \mathsf{VSETALL}(2^\ell - 1)$; $D \leftarrow \mathsf{VSETALL}(\delta)$ // limb mask and top limb of $(p+1)$
2: **for** $i = 0$ **to** $N-2$ **do** // carry-and-fold
3: $T_{i+1} \leftarrow \mathsf{VADD}(T_{i+1}, \mathsf{VSRA}(T_i, \ell))$
4: $T_i \leftarrow \mathsf{VAND}(T_i, L)$;
5: $F \leftarrow (T_i[1], 0, T_i[3], 0, T_i[5], 0, T_i[7], 0)$ // shuffle right lane to left lane
6: $T_{i+n_4} \leftarrow \mathsf{VADD}(T_{i+n_4}, F)$
7: **end for**
8: **for** $i = 0$ **to** $N-2$ **do** // elimination
9: **if** $i < n_4$ **then**
10: $I_{\text{lo}} \leftarrow I_{\text{hi}} \leftarrow (T_i[0], 0, T_i[2], 0, T_i[4], 0, T_i[6], 0)$
11: $l \leftarrow i+N-1$; $h \leftarrow i+N$
12: **else if** $i = n_4$ **then**
13: $I_{\text{lo}} \leftarrow (T_i[0], 0, T_i[2], 0, T_i[4], 0, T_i[6], 0)$
14: $I_{\text{hi}} \leftarrow (0, T_i[0], 0, T_i[2], 0, T_i[4], 0, T_i[6])$
15: $l \leftarrow N+n_4-1$; $h \leftarrow N$
16: **else**
17: $I_{\text{lo}} \leftarrow I_{\text{hi}} \leftarrow (0, T_i[0], 0, T_i[2], 0, T_i[4], 0, T_i[6])$
18: $l \leftarrow i+N-n_4-1$; $h \leftarrow i+N-n_4$
19: **end if**
20: $T_l \leftarrow \mathsf{VMULADDL}(T_l, I_{\text{lo}}, D)$; $H \leftarrow \mathsf{VMULADDH}(\mathsf{VSETALL}(0), I_{\text{hi}}, D)$
21: $T_h \leftarrow \mathsf{VADD}(T_h, \mathsf{VSL}(H, 52-\ell))$
22: **end for**
23: Run the last iteration $i = N-1$ of the loop in line 2
24: Run the last iteration $i = N-1$ of the loop in line 8
25: Fully propagate carries for $T_N, \ldots, T_{N+n_4-1}$ as described in Section 3.1.1

2×4 multiplication. Our novel (2×4)-way multiplication can be seen as a generalization of the (4×2) algorithm to handling cross-lane dependencies across four lanes.

Integer multiplication. We compute an integer multiplication in each lane simultaneously. Writing each $\mathbb{F}_p$ element as $a_0 + a_1 \cdot 2^{n_2\ell} + a_2 \cdot 2^{2 \cdot n_2\ell} + a_3 \cdot 2^{3 \cdot n_2\ell}$, it computes the product $a_j \cdot b$ in the j-th lane. The result is stored in $N+n_2$ vectors.

Montgomery reduction. As in the (4×2) case, we split the reduction into a carry-and-fold stage and an elimination stage. The fold shifts the contents of each lane down one lane. At iteration i, we add lanes 1, 2 and 3 of vector i to lanes 0, 1 and 2 of vector $i + n_2$, independently within each element's half of the vector. After the carry-and-fold pass, lane 0 of vector i holds the input u for the i-th elimination.

3.1.3 Other operations

We implement some additional operations for each format.

Division by 2. For an odd integer a, we have

$$a/2 \equiv (a-1)/2 + (p+1)/2 \equiv (a \gg 1) + c \cdot 2^{e-1} \bmod p.$$

Thus, we can divide by 2 using a right-shift, adding $c \cdot 2^{e-1}$ when a is odd. This is computed in parallel for each $\mathbb{F}_p$ element.

Inversion. The inversion of k $\mathbb{F}_p$-elements can be computed using Montgomery's trick, requiring $3 \cdot (k-1)$ $\mathbb{F}_p$ multiplications and one $\mathbb{F}_p$ inversion. We compute the multiplications in vectorized form. However, the single inversion is computed with a non-vector implementation of Pornin's constant-time binary GCD algorithm [Por20].

Square root. For a quadratic residue $a \in \mathbb{F}_p$ a square root can be computed as $a^{(p+1)/4} \equiv a^{c \cdot 2^{e-2}} \bmod p$. To compute k square roots in parallel, we simply compute this exponentiation with our vectorized multiplication. We first compute $a^c \bmod p$ using a short square-and-multiply loop, then we obtain the result with $e-2$ consecutive squarings.

3.2 Quadratic Field Extension

For the quadratic extension field $\mathbb{F}_{p^2}$, we write each $\mathbb{F}_{p^2}$ element as $a = a_0 + a_1 i$, where $i^2 = -1$ and $a_0, a_1 \in \mathbb{F}_p$ are the real and imaginary components of a. For the $\mathbb{F}_{p^2}$ layer, we adopt the notion of $(k \times y \times z)$-way parallelism from [CFGR22]: k independent $\mathbb{F}_{p^2}$ operations are performed at once, where each of those computes y parallel $\mathbb{F}_p$ operations. The number of lanes used per $\mathbb{F}_p$ operation is given by $z = 8/(k \cdot y)$.

Algorithm 4 $\mathbb{F}_{p^2}$ multiplication with 1-, 2-, and 4-way parallelism over $\mathbb{F}_p$.

Input: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$
Output: $r = r_0 + r_1 i = a \cdot b \in \mathbb{F}_{p^2}$

1-way

1: $s_1 \leftarrow a_0 + a_1$
2: $s_2 \leftarrow b_0 + b_1$
3: $t_1 \leftarrow a_0 b_0$
4: $t_2 \leftarrow a_1 b_1$
5: $t_3 \leftarrow s_1 s_2$
6: $r_0 \leftarrow t_1 - t_2$
7: $r_1 \leftarrow t_3 - t_1 - t_2$

2-way

1: $s_1 \leftarrow a_0 b_0$	$t_1 \leftarrow a_0 b_1$
2: $s_2 \leftarrow a_1 b_0$	$t_2 \leftarrow a_1 b_1$
3: $r_0 \leftarrow s_1 - t_2$	$r_1 \leftarrow s_2 + t_1$

4-way

1: $s_1 \leftarrow a_0 b_0$	$t_1 \leftarrow a_0 b_1$	$s_2 \leftarrow a_1 b_0$	$t_2 \leftarrow a_1 b_1$
2: $r_0 \leftarrow s_1 - t_2$	$r_1 \leftarrow s_2 + t_1$		

For multiplication, [CFGR22] provide formulas with 1-, 2- and 4-way parallelism over $\mathbb{F}_p$ (Algorithm 4), as well as specialized 1- and 2-way formulas for squaring (Algorithm 5). We build the vectorized $\mathbb{F}_{p^2}$ operations by evaluating one of the formulas with our vectorized $\mathbb{F}_p$ routines. Evaluating a t-way formula with the $(m \times z)$ operations for $\mathbb{F}_p$ yields a

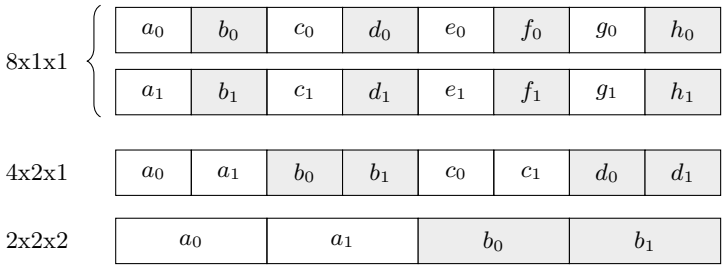

Figure 3: Contents of one vector under the three vectorized $\mathbb{F}_{p^2}$ formats, for elements $a, b, \ldots, h \in \mathbb{F}_{p^2}$ with $a = a_0 + a_1 i$. Each slot holds one $\mathbb{F}_p$ component, stored in the $\mathbb{F}_p$ format of Fig. 1. The $(8 \times 1 \times 1)$ format uses two vectors, one for the real and one for the imaginary components; shading distinguishes adjacent elements.

$(m/t \times t \times z)$-way $\mathbb{F}_{p^2}$ implementation. With the (8×1) operations, we get the following $\mathbb{F}_{p^2}$ formats:

- The 1-way formula gives an $(8 \times 1 \times 1)$-way multiplication. We place all the real components in one 512-bit register in (8×1) format and all the imaginary components in another one, and evaluate the 1-way formula lane-wise.
- The 2-way formula gives a $(4 \times 2 \times 1)$-way multiplication. We store both components, real and imaginary, of the four $\mathbb{F}_{p^2}$ elements in one 512-bit register, packed as $(a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1)$.
- The 4-way formula gives a $(2 \times 4 \times 1)$-way multiplication. Two elements do not fill all eight lanes in the (8×1)-format, so we use the redundant packing $(a_0, 0, a_1, 0, b_0, 0, b_1, 0)$.

Similarly, the (4×2) multiplier yields $(4 \times 1 \times 2)$-way multiplication from the 1-way formula and $(2 \times 2 \times 2)$-way multiplication from the 2-way formula. As observed in [CFGR22], the (8×1) multiplier has the lowest cost per instance, so it usually gives the fastest $\mathbb{F}_{p^2}$ operations. The exception is for the case of the 4-way formula: the $(2 \times 4 \times 1)$-way multiplication uses a redundant representation and has no specialized squaring operation. Whether $(2 \times 2 \times 2)$ or $(2 \times 4 \times 1)$ is fastest depends on the relative performance of the (8×1) and (4×2) multiplier (cf. Section 3.5). We illustrate the $(8 \times 1 \times 1)$, $(4 \times 2 \times 1)$ and $(2 \times 2 \times 2)$ formats in Fig. 3.

Algorithm 5 $\mathbb{F}_{p^2}$ squaring with 1- and 2-way parallelism over $\mathbb{F}_p$.

Input: $a = a_0 + a_1 i \in \mathbb{F}_{p^2}$
Output: $r = r_0 + r_1 i = a^2 \in \mathbb{F}_{p^2}$

1-way	*2-way*
1: $s_1 \leftarrow a_0 + a_1$	1: $s_1 \leftarrow a_0 + a_1$ $t_1 \leftarrow a_0 + a_0$
2: $s_2 \leftarrow a_0 - a_1$	2: $s_2 \leftarrow a_0 - a_1$
3: $s_3 \leftarrow a_0 + a_0$	3: $r_0 \leftarrow s_1 \cdot s_2$ $r_1 \leftarrow t_1 \cdot a_1$
4: $r_0 \leftarrow s_1 \cdot s_2$	
5: $r_1 \leftarrow s_3 \cdot a_1$	

3.2.1 Other operations

Besides additions, subtractions, multiplications and squarings, we also extend our additional operations from $\mathbb{F}_p$ to $\mathbb{F}_{p^2}$.

