On the Parallelization of Square-Root Vélu's Formulas

Jorge Chávez-Saab *, Odalis Ortega *, Amalia Pizarro-Madariaga *

Posted Date: 18 January 2024

doi: 10.20944/preprints202401.1366.v1

Keywords: isogenies; elliptic curves; parallelism; postquantum cryptography; efficient implementation

Article

On the Parallelization of Square-Root Vélu's Formulas

Jorge Chávez-Saab[1,†], Odalis Ortega [2,†], Amalia Pizarro-Madariaga [2,†]

[1] Cryptography Research Center, Technology Innovation Institute, Abu Dhabi, United Arab Emirates; jorge.saab@tii.ae

[2] Instituto de Matemáticas, Universidad de Valparaíso, Chile; odalis.ortega@postgrado.uv.cl, amalia.pizarro@uv.cl

† These authors contributed equally to this work and order is alphabetical.

Abstract: A primary challenge in isogeny-based cryptography lies in the substantial computational cost associated to computing and evaluating prime-degree isogenies. This computation traditionally relied on Vélu's formulas, an approach with time complexity linear in the degree but which was further enhanced by Bernstein, De Feo, Leroux, and Smith to a square-root complexity. The improved square-root Vélu's formulas exhibit a degree of parallelizability which has not been exploited in major implementations. In this study, we introduce a theoretical framework for parallelizing isogeny computations and provide a proof-of-concept implementation in C with OpenMP. While the parallelization effectiveness exhibits diminishing returns with the number of cores, we still obtain strong results when using a small number of cores. Concretely, our implementation shows that for large degrees it is easy to achieve speedup factors of up to 1.74, 2.54 and 3.44 for 2, 4 and 8 cores, respectively.

Keywords: isogenies; elliptic curves; parallelism; postquantum cryptography; efficient implementation

1. Introduction

Since the foundational work of De Feo, Jao and Plût [1,2], which resulted in the SIKE protocol [3], isogeny-based cryptography has received considerable interest in the development of post-quantum cryptography. Today, the security of SIKE has been broken by a wave of attacks which started with the work of Castryck and Decru [4] and was followed up by the works of Maino, Martindale, Panny, Pope, and Wesolowski [5] and of Robert [6], ultimately demonstrating the existence of a polynomial-time attack on SIKE with any starting curve. Nevertheless, there are still many isogeny-based protocols that remain unbroken, including the CSIDH [7] key exchange protocol and signature schemes like SeaSign [8], CSI-Fish [9] and SQISign [10], the last of which is currently under consideration in the NIST (National Institute of Standards and Technology) process for Standardization of Additional Digital Signature Schemes [11].

While SIKE only required the computation of isogenies of degrees 2 and 3, there has been a tendency in some of the newer isogeny-based protocols to move towards higher and higher prime degrees. This has brought increasing attention to the task of optimizing the evaluation of large-degree isogenies. For instance, the original CSIDH proposal used isogenies of degrees up to 587, and Chávez, Chi, Jacques and Rodríguez have suggested that a level 5 dummy-free implementation would need degrees as high as 2239 [12] after revising the security. More recently, the level 5 version of SQISign that was submitted to the NIST standarization process [13] brings the ceiling up with a colossal degree-318233 isogeny computation. Even the aforementioned attacks on SIKE require the evaluation of large prime-degree isogenies, with the techniques from [5] requiring the evaluation of an isogeny of degree 321193 over a large extension field for a direct key recovery attack. Therefore, optimizing the computation of large prime-degree isogenies is of great interest for constructive purposes as well as for cryptanalysis. Interestingly, a new generation has also risen of isogeny-based protocols which utilize the techniques of the attack constructively, such as FESTA [14], QFESTA [15], IS-CUBE [16] and constructions for VDFs [17] and VRFs [18], of which the last two also involve large-degree isogenies.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

Traditionally, the main method for computing isogenies of prime degree ℓ when there exists a point of order ℓ in the curve has been Vélu's formulas, initially introduced in [19] and expanded upon in [20, §2.4] and [21, Theorem 12.16]. In view of the ubiquitousness of large-degree isogenies, the more recent improvement from Bernstein, De Feo, Leroux, and Smith [22], which reduced the time complexity from $\mathcal{O}(\ell)$ to $\tilde{\mathcal{O}}(\sqrt{\ell})$, has been a keystone for the viability of many protocols. In addition to the straightforward savings that it provides, this new algorithm also exhibits a substantial degree of parallelizability that has remained largely unexplored.

Our Contributions. In this work we detail and showcase the parallelizability of the square-root Vélu formulas, focusing on the problem of computing a single large-degree isogeny from a fixed kernel generator and pushing a number of points through said isogeny. We analyze and provide a parallelism-friendly reformulation of the square-root Vélu formulas, and provide a proof-of-concept (PoC) implementation to illustrate the implications of our work. More precisely,

- We propose a new indexing system for the points in the kernel of the isogeny that is similar to the one presented in [22], but which can be naturally split into subsets that are assigned to each core in a multi-core implementation.
- We provide a parallelized expected cost function based on our square-root Vélu variant, as well as a revised expected cost of the sequential Vélu formulas that were presented in [23]. Assuming that the assigned task is to perform an isogeny construction and to push two points through it, the idealized speedup factors for large degrees is up to 1.79, 2.96 and 4.58 faster than the expected sequential cost function using two, four and eight cores, respectively.
- We give a PoC implementation utilising C and OpenMP to compute our isogeny formulas with large odd prime degrees. Our implementation assumes for concreteness that isogenies are evaluated over a