Division by 2. Division by 2 is $\mathbb{F}_p$-linear, so we simply divide component-wise.

Inversion. For $a = a_0 + a_1 i \in \mathbb{F}_{p^2}^{\times}$, we have $(a_0 + a_1 i)(a_0 - a_1 i) = a_0^2 + a_1^2$, which must be non-zero as a has no zero-divisors. Hence, $a^{-1} = (a_0 - a_1 i)(a_0^2 + a_1^2)^{-1}$, computed using our $\mathbb{F}_p$ operations with one inversion, two squarings and two multiplications.

Square root. Usually an $\mathbb{F}_{p^2}$ square root costs two $\mathbb{F}_p$ square roots and two $\mathbb{F}_p$ inversions. However, since $p = 3 \bmod 4$, we can apply the faster algorithm from [AAA+25] which only requires one square root and one exponentiation by $(p-3)/4$. Our code generator produces a short addition chain for this exponentiation using McLoughlin's `addchain` library [McL21].

3.3 Prime-dependent optimizations

There are several optimizations for multiplication and squaring available depending on the prime p. First, it is often possible to skip the final conditional subtraction in the Montgomery reduction, and leave the $\mathbb{F}_p$ elements reduced modulo $2p$ instead of p. Further speedups can be obtained by computing the formulas for $\mathbb{F}_{p^2}$ multiplication and squaring over the integers, delaying carry propagation and modular reduction to the very end. The two strategies are coupled: enlarging the $\mathbb{F}_p$ elements to the range $[0, 2p)$ increases the bit-length of the top limb by one, shrinking the headroom for lazy reduction on the $\mathbb{F}_{p^2}$ layer.

Over the integers, with base-field elements in $[0, m)$ for $m \in \{p, 2p\}$, the product of $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$ is $r_0 + r_1 i$ with

$$r_0 = a_0 b_0 - a_1 b_1 \in (-m^2, m^2) \text{ and } r_1 = a_0 b_1 + a_1 b_0 \in [0, 2m^2).$$

Montgomery reduction requires that the input is in $[0, pR)$, therefore, we need to ensure $2m^2 \leq pR$ and find a non-negative representative for r_0. Additionally, we need to ensure that the inputs to the AVX-512 multiplier never exceed 52-bits (in particular, they cannot be negative). Whether these conditions can be satisfied depends on the $\mathbb{F}_{p^2}$ formula used, the $\mathbb{F}_p$ multiplier, and the available headroom given by ℓ and ℓ_{top}.

When reducing modulo $m = 2p$, the requirement $2m^2 \leq pR$ is equivalent to $8p \leq R$. With the standard Montgomery radix $R = 2^{N\ell}$, this holds if and only if $\ell_{\mathsf{top}} \leq \ell - 3$. However, when $8p > R$, it is possible to change the Montgomery radix so that we can still reduce modulo $2p$. By instead choosing the extended radix $R_{\text{ext}} = 2^{(N+1)\ell}$, and thereby doing one extra iteration of the word-by-word loop, we can easily satisfy the constraint $8p \leq R_{\text{ext}}$. The cost of one extra iteration of the loop is much cheaper than reducing modulo p with a conditional subtraction. As we will see later, the larger range can also be useful when finding a non-negative representative for r_0.

The most restrictive case in terms of headroom is the $(8 \times 1 \times 1)$ multiplication, as it is the only format that applies two layers of Karatsuba. Without carry propagation, each layer of Karatsuba computes a product $(a_{\text{lo}} + a_{\text{hi}}) \cdot (b_{\text{lo}} + b_{\text{hi}})$, increasing the limbs of the operands by one bit, thus growing closer to the 52-bit threshold for the AVX-512 multiplier. We therefore detail how we implement $(8 \times 1 \times 1)$ multiplication depending on ℓ and ℓ_{top}, as it largely determines the strategy we apply, in particular which modulus we choose and whether we extend the Montgomery radix.

- $\ell \leq 50$: We have at least two bits of headroom, allowing us to fully implement lazy reduction on the $\mathbb{F}_{p^2}$ layer, delaying carry propagation and modular reduction to the very end. However, there are some nuances related to the top limb:
  - $\ell_{\mathsf{top}} \leq \ell - 2$: We choose modulus $m = 2p$ and use the extended Montgomery radix R_{ext}. To make r_0 non-negative, we always add $p \cdot 2^{N\ell}$. Since $\ell_{\mathsf{top}} \leq \ell - 2$ we have $4p < 2^{N\ell}$, ensuring that $p \cdot 2^{N\ell} + r_0 > p \cdot 2^{N\ell} - 4p^2 > 0$. At the same time $p \cdot 2^{N\ell} + r_0 < p(4p + 2^{N\ell}) < p2^{N\ell+1} < pR_{\text{ext}}$, so it is still within bounds.

  - $\ell_{\mathsf{top}} = \ell - 1$: Since $4p > 2^{N\ell}$, we can no longer always add $p \cdot 2^{N\ell}$ to r_0 to ensure it is non-negative. Instead, we conditionally add $p \cdot R_{\text{ext}}$, which is slightly slower, as it requires an extra carry propagation to determine the sign of r_0.
  - $\ell_{\mathsf{top}} = \ell < 50$: With modulus $m = 2p$, the top limb becomes one bit larger than the other limbs. However, since $\ell_{\mathsf{top}} + 1 \leq 50$, we still have enough headroom.
  - $\ell_{\mathsf{top}} = \ell = 50$: To ensure two bits of headroom in the top limb, we have to choose modulus $m = p$. Despite this, we still use R_{ext} to ensure r_1 is in the range $[0, pR_{\text{ext}})$.

- $\ell = 51$: We have only one bit of headroom, but by propagating carries after computing $s_1 \leftarrow a_0 + a_1$ and $s_2 \leftarrow b_0 + b_1$ in the $\mathbb{F}_{p^2}$ formula, the first $N - 1$ limbs fit within 51-bits. However, the top limb grows by one bit, leading again to several cases:
  - $\ell_{\mathsf{top}} \leq 49$: With modulus $m = 2p$, the top limb of s_1 and s_2 after carry propagation is at most 51 bits, so we can apply the (8×1) integer multiplication. We use R_{ext} and since $4p < 2^{N\ell}$ we add $p \cdot 2^{N\ell}$ to r_0.
  - $\ell_{\mathsf{top}} = 50$: We can no longer use modulus $m = 2p$, as the top limbs would grow to 52 bits, too large for the (8×1) multiplier. We still use R_{ext} and conditionally add $p \cdot R_{\text{ext}}$ to r_0.
  - $\ell_{\mathsf{top}} = 51$: No headroom is available, so we cannot implement lazy reduction on the $\mathbb{F}_{p^2}$ layer, nor is there any advantage to using R_{ext}.

Our code generator chooses the limb length ℓ that ensures the best performance for each prime. With $\ell \leq 51$, the minimal number of limbs needed to store an $\mathbb{F}_p$ element is $N = \lceil \log_2(p)/51 \rceil$. The smallest limb length which does not increase the number of limbs is $\ell_{\min} = \lceil \log_2(p)/N \rceil$. We never go below this bound, as the cost of multiplication grows quadratically in N, so the cost of adding an extra limb is greater than what we get from lazy reduction. We also never go beyond the largest $\ell_{\max} \leq 51$ such that our assumption $e \geq (N-1)\ell$ no longer holds. Within the range $[\ell_{\min}, \ell_{\max}]$, the code generator selects the limb length that gives the fastest multiplication according to our analysis. Our first priority is to enable reduction modulo $2p$ by having $\ell_{\mathsf{top}} < 50$. Our second priority is to have $\ell < 51$, which avoids propagating carries after computing the sums $s_1 \leftarrow a_0 + a_1$ and $s_2 \leftarrow b_0 + b_1$ in the $(8 \times 1 \times 1)$ multiplication. If there are multiple candidates, we prefer a gap between ℓ and ℓ_{top} of at least two; failing that, a gap of one; and otherwise a gap of zero. In case of a tie, we pick the shortest ℓ among the candidates.

3.4 Conversions between representations

The code generator provides functions for converting between vectorized representations which we rely on heavily when vectorizing the later stages. These conversions are very cheap as they can be implemented with a single pass over the vectors. This lets us lift a group of independent multiplications from a slower multiplier to a faster multiplier. For example, the benchmarks in Section 3.5 show that for two independent $(4 \times 2 \times 1)$ multiplications, it is faster to convert the operands to the $(8 \times 1 \times 1)$ format, perform a single multiplication there and convert back. Hence, we restructure our code to exploit this as much as possible. More generally, our guiding principle for implementing the $(2^n, 2^n)$-isogenies and SQIsign verification is not to commit to a single vectorization format, but to leverage these cheap conversions to work in whichever format best suits each part of the computation.

We also provide conversions to and from an unvectorized representation, storing a field element as a plain array of integers. This conversion is quite expensive, but remains useful in two situations. First, it allows plugging our vectorized $(2^n, 2^n)$-isogeny implementation into an otherwise unvectorized codebase. One can convert to the vectorized representation

once, run the vectorized $(2^n, 2^n)$-isogeny code, and then convert back. Second, in some cases, parts of a computation do not achieve a meaningful gain from vectorization, and are cleaner to perform with unvectorized code. We apply this approach for inversion over $\mathbb{F}_p$: inputs are converted to unvectorized form, run through Pornin's binary GCD algorithm, and then converted back.

Table 3: Cost of $\mathbb{F}_p$ multiplication for the assembly baseline and the 2-, 4-, and 8-way AVX-512 implementations.

Level	Format	Cycles	Cycles/inst.	Speedup (×)
I	asm	33	33	1.00
	(2×4)	92	46	0.72
	(4×2)	97	24	1.36
	(8×1)	76	10	3.47
III	asm	72	72	1.00
	(2×4)	154	77	0.94
	(4×2)	121	30	2.38
	(8×1)	140	18	4.11
V	asm	104	104	1.00
	(2×4)	164	82	1.27
	(4×2)	192	48	2.17
	(8×1)	188	24	4.43

Table 4: Cost of $\mathbb{F}_{p^2}$ multiplication and squaring for the assembly baseline and the 2-, 4-, and 8-way AVX-512 implementations.

		Multiplication			Squaring		
Level	Format	Cycles	Cyc/inst.	×	Cycles	Cyc/inst.	×
I	asm	108	108	1.00	74	74	1.00
	$(2 \times 2 \times 2)$	126	63	1.71	122	61	1.21
	$(2 \times 4 \times 1)$	118	59	1.83	125	63	1.18
	$(4 \times 2 \times 1)$	144	36	3.00	105	26	2.82
	$(8 \times 1 \times 1)$	204	26	4.24	165	21	3.59
	[WWW+26], 8-way	304	38	2.84	176	22	3.36
III	asm	265	265	1.00	161	161	1.00
	$(2 \times 2 \times 2)$	186	93	2.85	148	74	2.18
	$(2 \times 4 \times 1)$	201	101	2.64	213	107	1.51
	$(4 \times 2 \times 1)$	241	60	4.40	166	42	3.88
	$(8 \times 1 \times 1)$	364	46	5.82	281	35	4.58
V	asm	383	383	1.00	221	221	1.00
	$(2 \times 2 \times 2)$	280	140	2.74	232	116	1.91
	$(2 \times 4 \times 1)$	254	127	3.02	264	132	1.67
	$(4 \times 2 \times 1)$	346	87	4.43	243	61	3.64
	$(8 \times 1 \times 1)$	518	65	5.92	410	51	4.31

3.5 Benchmarks

To demonstrate the performance of our vectorization approach using the code generator, we present benchmarks for $\mathbb{F}_p$ multiplication in Table 3, the benchmarks for $\mathbb{F}_{p^2}$ multiplication and squaring in Table 4 and the cost of converting between representations in Table 5. The results correspond to the SQIsign primes from Round 2. The implementations were all

benchmarked on a 16-core Intel Xeon Silver 4314 (Ice Lake-SP) server running at 2.4GHz.

For the $\mathbb{F}_p$ multiplication, we note that, as the prime grows, the quadratic-in-N cost of multiplication increasingly outweighs the cost of carry-propagation of the formats with less parallelism (i.e., (2×4) and (4×2)). Hence, the (4×2) format already outperforms the baseline implementation in assembly at level I, and the (2×4) format at level V. We also observe that the level III prime enjoys a relatively faster $\mathbb{F}_{p^2}$ squaring, as the code generator is able to use limb length $\ell = 49$ and $\ell_{\mathsf{top}} = 40$, which enables the fastest multiplication strategy in Section 3.3. For levels I and V, $\ell = 51$ is the smallest possible limb length that does not increase the number of limbs N.

For comparison, Table 4 also includes the performance results for $\mathbb{F}_{p^2}$ multiplication and squaring from Wang et al. [WWW+26], who also presents a vectorized implementation of SQIsign using AVX-512 (Wang et al. only report $\mathbb{F}_{p^2}$ results for level I). As can be seen, our 8-way multiplier is close to 1.5× faster.

Table 5: Cost in cycles of converting between representations of $\mathbb{F}_{p^2}$ elements at each security level. "U" is the unvectorized form; "2", "4", "8" are the vectorized formats. The arrow points to the target, e.g., U → 2 converts U into the 2-way format and U ← 2 does the reverse. Conversion between 2-way and 8-way is done via 4-way and is omitted.

Vectorize / devectorize			
	I	III	V
U → 2	253	334	484
U ← 2	439	756	1120
U → 4	407	538	863
U ← 4	704	1213	2047
U → 8	719	1048	1540
U ← 8	1412	2409	4071

Repack between formats			
	I	III	V
2 → 4	9	14	17
2 ← 4	9	14	17
4 → 8	17	28	35
4 ← 8	17	28	45

4 Vectorized (2,2)-Isogeny Chains in the Theta Model

In this section, we give a high-level overview of the algorithms required to compute $(2, 2)$-isogeny chains. We focus primarily on how the algorithms of [DMPR24] can be augmented to leverage the available two-, four- and eight-way $\mathbb{F}_{p^2}$ arithmetic.

For the description of isogenies in the theta model we essentially need two building blocks to realize a vectorized implementation: the Hadamard transform $\mathcal{H}(P)$ and coordinate-wise multiplication $P \star Q$. The special case of $\mathcal{S}(P) = P \star P$ is referred to as the squaring operator.

For level-two theta coordinates in dimension two, the Hadamard transform can be computed with eight additions, which can be carried out through two four-way addition calls, as shown in Algorithm 6. For points P and Q, coordinate wise multiplication $P \star Q = (P^x \cdot Q^x : P^y \cdot Q^y : P^z \cdot Q^z : P^t \cdot Q^t)$ costs one four-way multiplication. The special case of $\mathcal{S}(P)$ then costs a single four-way square.

Algorithm 6 Vectorized Hadamard

Input: Coordinates of a theta point $P = (x : y : z : t)$
Output: Coordinates of the Hadamard transform $\mathcal{H}(P) = (x' : y' : z' : t')$

1: $t_1 \leftarrow x + y$ $t_2 \leftarrow x - y$ $t_3 \leftarrow z + t$ $t_4 \leftarrow z - t$
2: $x' \leftarrow t_1 + t_3$ $y' \leftarrow t_2 + t_4$ $z' \leftarrow t_1 - t_3$ $t' \leftarrow t_2 - t_4$

Cost: $2\mathbf{A}_4$

Using these building blocks, it then becomes easy to describe several of the core algorithms for $(2, 2)$-isogeny chains. Concretely, we are able to write down both doubling

and point evaluation[3] only using the operations $\mathcal{H}$, $\mathcal{S}$ and $\star$, which implies that these functions can be immediately vectorized using four-way arithmetic.

However, it is possible to improve performance further. When processing pairs of points in parallel (for example, when working with the kernel of a $(2,2)$-isogeny: $\ker\Phi = \langle P, Q\rangle$), there is a trivial extension to use eight-way arithmetic. A pair of Hadamard transforms, denoted as $\mathcal{H}_2(P_1, P_2)$, costs $2\mathbf{A}_8$, coordinate wise multiplication of a pair of points, denoted as $(P_1, P_2) \star_2 (Q_1, Q_2)$, costs a single $\mathbf{M}_8$, and coordinate wise squaring of a pair of points, denoted as $\mathcal{S}_2(P_1, P_2)$, costs a single $\mathbf{S}_8$.

For the remainder of this section, we give explicit algorithms using vectorized finite field arithmetic for point doubling, and codomain and evaluation computations of $(2,2)$-isogenies in the theta model. From these three simple functions, we give a description of the $(2^n, 2^n)$-chain itself, showing how well suited these isogeny chains are to eight-way arithmetic using SIMD instructions.

Subsequently, two additional sections are devoted to a high-level discussion of special maps that are required when computing chains of $(2,2)$-isogenies between elliptic curve products $\Phi : E_1 \times E_2 \to E_3 \times E_4$. The first one is known as a *gluing isogeny* $\Gamma : E_1 \times E_2 \to A$, which, given two elliptic curves (and the torsion data from the kernel), computes a $(2,2)$-isogeny where the codomain is an abelian surface described using dimension two, level-two theta coordinates. The second one is the *splitting isomorphism* $\sigma : A \to E_1 \times E_2$, which converts explicitly from the level-two theta null point of a product structure to the pair of elliptic curves in the Montgomery model. Less explicit discussions of the vectorization of these maps are given, but complete vectorized implementations of all the necessary functions are available in our accompanying code.

4.1 Vectorized Arithmetic in the Level-Two Theta Model

The core of the work in [DMPR24] is based on a proper understanding of the *duplication formula*, which gives concrete relations between points $P+Q$ and $P-Q$ and the images $f(P), f(Q)$ and $\tilde{f}(P), \tilde{f}(Q)$, where $f : A \to B$ is a $(2,2)$-isogeny and $\tilde{f}$ is its dual. We denote the coordinates of the theta null point of A by $\theta_A = (a : b : c : d)$ and the *dual* coordinates of the theta null point of B by $\tilde{\theta}^B = (\alpha : \beta : \gamma : \delta)$. Complete definitions and proofs of correctness are not provided here. The interested reader is referred to [DMPR24].

When computing doubles in the theta model, a pre-computation must be made for each theta null point $\theta_A = (a : b : c : d)$ to compute $\lambda = (a^{-1} : b^{-1} : c^{-1} : d^{-1})$ and $\tilde{\lambda} = (\alpha^{-2} : \beta^{-2} : \gamma^{-2} : \delta^{-2})$. Use of the duplication formula [DMPR24, Equation 2] recovers $(\alpha^2 : \beta^2 : \gamma^2 : \delta^2) = \mathcal{H}(a^2 : b^2 : c^2 : d^2)$, showing that the doubling computation can be done from knowledge of θ_A alone. A vectorized implementation of this precomputation is given in Algorithm 7, where it is assumed that we can work projectively and, importantly, that none of the values of $a, b, c, d, \alpha^2, \beta^2, \gamma^2$ or δ^2 are zero.

Following [DMPR24, Algorithm 4], doubling can be written using the operations $\mathcal{H}$, $\mathcal{S}$ and $\star$, where the four-way parallelism is implicit. Accordingly, pairs of points can be doubled using eight-way arithmetic, as shown in Algorithm 8.

4.2 Vectorized (2, 2)-Isogenies in the Theta Model

Following [DMPR24, Algorithm 6], evaluation, much like doubling, can be written entirely with the core operators, in which four-way vectorization is implicit. In Algorithm 9, we write explicitly the evaluation of a pair of points using eight-way arithmetic.

The codomain computation does not directly exhibit the same level of parallelism. However, after specializing the projective codomain computation of [DMPR24, Algorithm

[3] The fact that we have "closed-forms" of both doubling and evaluation is actually no surprise. In the theta model, the double of a point is essentially computed by evaluating a point through f and $\tilde{f}$, so if we can write down evaluation with these simple functions then we can also write doubling in a similar fashion.

Algorithm 7 Vectorized arithmetic precomputation

Input: The theta null point $\theta_A = (a : b : c : d)$
Output: The constants $\lambda = (a^{-1} : b^{-1} : c^{-1} : d^{-1})$ and $\tilde{\lambda} = (\alpha^{-2} : \beta^{-2} : \gamma^{-2} : \delta^{-2})$

$(\alpha^2 : \beta^2 : \gamma^2 : \delta^2) \leftarrow \mathcal{S}(\theta_A)$ //$\mathbf{S}_4$
$(\alpha^2 : \beta^2 : \gamma^2 : \delta^2) \leftarrow \mathcal{H}(\alpha^2 : \beta^2 : \gamma^2 : \delta^2)$ //$2\mathbf{A}_4$
$T_1 \leftarrow (\alpha^2 : \gamma^2 : a : c)$
$T_2 \leftarrow (\beta^2 : \delta^2 : b : d)$
$T_3 \leftarrow T_1 \star T_2$ //$\mathbf{M}_4$
$T_1 \leftarrow (T_3^t : T_3^t : T_3^z : T_3^z)$
$T_2 \leftarrow (T_3^y : T_3^y : T_3^x : T_3^x)$
$\lambda \leftarrow (b : a : d : c)$
$\tilde{\lambda} \leftarrow (\beta^2 : \alpha^2 : \delta^2 : \gamma^2)$
$\lambda, \tilde{\lambda} \leftarrow (\lambda, \tilde{\lambda}) \star_2 (T_1, T_2)$ //$\mathbf{M}_8$
return $\lambda, \tilde{\lambda}$ Cost: $\mathbf{M}_8 + \mathbf{M}_4 + \mathbf{S}_4 + 2\mathbf{A}_4$

Algorithm 8 Vectorized doubling of a pair of level-two theta points

Input: The theta points P, Q together with two precomputed constants λ and $\tilde{\lambda}$
Output: The theta points $[2]P$ and $[2]Q$.

$(P_2, Q_2) \leftarrow \mathcal{S}_2(P, Q)$ //$\mathbf{S}_8$
$(P_2, Q_2) \leftarrow \mathcal{H}_2(P_2, Q_2)$ //$2\mathbf{A}_8$
$(P_2, Q_2) \leftarrow \mathcal{S}_2(P_2, Q_2)$ //$\mathbf{S}_8$
$(P_2, Q_2) \leftarrow (P_2, Q_2) \star_2 (\tilde{\lambda}, \tilde{\lambda})$ //$\mathbf{M}_8$
$(P_2, Q_2) \leftarrow \mathcal{H}_2(P_2, Q_2)$ //$2\mathbf{A}_8$
$(P_2, Q_2) \leftarrow (P_2, Q_2) \star_2 (\lambda, \lambda)$ //$\mathbf{M}_8$
return (P_2, Q_2) Cost: $2\mathbf{M}_8 + 2\mathbf{S}_8 + 4\mathbf{A}_8$

Algorithm 9 Vectorized $(2, 2)$-isogeny evaluation of a pair of points

Input: A pair of theta points P, Q together with the precomputed constant I, the inverse of the dual theta-null point of the codomain.
Output: The images of P and Q under the action of the isogeny $f : A \to B$.

$f(P), f(Q) \leftarrow \mathcal{S}_2(P, Q)$ //$\mathbf{S}_8$
$f(P), f(Q) \leftarrow \mathcal{H}_2(f(P), f(Q))$ //$2\mathbf{A}_8$
$f(P), f(Q) \leftarrow (f(P), f(Q)) \star_2 (I, I)$ //$\mathbf{M}_8$
$f(P), f(Q) \leftarrow \mathcal{H}_2(f(P), f(Q))$ //$2\mathbf{A}_8$
return $f(P), f(Q)$ Cost: $\mathbf{M}_8 + \mathbf{S}_8 + 4\mathbf{A}_8$

7], we find a four-way vectorized implementation that uses eight-way multiplication by leveraging the fast conversion between data types. This is shown in Algorithm 10.

4.3 Vectorized $(2, 2)$-Isogeny Chains

As an isogeny chain is computed codomain by codomain, the intuition is that speedups from vectorization would come from improvements within each step, rather than from the chain computation directly. However, there are a few elements of the two-dimensional chain that make it particularly attractive for a vectorized implementation.

At a high level, a chain of 2- or $(2, 2)$-isogenies begins with the generators of the kernel, which are points of order 2^n. The generators are repeatedly doubled to recover points of order two [4], and this point is used to compute the codomain of the isogeny. The kernel

[4]For performance reasons, in the $(2, 2)$-isogeny chain in the theta model we work with points of order eight above the kernel of order two. Thus, the input kernel generators have order 2^{n+2} and each step takes as input a pair of points of order 2^{1+2}. But from the perspective of describing the chain this is not important.

Algorithm 10 Vectorized $(2,2)$-isogeny codomain

Input: Theta points T_1'' and T_2'' of order eight such that $\ker(f) = \langle [4]T_1'' \oplus [4]T_2'' \rangle$.
Output: The null point θ_B of the codomain $f : A \to B$ together with the inverse of the dual theta-point I for computing evaluations.

$K_1, K_2 \leftarrow \mathcal{S}_2(T_1'', T_2'')$ //$\mathbf{S}_8$
$K_1, K_2 \leftarrow \mathcal{H}_2(K_1, K_2)$ //$2\mathbf{A}_8$
$\theta_B \leftarrow K_2$
$S_1 \leftarrow (K_1^x : K_2^x : K_2^z : 1)$
$S_2 \leftarrow (K_2^y : K_1^y : K_2^t : 1)$
$S_2 \leftarrow S_1 \star S_2$ //$\mathbf{M}_4$
$S_1 \leftarrow (S_2^x : S_2^y : S_2^x : S_2^y)$
$S_2 \leftarrow (S_2^z : S_2^z : 1 : 1)$
$I \leftarrow (K_1^y : K_1^x : 1 : 1)$
$\theta_B, I \leftarrow (\theta_B, I) \star_2 (S_1, S_2)$ //$\mathbf{M}_8$
$I \leftarrow (I^x : I^y : \theta_B^t : \theta_B^z)$
$\theta_B \leftarrow \mathcal{H}(\theta_B)$ //$2\mathbf{A}_4$
return θ_B, I Cost: $\mathbf{M}_8 + \mathbf{S}_8 + 2\mathbf{A}_8 + \mathbf{M}_4 + 2\mathbf{A}_4$

generators are evaluated through this isogeny and the chain continues with this new point of order 2^{n-1}, until the full chain is completed after n steps and the codomain is returned.

In cryptographic applications, in dimension one the kernel of the 2^n-isogeny is generated by a single point. In contrast, in dimension two, the kernel is generated by a pair of points, both of which must be doubled at each step. This means that a vectorized implementation of the $(2,2)$-chain can vectorize both the doubling of the kernel generators and the evaluation of the points through the isogeny at each step. From this perspective, the dimension-two isogeny chain is not only vectorization friendly at each step, but also from the perspective of the chain itself.

Additionally, as described above, the isogeny chain requires a quadratic number of point doublings but only n evaluation and codomain computations. The use of alternative computation strategies (such as balanced or optimized strategies) results in a chain computation that only requires a quasi-linear number of doubles and evaluations at the memory cost of storing intermediate points through the computation.

For the case of one-dimensional isogeny chains [CFGR22], the application of these strategies opens up optimizations in vectorized implementations as the stored strategy points can be evaluated in parallel. However, for the cases we consider here, we find that regardless of the strategies used, the points to be evaluated always appear in pairs. As we use the optimal eight-way finite field arithmetic when processing a pair of points, for $(2,2)$-isogeny chains, the fastest method is also the simplest.

We note that for cryptographic applications such as SQIsign the description of Algorithm 11 is incomplete, as it misses the initial gluing isogeny as well as the final splitting to elliptic curves in the Montgomery model. These two final stages are briefly summarized below; for a complete algorithmic description readers are referred to [AAA+24, Algorithm 8.47] or our accompanying implementation.

4.4 Vectorized Gluing

The gluing isogeny is a $(2,2)$-isogeny $\Gamma\colon E_1 \times E_2 \to A$ from a product of elliptic curves to a principally polarized abelian surface A. In this subsection, we discuss two approaches in the theta model that fix a level-2 theta structure $\theta^{E_1 \times E_2}(P) = M \cdot (P_1 \otimes P_2)$, where $P_i = (x_i : z_i)$ are Kummer-line coordinates, $M \in \mathrm{GL}_4(\mathbb{F}_q)$ is a change-of-coordinates matrix determined by a symplectic basis of $(E_1 \times E_2)[4]$ compatible with $\ker \Phi$, and images are computed through the duplication formula $\mathcal{H}\big(\theta(P+Q) \star \theta(P-Q)\big) = \widetilde{\theta^J}(\Phi(P)) \star \widetilde{\theta^J}(\Phi(Q))$,

Algorithm 11 Vectorized $(2^n, 2^n)$-Isogeny chain using a balanced strategy

Input: The theta null point θ_A of the domain A and the theta points P, Q of order 2^{n+2} such that $\ker \Phi = \langle [4]P, [4]Q \rangle$.
Output: The theta null point θ_B of the codomain B of the $(2^n, 2^n)$-isogeny $\Phi : A \to B$.

```
space ← ⌈log_2(n)⌉ + 1
strategy_pairs ← array of length space which holds pairs of theta points
orders ← array of length space which holds integers
strategy_pairs[0] ← (P, Q)
orders[0] ← n
k ← 0
θ_B ← θ_A
while k ≥ 0 and orders[k] ≠ 0 do
  consts ← ArithmeticPrecomp(θ_B)
  while orders[k] ≠ 1 do
    k ← k + 1
    n ← ⌊orders[k − 1]/2⌋
    strategy_pairs[k] ← strategy_pairs[k − 1]
    for i from 0 up to n − 1 do
      strategy_pairs[k] ← DoublePair(strategy_pairs[k], consts)
    end for
    orders[k] ← orders[k − 1] − n
  end while
  θ_B, I ← Codomain(strategy_pairs[k])
  for i from 0 up to k − 1 do
    strategy_pairs[i] ← EvalPair(strategy_pairs[i], I)
    orders[i] ← orders[i] − 1
  end for
  k ← k − 1
end while
return θ_B
```

with $\mathcal{H}$ the Hadamard transform and $\star$ the coordinate-wise product.

Vectorized doubling of the Gluing Kernel. For a gluing isogeny $\Gamma : E_1 \times E_2 \to A$, the kernel is generated by a pair of points on a pair of curves: $\ker \Gamma = \langle (P_1, P_2), (Q_1, Q_2) \rangle$ for $P_1, Q_1 \in E_1$ and $P_2, Q_2 \in E_2$. As a result, when we are given the kernel which generates the full isogeny chain from $E_1 \times E_2$ we must repeatedly double four points until we have the points of order eight above the kernel. As there are always four points to process, a vectorized implementation of doubling can always make use of four-way parallelism, and in most cases, multiplications can be fused into pairs to allow for eight-way parallelism.

In the following, we give explicit algorithms for the two-way vectorization of doubling for both: x-only points on the Kummer line of a Montgomery curve $(X : Z)$ (Algorithm 12) as well as modified Jacobian points on a Weierstrass curve $(X : Y : Z : T)$ (Algorithm 13). Given these algorithms, working with a pair of points can then use up to four-way parallelism and working with four points allows the use of up to eight-way parallelism.

The biggest performance benefit of this vectorization is seen in the repeated doubling of the modified Jacobian coordinates on the Weierstrass curve, but the x-only double is used to save a few cycles in the gluing codomain computation and is also useful when repeatedly doubling x-only points for SQIsign, as we will see later.

Classical approach ([DMPR24] and SQIsign). The matrix M is assembled from the symmetric elements g_T of the 4-torsion basis points; in its original formulation this

Algorithm 12 Two-way Doubling of x-only Points

Input: Coordinates of a point $P = (X : Z)$ and the Montgomery coefficient $A_{24} = (A+2)/4$
Output: Coordinates of the point $[2]P = (X_2 : Z_2)$

1:	$t_0 \leftarrow X + Z$	$t_1 \leftarrow X - Z$
2:	$t_0 \leftarrow t_0 \cdot t_0$	$t_1 \leftarrow t_1 \cdot t_1$
3:	$t_2 \leftarrow t_0 - t_1$	
4:	$X_2 \leftarrow t_0 \cdot t_1$	$t_0 \leftarrow t_2 \cdot A_{24}$
5:	$t_0 \leftarrow t_0 + t_1$	
6:	$Z_2 \leftarrow t_0 \cdot t_2$	

Cost: $2\mathbf{A}_1 + \mathbf{M}_1 + \mathbf{A}_2 + \mathbf{M}_2 + \mathbf{S}_2$

Algorithm 13 Two-way doubling of modified Jacobian points on Weierstrass curves

Input: Coordinates of a point $P = (X : Y : Z : T)$
Output: Coordinates of the point $[2]P = (X_2 : Y_2 : Z_2 : T_2)$

1:	$X^2 \leftarrow X \cdot X$	$t_0 \leftarrow Y \cdot Y$
2:	$A \leftarrow t_0 + t_0$	$t_1 \leftarrow X^2 + X^2$
3:	$t_2 \leftarrow X + A$	$t_3 \leftarrow t_1 + X^2$
4:	$t_2 \leftarrow t_2 \cdot t_2$	$A^2 \leftarrow A \cdot A$
5:	$t_2 \leftarrow t_2 - X^2$	$U \leftarrow A^2 + A^2$
6:	$M \leftarrow t_3 + T$	$S \leftarrow t_2 - A^2$
7:	$t_0 \leftarrow M \cdot M$	$t_1 \leftarrow Y \cdot Z$
8:	$Z_2 \leftarrow t_1 + t_1$	$t_2 \leftarrow S + S$
9:	$X_2 \leftarrow t_0 - t_2$	
10:	$t_0 \leftarrow S - X_2$	
11:	$t_0 \leftarrow M \cdot t_0$	$t_1 \leftarrow U \cdot T$
12:	$Y_2 \leftarrow t_0 - U$	$T_2 \leftarrow t_1 + t_1$

Cost: $2\mathbf{A}_1 + 6\mathbf{A}_2 + 2\mathbf{M}_2 + 2\mathbf{S}_2$

requires modular inversions, batched to a single inversion in SQIsign. The codomain null point is obtained by pushing isotropic 8-torsion through the duplication formula. A generic point is evaluated by applying the duplication formula *twice*: once at P, recovering three coordinates after dividing by the dual nulls, and once at the translate $P + T_1'$ by a 4-torsion point above the kernel, whose image is a coordinate permutation of $\Gamma(P)$; the missing coordinate is then recovered, at the cost of a projective realignment between the two evaluations. Our vectorized formulas cost $5\mathbf{M}_8 + 16\mathbf{M}_4 + 2\mathbf{M}_2 + \mathbf{S}_4$ and $11\mathbf{M}_8 + 2\mathbf{S}_8 + \mathbf{M}_2$ for codomain and evaluation, respectively.

Superglue [Dup25]. Two ideas remove both the inversions and the double evaluation. First, symmetric elements are computed *projectively* and coherently (sharing one scaling factor per curve), so the theta structure requires no inversion; moreover only six entries of M are ever needed. Second, writing the couple (P, Q) in barycentric coordinates $P_i \pm Q_i = (u_i : v_i : w_i)$, the left-hand side of the duplication formula becomes a homogeneous quartic in these coordinates whose coefficients are Hadamard transforms of column products $\mathcal{H}[M_i M_j]$. These are dual theta points of at most 4-torsion and are therefore *sparse* (typically two nonzero entries), with the sparsity pattern determined by the *position* of the canonical 2-torsion points $(C, 0), (0, C)$ in the symplectic basis, of which there are nine cases. A generic point is then evaluated by a *single* shifted duplication with $Q = T_1'$: since $\widetilde{\theta^J}(\Phi(T_1'))$ has no zero coordinate and only two distinct values, dividing it out is a cheap cofactor multiplication. The same barycentric representation extends to the quadratic twist at negligible cost, a capability absent in the classical formulation.

Since gluing occurs once per $(2^n, 2^n)$-chain, the overall speedup of Superglue is small,

but gluing was previously the most expensive step and the only one requiring inversion. In terms of performance, the original paper reports a speedup of roughly 2 times in the gluing step, essentially by cutting the costs from $175\mathbf{M}_1 + 24\mathbf{S}_1 + 1\mathbf{I}_1$ to $94\mathbf{M}_1 + 23\mathbf{S}_1$ in the gluing codomain computation; and $40\mathbf{M}_1 + 8\mathbf{S}_1$ to $26\mathbf{M}_1 + 2\mathbf{S}_1$ in the gluing evaluation. Our vectorized Superglue formulas for the codomain and evaluation cost $6\mathbf{M}_8 + \mathbf{S}_8 + \mathbf{M}_4 + \mathbf{M}_2$ and $7\mathbf{M}_8$, respectively; improving over our formulas for the previous gluing approach.

4.5 Vectorized Splitting

At the end of the isogeny chain between elliptic products, the codomain has been computed but given the work of Algorithm 10, only the level-two theta null point of the product structure is known. In SQIsign and many other cryptographic applications, an additional isomorphism must be computed to allow the recovery of the Montgomery models of both E_3 and E_4.[5]

In general, given the level two theta null point of a product structure, two steps are required to recover the Montgomery model. A product structure can be detected by checking whether exactly one of the ten even index level-(2,2) theta coordinates $U_{i,j}(0)$ is zero. When $U_{11,11}(0) = 0$, we are in the special case that for $\theta_B = (a : b : c : d)$ then the level two theta null coordinates of E_3 and E_4 can be read off as $(a : b)$ and $(a : c)$. Thus to recover the Montgomery models of the codomain, an isomorphism such that $U_{11,11}(0) = 0$ must be computed, then the Montgomery coefficient A itself can be recovered using the formula of [BRS25].

In [DMPR24], it was shown that ensuring $U_{11,11}(0) = 0$ could always be achieved by simply applying one of ten possible isomorphisms to the product structure. The transforms themselves are the action of a four by four matrix on the theta null point, and all ten transforms can be precomputed and then selected from at runtime using the index (i, j) such that $U_{i,j}(0) = 0$. As a result, splitting is relatively cheap and there are not many cycles to try and save through vectorization. However, there are still a few things we can do to save cycles wherever possible.

The main place vectorization helps is in computing the isomorphism itself. Computing the action of a four by four matrix on a point with four coordinates costs 16 field multiplications and 12 field additions. We can vectorize this to require $2\mathbf{M}_8$ and $3\mathbf{A}_4$. We can also improve this further if we compute the action on pairs of points, which requires $4\mathbf{M}_8$ and $3\mathbf{A}_8$, which is important when evaluating points through an isogeny chain between elliptic products.

When the splitting isomorphism is computed in constant time to avoid any leakage of torsion data, all ten symplectic transforms must be computed and then conditionally swapped. However, in SQIsign, not only does this not need to be done in verification, but the construction of the chain actually leads to a deterministic index $U_{i,j}(0) = 0$ for all $(2, 2)$-chains. As a result, the cost of the isomorphism is reduced to only 8 field additions and the splitting cost is essentially negligible.

4.6 Benchmarks

Table 6 presents the results in cycles for evaluating the different parts of the $(2^n, 2^n)$-isogeny chain in the theta model within SQIsign verification on our benchmarking machine. The baseline was obtained by microbenchmarking parts of the assembly implementation of Round-2 SQIsign. Operations include the two approaches to gluing, the generic (2,2)-isogeny steps forming the middle of the chain, and the splitting step at the end.

[5] In SQIsign we actually only ever need one of these models, but the splitting isomorphism nevertheless allows both models to be extracted.

Table 6: Cost of various operations in the theta model for the assembly baseline and AVX-512 implementation, with the performance improvement due to vectorization (×). The figures highlighted in bold are in millions of cycles.

	I			III			V		
	asm	AVX	×	asm	AVX	×	asm	AVX	×
GluingCodomain	39,634	7,781	5.09	84,960	14,922	5.69	119,985	19,712	6.09
SuperCodomain	-	5,122	7.74	-	10,349	8.21	-	12,781	9.39
GluingEval	17,409	6,232	2.79	41,440	10,294	4.03	58,965	14,281	4.13
SuperEval	-	4,290	4.06	-	8,381	4.94	-	10,955	5.38
Splitting	11,900	719	16.55	23,899	2,054	11.64	33,774	2,866	11.78
Middle	**2.33**	**0.78**	3.01	**8.47**	**2.18**	3.89	**16.83**	**4.39**	3.83
IsogenyChain	**2.98**	**0.96**	3.11	**10.29**	**2.65**	3.88	**20.14**	**5.18**	3.89

5 Application to SQIsign Verification

In this section, we show how to apply our vectorization algorithms for two-dimensional isogenies to the representative case of SQIsign verification. In addition, we extend the vectorization to other parts of the protocol to maximize performance. We dedicate the bulk of the analysis to the Round 2 version of SQIsign [AAA+24]. And at the end, we present results for the recently proposed Round 3 version of SQIsign [AAA+26] after adapting our algorithms and leveraging the efficiency of the code generator.

5.1 Round-2 SQIsign

SQIsign verification has a cost dominated by a $(2,2)$-isogeny chain between elliptic curve products. However, there are additional operations that can be targeted to maximize the use of vectorization. At a high level, the verification cost is dominated by the following three steps:

1. Compute the challenge curve. Given the public key E_{pk} together with a scalar chl and two $\mathtt{u8}$ integers n_{bt} and $\mathsf{hint}_{\mathsf{pk}}$, the challenge curve is computed as follows. First, $\mathsf{hint}_{\mathsf{pk}}$ is used to efficiently compute a deterministic basis $E_{\mathsf{pk}}[2^f] = \langle P_{\mathsf{pk}}, Q_{\mathsf{pk}} \rangle$. From this, the kernel of the $2^{f-n_{\mathsf{bt}}}$-isogeny $\phi : E_{\mathsf{pk}} \to E_{\mathsf{chl}}$ is computed as $\ker(\phi) = \langle [2^{n_{\mathsf{bt}}}](P_{\mathsf{pk}} + [\mathsf{chl}]Q_{\mathsf{pk}}) \rangle$ using x-only arithmetic, and the codomain is recovered using a chain of 4-isogenies (and a single 2-isogeny when $f - n_{\mathsf{bt}}$ is odd).

2. Reconstruct the dimension-two kernel generator. Given E_{aux} from the signature, together with the computed E_{chl} and two additional $\mathtt{u8}$ values, $\mathsf{hint}_{\mathsf{aux}}$ and $\mathsf{hint}_{\mathsf{chl}}$, two additional bases are computed: $E_{\mathsf{aux}}[2^f]$ and $E_{\mathsf{chl}}[2^f]$. Half the kernel generators for the dimension-two isogeny is recovered from repeated doublings: $P_1, Q_1 = [2^{f-e'_{\mathsf{rsp}}-2}]P_{\mathsf{aux}}, [2^{f-e'_{\mathsf{rsp}}-2}]Q_{\mathsf{aux}}$. The other two points must be obtained from a change of basis: $P_2 = [2^{f-r_{\mathsf{rsp}}-e'_{\mathsf{rsp}}-2}]([a_{00}]P_{\mathsf{chl}} + [a_{01}]Q_{\mathsf{chl}})$ and $Q_2 = [2^{f-r_{\mathsf{rsp}}-e'_{\mathsf{rsp}}-2}]([a_{10}]P_{\mathsf{chl}} + [a_{11}]Q_{\mathsf{chl}})$. Finally, when $r_{\mathsf{rsp}} > 0$, the points P_2, Q_2 are additionally pushed through a short $2^{r_{\mathsf{rsp}}}$-isogeny before continuing.

3. Evaluate the response isogeny. Given $\ker(\Phi) = \langle (P_1, P_2), (Q_1, Q_2) \rangle$, the isogeny $\Phi : E_{\mathsf{aux}} \times E_{\mathsf{chl}} \to F_1 \times F_2$ is computed using a chain of $(2,2)$-isogenies. Verification passes provided the kernel is well-formed and the codomain F_1, when hashed, produces a challenge matching the value chl used initially for recovering E_{chl}.

Algorithm 14 SQIsign Verification

Input: A message msg, public key $\mathsf{pk} = (E_{\mathsf{pk}}, \mathsf{hint}_{\mathsf{pk}})$ and a signature $\mathsf{sig} = (E_{\mathsf{aux}}, n_{\mathsf{bt}}, r_{\mathsf{rsp}}, M = \{a_{00}, a_{01}, a_{10}, a_{11}\}, \mathsf{chl}, \mathsf{hint}_{\mathsf{aux}}, \mathsf{hint}_{\mathsf{chl}})$
Output: A boolean determining whether the signature $\mathsf{sig}(m)$ is valid

$E_{\mathsf{pk}}[2^f] = \langle P_{\mathsf{pk}}, Q_{\mathsf{pk}} \rangle \leftarrow \mathsf{DeterministicTorsionBasis}(E_{\mathsf{pk}}, f, \mathsf{hint}_{\mathsf{pk}})$
$K_{\mathsf{pk}} \leftarrow [2^{n_{\mathsf{bt}}}](P_{\mathsf{pk}} + [\mathsf{chl}]Q_{\mathsf{pk}})$
$E_{\mathsf{chl}}, \mathtt{check} \leftarrow \mathsf{TwoIsogenyChain}(E_{\mathsf{pk}}, K_{\mathsf{pk}}, f - n_{\mathsf{bt}}, [\cdot])$ // Computes a $2^{f-n_{\mathsf{bt}}}$ isogeny $\phi : E_{\mathsf{pk}} \to E_{\mathsf{chl}}$ with $\ker \phi = \langle K_{\mathsf{pk}} \rangle$. The value f is a parameter of SQIsign where the characteristic $p = c \cdot 2^f - 1$ for odd c.
if $\mathtt{check} = 0$ **then**
  return `false` // `check` is zero if E_{pk} is not supersingular
end if
$E_{\mathsf{aux}}[2^f] = \langle P_{\mathsf{aux}}, Q_{\mathsf{aux}} \rangle \leftarrow \mathsf{DeterministicTorsionBasis}(E_{\mathsf{aux}}, f, \mathsf{hint}_{\mathsf{aux}})$
$E_{\mathsf{chl}}[2^f] = \langle P_{\mathsf{chl}}, Q_{\mathsf{chl}} \rangle \leftarrow \mathsf{DeterministicTorsionBasis}(E_{\mathsf{chl}}, f, \mathsf{hint}_{\mathsf{chl}})$
$e'_{\mathsf{rsp}} \leftarrow n - n_{\mathsf{bt}} - r_{\mathsf{rsp}}$ // n is a SQIsign parameter, which sets the response length
$(P_1, Q_1) \leftarrow [2^{f-e'_{\mathsf{rsp}}-2}]P_{\mathsf{aux}}, [2^{f-e'_{\mathsf{rsp}}-2}]Q_{\mathsf{aux}}$
$P_2 \leftarrow [2^{f-r_{\mathsf{rsp}}-e'_{\mathsf{rsp}}-2}]([a_{00}]P_{\mathsf{chl}} + [a_{01}]Q_{\mathsf{chl}})$
$Q_2 \leftarrow [2^{f-r_{\mathsf{rsp}}-e'_{\mathsf{rsp}}-2}]([a_{10}]P_{\mathsf{chl}} + [a_{11}]Q_{\mathsf{chl}})$
if $\mathsf{IsEven}(a_{00})$ **and** $\mathsf{IsEven}(a_{01})$ **then**
  $K_{\mathsf{rsp}} \leftarrow [2^{e'_{\mathsf{rsp}}+2}]Q_2$
else
  $K_{\mathsf{rsp}} \leftarrow [2^{e'_{\mathsf{rsp}}+2}]P_2$
end if
$E_{\mathsf{chl}}, (P_2, Q_2) \leftarrow \mathsf{TwoIsogenyChain}(E_{\mathsf{chl}}, K_{\mathsf{rsp}}, r_{\mathsf{rsp}}, [P_2, Q_2])$
$F_1, \mathtt{check} \leftarrow \mathsf{TwoTwoIsogenyChain}((E_{\mathsf{aux}}, E_{\mathsf{chl}}), ((P_1, Q_1), (P_2, Q_2)), e'_{\mathsf{rsp}})$
if $\mathtt{check} = 0$ **then**
  return `false` // `check` is zero if E_{aux} is not supersingular or the $(2^{e'_{\mathsf{rsp}}}, 2^{e'_{\mathsf{rsp}}})$-isogeny kernel is non-isotropic
end if
$\mathsf{chl}' \leftarrow \mathsf{Hash}(\mathsf{pk} || j(F_1) || \mathsf{msg})$
return $\mathsf{chl} = \mathsf{chl}'$

The computation of the challenge curve is functionally equivalent to algorithms required in SIKE [JAC+22], and was previously vectorized using AVX-512 in [CFGR22]. In particular, the kernel is recovered using a four-way implementation of the three point ladder which relies on a four-way xDBLADD and the isogeny chain itself computes the codomain and single point evaluations with two-way arithmetic. When multiple points are evaluated for optimized strategies, these can also be parallelized using up to eight-way arithmetic. Vectorization of $\Phi : E_{\mathsf{aux}} \times E_{\mathsf{chl}} \to F_1 \times F_2$ is realized as explained in Section 4, and so the only remaining vectorization design required for SQIsign verification is in the reconstruction of the kernel from the signature data.

Hashing. Although not a dominant cost in SQIsign verification, the hash remains another avenue for optimization. Computation of the scalar chl requires iterated hashing to make up for some security margin using SHAKE256. This can be vectorized by applying the work of [CL18], which implements the Keccak-f permutation using AVX-512 instructions. Since the 64, 256 and 512 calls to SHAKE256 for levels I, III and V (resp.) of Round-2 SQIsign are inherently sequential, the gain comes from vectorizing the permutation within each call, leading to a saving of roughly $24 \cdot 10^3$, $117 \cdot 10^3$ and $266 \cdot 10^3$ cycles (resp.), i.e., a 1.4× to 1.6× speedup of the challenge hash computation.

5.2 Vectorizing the Kernel Generator Reconstruction

After computing the challenge isogeny to recover E_{chl}, a SQIsign signature has all the data required to compute the $(2^{e'_{\mathsf{rsp}}}, 2^{e'_{\mathsf{rsp}}})$-isogeny $\Phi : E_{\mathsf{aux}} \times E_{\mathsf{chl}} \to F_1 \times F_2$, but the computation of $\ker(\Phi) = \langle (P_1, P_2), (Q_1, Q_2) \rangle$ requires some additional work before the chain itself can be computed.

Computing deterministic torsion bases. Due to the presence of the hints in the SQIsign signature, the cost of torsion basis generation is the cost of cofactor-clearing two points $x(P)$ and $x(Q)$, followed by a square-root that recovers a difference point $x(P-Q)$. As the cofactors are very small ($5, 65, 27$ for levels I, III, V, resp.), the entire cost is essentially determined by the square root. When generating bases in parallel, we compute a two-way square root and a four-way clearing of the cofactor.

Repeated basis doubling. As each basis is represented with a triple of points: $B = \{x(P), x(Q), x(P-Q)\}$, the repeated doubling required to recover the kernel means we need to repeatedly double three x-only points for each basis, or six points in total. As a single double can be performed with two-way vectorization (Algorithm 12), repeatedly doubling six points can be done with an eight-way doubling for four points and a four-way doubling for the remaining two.

Change of basis computation. The points $x(P_2), x(Q_2)$ and $x(P_2 - Q_2)$ are computed from four scalars a_{00}, a_{01}, a_{10} and a_{11}. The change of basis requires three calls to LadderBiscalar [AAA+24, Algorithm 8.8] to compute $x(P_2) = [a_{00}]P + [a_{01}]Q$, $x(Q_2) = [a_{10}]P + [a_{11}]Q$ and $x(P_2 - Q_2) = [a_{00} - a_{10}]P + [a_{01} - a_{11}]Q$ using x-only arithmetic.

Round-2 SQIsign uses a constant-time implementation of LadderBiscalar which, for n-bit scalars a, b, computes $R = [a]P + [b]Q$ using $2n$ xDBL and $4n$ calls to xADD. As both xDBL and xADD have two-way vectorization, a naive implementation of this method would consist of a four-way vectorization for two of the LadderBiscalar calls by working with a pair of inputs, and a two-way vectorization for the final call. We propose a more efficient approach in Section 5.3.

Computation of the short isogeny. Here we apply the same vectorization approach used in the computation of the challenge curve, with the difference that in this case the isogeny is significantly shorter. The kernel is computed by repeatedly doubling a single point, which benefits from a two-way vectorization. The isogeny itself is the same as the previous chain, with the exception that now we are required to evaluate the basis points $x(P_2), x(Q_2), x(P_2 - Q_2)$ as well as the kernel generator itself.

5.3 Algorithmic improvements to the change of basis computation

The following section details two algorithmic improvements for the change of basis computation which apply to both the scalar and vectorized verification. Neither approach dominates in our vectorized implementation: the second one is faster when $r_{\mathsf{rsp}} = 0$, while the first one is faster otherwise.

5.3.1 Using variable-time algorithms for the biscalar ladder

The first modification is simple: since verification does not need to be constant time, we can use the variable-time algorithm of [CS17, Algorithm 9] to compute LadderBiscalar with $\sim 2n$ xDBLADD calls. A single xDBLADD can be computed using four-way arithmetic (Algorithm 15); thus, a call to LadderBiscalar uses four-way vectorization for a single input or eight-way vectorization for a pair of inputs. Moreover, as LadderBiscalar is variable time,

it is possible that one of the loops for a pair of inputs terminates before the other. In this case, we terminate LadderBiscalarPair and begin a second call to LadderBiscalar (with a much smaller bitlength) to finish the computation with four-way arithmetic.

Algorithm 15 Vectorized xDBLADD

Input: Coordinates of x-only point $x(P) = (X_P : Z_P)$, $x(Q) = (X_Q : Z_Q)$ and $x(P-Q) = (X_\ominus : Z_\ominus)$ and the Montgomery coefficient A.
Output: Coordinates of the points $x([2]P) = (X_2 : Z_2)$ and $x(P+Q) = (X_\oplus : Z_\oplus)$

1:	$t_1 \leftarrow X_P + Z_P$	$t_2 \leftarrow X_P - Z_P$	$t_3 \leftarrow X_Q + Z_Q$	$t_4 \leftarrow X_Q - Z_Q$
2:	$s_1 \leftarrow t_1 \cdot t_1$	$s_2 \leftarrow t_2 \cdot t_2$	$s_3 \leftarrow t_3 \cdot t_2$	$s_4 \leftarrow t_4 \cdot t_1$
3:	$t_1 \leftarrow s_1 + s_2$	$t_2 \leftarrow s_1 - s_2$	$t_3 \leftarrow s_3 + s_4$	$t_4 \leftarrow s_3 - s_4$
4:	$s_1 \leftarrow t_1^2$	$s_2 \leftarrow t_2^2$	$s_3 \leftarrow t_3^2$	$s_4 \leftarrow t_4^2$
5:	$t_1 \leftarrow t_1 \cdot t_2$	$t_2 \leftarrow s_2 \cdot A$	$t_3 \leftarrow s_3 \cdot Z_\ominus$	$t_4 \leftarrow s_4 \cdot X_\ominus$
6:	$X_2 \leftarrow s_1 - s_2$	$Z_2 \leftarrow t_1 + t_2$	$X_\oplus \leftarrow t_3$	$Z_\oplus \leftarrow t_4$
7:	$Z_2 \leftarrow Z_2 + t_1$			

Cost: $2\mathbf{M}_4 + \mathbf{S}_4 + 4\mathbf{A}_4$

5.3.2 Using x-only Montgomery ladders with Jacobian Points

The second modification requires changing the computation order within SQIsign verification (Algorithm 14). The core idea is that if we move the 'x-only to Jacobian coordinates' lifting from inside the dimension-two chain to before the change of basis, we can trade most of the LadderBiscalar calls for cheaper Montgomery ladders for scalar multiplication. The proposed procedure is as follows.

1. Compute the two torsion bases and repeatedly double them to recover $P_1, Q_1 \in E_{\mathsf{aux}}$ and a basis P, Q of E_{chl} of the correct order.
2. Before computing the change of basis, compute the short $2^{r_{\mathsf{rsp}}}$-isogeny and evaluate it on P and Q.
3. The images of P and Q through this isogeny are directly lifted to full Jacobian points $(X : Y : Z)$ before the change of basis. Then, four scalar multiplications, $[a_{00}]P$, $[a_{10}]P$, $[a_{01}]Q$ and $[a_{11}]Q$, are computed using x-only arithmetic with the Okeya-Sakurai formula [CS17, Algorithm 5] to recover the y-coordinate without requiring a square root.
4. The basis points are then recovered with two additions: $P_2 = [a_{00}]P + [a_{01}]Q$ and $Q_2 = [a_{10}]P + [a_{11}]Q$.
5. The lifted points are passed as input into the $(2,2)$-isogeny chain, which no longer needs to perform the lifting to the Jacobian points.

The proposed approach adds the cost of one LadderBiscalar computation when $r_{\mathsf{rsp}} > 0$, but replaces three calls to LadderBiscalar with the full scalars by four Montgomery ladders. As each Montgomery ladder requires n calls to xDBLADD, pairs of points can be multiplied using eight-way arithmetic. The most optimistic cost estimate of our method is $\frac{4}{6}$ of the original cost. However, note that the actual cost of LadderBiscalar is more complex as the main loop terminates once $\gcd(a, b)$ is computed, for which we only have an upper bound for the number of required steps.

5.3.3 Comparison of the Two Methods

Table 7 compares the two approaches over the KAT signatures of Round-2 SQIsign, excluding the lifting to Jacobian coordinates and the short $2^{r_{\mathsf{rsp}}}$-isogeny, which are common

to both. For signatures with $r_{\mathsf{rsp}} = 0$, the second approach is 2–13% faster, as it replaces three biscalar ladders by four Montgomery ladders. When $r_{\mathsf{rsp}} > 0$, the additional biscalar ladder for the kernel generator makes it 21–27% slower. Over the KAT signatures, the first approach is 2–11% faster on average, which amounts to at most 1.5% of the verification cost.

Table 7: Cost of the change of basis (in thousands of cycles, averaged over the KAT signatures) using LadderBiscalar (x-only) and Montgomery ladders on Jacobian points (Jacobian), depending on the length r_{rsp} of the short isogeny.

	I		III		V	
	x-only	Jacobian	x-only	Jacobian	x-only	Jacobian
$r_{\mathsf{rsp}} = 0$	270	234	770	753	1,348	1,254
$r_{\mathsf{rsp}} > 0$	270	328	765	970	1,346	1,676
All	270	274	767	861	1,347	1,491

5.4 Benchmarks

Table 8 presents the timing results in clock cycles for verifying Round-2 SQIsign signatures at the three NIST security levels on our benchmarking machine, a 16-core Intel Xeon Silver 4314 (Ice Lake-SP) server running at 2.4GHz. Compared to the optimized baseline implementation using x86-64 hand-written assembly, our vectorized implementations achieve speedups that range from 2.5× to 3.3×. The gap between level I and the higher levels is largely explained by the performance of the vectorized field arithmetic.

As can be seen in Table 8, our AVX-512 implementation is up to 1.83× faster than the AVX-512 implementation by Wang et al. [WWW+26], which demonstrates the effectiveness of the proposed vectorization. Similarly, our performance improvements compare favorably with the roughly 1.5× speedup from work vectorizing SQIsign on Armv8 [FJWY26]. In this case, some of the performance gap can be explained by the wider SIMD registers and larger multipliers found in AVX-512 in comparison to NEON.

Table 8: Cost of Round-2 SQIsign verification (in millions of cycles) for the assembly baseline and the proposed AVX-512 implementation.

	I		III		V	
	Cycles ($\times 10^6$)	Speedup (×)	Cycles ($\times 10^6$)	Speedup (×)	Cycles ($\times 10^6$)	Speedup (×)
asm	5.8	-	20.2	-	39.0	-
[WWW+26], AVX-512	3.9	1.5	11.5	1.8	21.6	1.8
This work, AVX-512	**2.3**	**2.5**	**6.4**	**3.2**	**11.8**	**3.3**

5.5 Round-3 SQIsign

Recently, SQIsign was selected to advance to Round 3 of the NIST standardization process for additional post-quantum digital signatures [AAA+26]. The Round 3 submission made several adjustments to the protocol, which we summarize below.

1. Due to Wesolowski's new attack on the supersingular isogeny problem [Wes26], SQIsign updated all three of its parameter sets, which now use larger primes.

2. Signing was modified to ensure that $n_{\mathsf{bt}} = 0$, which simplifies verification by removing the second short 2-isogeny chain and fixes the length of the $(2,2)$-isogeny chain.

3. The gluing isogeny algorithm was refactored to allow the entire operation to be performed with x-only arithmetic, avoiding the requirement of projective points for computing additions.

By virtue of our code generator for vectorized arithmetic, we are able to produce efficient field arithmetic for the updated primes with no additional work. We also updated the parameter constants and removed the backtracking isogeny, which only simplifies the existing implementation. With these straightforward changes, our implementation becomes compatible with Round-3 SQIsign and passes all the known-answer tests from the Round 3 submission.

With a little more work, we additionally include gluing following the Round 3 specification. This results in removing the lift from x-only points to projective points, saving some inversions and square roots. It also means swapping the vectorized doubling of Jacobian points on Weierstrass curves for the GluingLadder [AAA+26, Algorithm 4.90], which is simply a chain of xDBLADD. As before, xDBLADD is implemented with four-way vectorization for a single point, or eight-way for pairs of points.

We present timings for the vectorized verification of Round-3 SQIsign in Table 9. The assembly baseline is the Round 3 submission built with its optimized broadwell backend and default Release configuration. For comparison, the table also includes the concurrent AVX-512 implementation by Wen et al. [WLH+26], measured on an Intel Core i7-11800H, where the same baseline costs agree with ours to within 1%.

Table 9: Cost of Round-3 SQIsign verification (in millions of cycles) for the assembly baseline and the proposed AVX-512 implementation.

	I		III		V	
	Cycles ($\times 10^6$)	Speedup ($\times$)	Cycles ($\times 10^6$)	Speedup ($\times$)	Cycles ($\times 10^6$)	Speedup ($\times$)
asm	13.1	-	34.7	-	89.7	-
[WLH+26], AVX-512	6.1	2.1	16.7	2.1	38.4	2.3
This work, AVX-512	**4.0**	**3.3**	**10.1**	**3.4**	**28.8**	**3.1**

References

[AAA+24] Marius A. Aardal, Gora Adj, Diego F. Aranha, Andrea Basso, Isaac Andrés Canales Martínez, Jorge Chávez-Saab, Maria Corte-Real Santos, Pierrick Dartois, Luca De Feo, Max Duparc, Jonathan Komada Eriksen, Tako Boris Fouotsa, Décio Luiz Gazzoni Filho, Basil Hess, David Kohel, Antonin Leroux, Patrick Longa, Luciano Maino, Michael Meyer, Kohei Nakagawa, Hiroshi Onuki, Lorenz Panny, Sikhar Patranabis, Christophe Petit, Giacomo Pope, Krijn Reijnders, Damien Robert, Francisco Rodríguez Henríquez, Sina Schaeffler, and Benjamin Wesolowski. SQIsign. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[AAA+25] Marius A. Aardal, Gora Adj, Arwa Alblooshi, Diego F. Aranha, Isaac Andrés Canales Martinez, Jorge Chávez-Saab, Décio Luiz Gazzoni Filho, Krijn Reijnders, and Francisco Rodríguez-Henríquez. Optimized one-dimensional SQIsign verification on intel and cortex-M4. *IACR TCHES*, 2025(1):497–522, 2025. doi:10.46586/tches.v2025.i1.497-522.

[AAA+26] Marius A. Aardal, Gora Adj, Diego F. Aranha, Andrea Basso, Giacomo Borin, Isaac Andrés Canales Martínez, Jorge Chávez-Saab, Maria Corte-Real Santos,

Pierrick Dartois, Luca De Feo, Max Duparc, Thomas Espitau, Jonathan Komada Eriksen, Tako Boris Fouotsa, Décio Luiz Gazzoni Filho, Basil Hess, Riccardo Invernizzi, David Kohel, Antonin Leroux, Patrick Longa, Luciano Maino, Michael Meyer, Marzio Mula, Kohei Nakagawa, Hiroshi Onuki, Lorenz Panny, Sikhar Patranabis, Christophe Petit, Giacomo Pope, Krijn Reijnders, Damien Robert, Francisco Rodríguez Henríquez, Sina Schaeffler, Frederik Vercauteren, Alexandre Wallet, Benjamin Wesolowski, and Wessel van Woerden. SQIsign. Technical report, National Institute of Standards and Technology, 2026. Available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-3-additional-signatures.

[BBC+25] Andrea Basso, Giacomo Borin, Wouter Castryck, Maria Corte-Real Santos, Riccardo Invernizzi, Antonin Leroux, Luciano Maino, Frederik Vercauteren, and Benjamin Wesolowski. PRISM: Simple and compact identification and signatures from large prime degree isogenies. In Tibor Jager and Jiaxin Pan, editors, *PKC 2025, Part III*, volume 15676 of *LNCS*, pages 300–332. Springer, Cham, May 2025. doi:10.1007/978-3-031-91826-1_10.

[BBRS26] Andrea Basso, Giacomo Borin, Ryan Rueger, and Sina Schaeffler. CORAL faster isogeny group action for post-quantum NIKE. *Cryptology ePrint Archive*, 2026.

[BCE+25] Giacomo Borin, Maria Corte-Real Santos, Jonathan Komada Eriksen, Riccardo Invernizzi, Marzio Mula, Sina Schaeffler, and Frederik Vercauteren. Qlapoti: Simple and efficient translation of quaternion ideals to isogenies. In Goichiro Hanaoka and Bo-Yin Yang, editors, *ASIACRYPT 2025, Part IV*, volume 16248 of *LNCS*, pages 174–205. Springer, Singapore, December 2025. doi:10.1007/978-981-95-5113-2_6.

[BDD+24] Andrea Basso, Pierrick Dartois, Luca De Feo, Antonin Leroux, Luciano Maino, Giacomo Pope, Damien Robert, and Benjamin Wesolowski. SQIsign2D-West - the fast, the small, and the safer. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part III*, volume 15486 of *LNCS*, pages 339–370. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0891-1_11.

[BFLS20] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. Faster computation of isogenies of large prime degree. *CoRR*, abs/2003.10118, 2020.

[BM25] Andrea Basso and Luciano Maino. POKÉ: A compact and efficient PKE from higher-dimensional isogenies. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part II*, volume 15602 of *LNCS*, pages 94–123. Springer, Cham, May 2025. doi:10.1007/978-3-031-91124-8_4.

[BMP23] Andrea Basso, Luciano Maino, and Giacomo Pope. FESTA: Fast encryption from supersingular torsion attacks. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VII*, volume 14444 of *LNCS*, pages 98–126. Springer, Singapore, December 2023. doi:10.1007/978-981-99-8739-9_4.

[BRS25] Razvan Barbulescu, Damien Robert, and Nicolas Sarkis. Models of kummer lines and galois representations. Cryptology ePrint Archive, Report 2025/543, 2025. URL: https://eprint.iacr.org/2025/543.

[CD23] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 423–447. Springer, Cham, April 2023. doi:10.1007/978-3-031-30589-4_15.

[CFGP25] Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Daniel Page. Fast AVX-512 implementation of the optimal ate pairing on BLS12-381. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(4):848–872, 2025.

[CFGR22] Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Peter Y. A. Ryan. Highly vectorized SIKE for AVX-512. *IACR TCHES*, 2022(2):41–68, 2022. doi:10.46586/tches.v2022.i2.41-68.

[CL18] Roberto Cabral and Julio López. Implementation of the SHA-3 family using AVX512 instructions. *Anais do XVIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg 2018)*, 2018. URL: https://api.semanticscholar.org/CorpusID:195491940.

[CLP24] Mingjie Chen, Antonin Leroux, and Lorenz Panny. SCALLOP-HD: group action from 2-dimensional isogenies. In Qiang Tang and Vanessa Teague, editors, *Public-Key Cryptography - PKC 2024*, volume 14603 of *Lecture Notes in Computer Science*, pages 190–216. Springer, 2024.

[Cor25a] Intel Corporation. The converged vector ISA: Intel advanced vector extensions 10. Technical Paper, Revision 3.0, 2025. URL: https://www.intel.com/content/www/us/en/content-details/849709/the-converged-vector-isa-intel-advanced-vector-extensions-10-technical-paper.html.

[Cor25b] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2025.

[CS17] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic: The case of large characteristic fields. Cryptology ePrint Archive, Report 2017/212, 2017. URL: https://eprint.iacr.org/2017/212.

[CSD+23] Jorge Chavez-Saab, Maria Corte-Real Santos, Luca De Feo, Jonathan Komada Eriksen, Basil Hess, David Kohel, Antonin Leroux, Patrick Longa, Michael Meyer, Lorenz Panny, Sikhar Patranabis, Christophe Petit, Francisco Rodríguez Henríquez, Sina Schaeffler, and Benjamin Wesolowski. SQIsign. Technical report, National Institute of Standards and Technology, 2023. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

[DEF+25] Pierrick Dartois, Jonathan Komada Eriksen, Tako Boris Fouotsa, Arthur Herlédan Le Merdy, Riccardo Invernizzi, Damien Robert, Ryan Rueger, Frederik Vercauteren, and Benjamin Wesolowski. PEGASIS: Practical effective class group action using 4-dimensional isogenies. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part I*, volume 16000 of *LNCS*, pages 67–99. Springer, Cham, August 2025. doi:10.1007/978-3-032-01855-7_3.

[DEIV26] Pierrick Dartois, Jonathan Komada Eriksen, Riccardo Invernizzi, and Frederik Vercauteren. sfqt-sfPegasis: Simpler and faster effective class group actions. In Joan Daemen and Emmanuel Thomé, editors, *EUROCRYPT 2026, Part IV*, volume 16545 of *LNCS*, pages 636–665. Springer, Cham, May 2026. doi:10.1007/978-3-032-25327-9_22.

[DF24] Max Duparc and Tako Boris Fouotsa. SQIPrime: A dimension 2 variant of SQISignHD with non-smooth challenge isogenies. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part III*, volume 15486 of *LNCS*, pages 396–429. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0891-1_13.

[DKL+20] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 64–93. Springer, Cham, December 2020. doi:10.1007/978-3-030-64837-4_3.

[DLLW23] Luca De Feo, Antonin Leroux, Patrick Longa, and Benjamin Wesolowski. New algorithms for the Deuring correspondence - towards practical and secure SQISign signatures. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 659–690. Springer, Cham, April 2023. doi:10.1007/978-3-031-30589-4_23.

[DLRW24] Pierrick Dartois, Antonin Leroux, Damien Robert, and Benjamin Wesolowski. SQIsignHD: New dimensions in cryptography. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part I*, volume 14651 of *LNCS*, pages 3–32. Springer, Cham, May 2024. doi:10.1007/978-3-031-58716-0_1.

[DMPR24] Pierrick Dartois, Luciano Maino, Giacomo Pope, and Damien Robert. An algorithmic approach to (2, 2)-isogenies in the theta model and applications to isogeny-based cryptography. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part III*, volume 15486 of *LNCS*, pages 304–338. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0891-1_10.

[Dup25] Max Duparc. Superglue: Fast formulae for (2,2)-gluing isogenies. In Goichiro Hanaoka and Bo-Yin Yang, editors, *ASIACRYPT 2025, Part IV*, volume 16248 of *LNCS*, pages 372–400. Springer, Singapore, December 2025. doi:10.1007/978-981-95-5113-2_12.

[FJWY26] Luca De Feo, Li-Jie Jian, Ting-Yuan Wang, and Bo-Yin Yang. SQISign on ARM. Cryptology ePrint Archive, Paper 2026/394, 2026. URL: https://eprint.iacr.org/2026/394.

[JAC+22] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[Kan97] Ernst Kani. The number of curves of genus two with elliptic differentials. *Journal fur die reine und angewandte Mathematik*, 485:93–122, 1997.

[KG19] Dusan Kostic and Shay Gueron. Using the new VPMADD instructions for the new post quantum key encapsulation mechanism SIKE. In *Symposium on Computer Arithmetic, ARITH 2019*, pages 215–218. IEEE, 2019.

[KLPT14] David Kohel, Kristin Lauter, Christophe Petit, and Jean-Pierre Tignol. On the quaternion l-isogeny path problem. *LMS Journal of Computation and Mathematics*, 17, 06 2014.

[McL21] Michael B. McLoughlin. addchain: Cryptographic addition chain generation in go. Repository https://github.com/mmcloughlin/addchain, October 2021. doi:10.5281/zenodo.5622943.

[MMP+23] Luciano Maino, Chloe Martindale, Lorenz Panny, Giacomo Pope, and Benjamin Wesolowski. A direct key recovery attack on SIDH. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 448–471. Springer, Cham, April 2023. doi:10.1007/978-3-031-30589-4_16.

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

[Mor23] Tomoki Moriya. IS-CUBE: an isogeny-based compact KEM using a boxed SIDH diagram. 2023. URL: https://eprint.iacr.org/2023/1506.

[Nat26] National Institute of Standards and Technology (NIST). Post-quantum cryptography: Additional digital signature schemes, 2022–2026. Call for proposals: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf. URL: https://csrc.nist.gov/Projects/pqc-dig-sig/standardization.

[NO24] Kohei Nakagawa and Hiroshi Onuki. QFESTA: Efficient algorithms and parameters for FESTA using quaternion algebras. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part V*, volume 14924 of *LNCS*, pages 75–106. Springer, Cham, August 2024. doi:10.1007/978-3-031-68388-6_4.

[NOC+24] Kohei Nakagawa, Hiroshi Onuki, Wouter Castryck, Mingjie Chen, Riccardo Invernizzi, Gioella Lorenzon, and Frederik Vercauteren. SQIsign2D-East: A new signature scheme using 2-dimensional isogenies. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part III*, volume 15486 of *LNCS*, pages 272–303. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0891-1_9.

[OLHA18] Gabriell Orisaka, Julio César López-Hernández, and Diego F. Aranha. Finite field arithmetic using AVX-512 for isogeny-based cryptography. In *18th Brazilian Symposium on Information and Computer Systems Security (SBSeg)*, pages 49–56, 2018.

[Por20] Thomas Pornin. Optimized binary GCD for modular inversion. Cryptology ePrint Archive, Report 2020/972, 2020. URL: https://eprint.iacr.org/2020/972.

[Rob23] Damien Robert. Breaking SIDH in polynomial time. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 472–503. Springer, Cham, April 2023. doi:10.1007/978-3-031-30589-4_17.

[Rob24] Damien Robert. The module action for isogeny based cryptography. Cryptology ePrint Archive, Report 2024/1556, 2024. URL: https://eprint.iacr.org/2024/1556.

[Wes26] Benjamin Wesolowski. The supersingular isogeny problem in time and memory $p^{1/3+o(1)}$. Cryptology ePrint Archive, Paper 2026/1486, 2026. URL: https://eprint.iacr.org/2026/1486.

[WLH+26] Yunqi Wen, Xuzhe Liu, Zhi Hu, Saiyu Wang, and Chang-An Zhao. AVX-512 acceleration of SQIsign 3.0 verification. Cryptology ePrint Archive, Paper 2026/2505, 2026. URL: https://eprint.iacr.org/2026/2505.

[WWW+26] Weize Wang, Chutong Wang, Yu Wu, Qifan Xue, Jieyu Zheng, and Yunlei Zhao. Exposing SIMD parallelism in sqisign: An AVX-512 implementation. Cryptology ePrint Archive, Paper 2026/1713, 2026. URL: https://eprint.iacr.org/2026/1713.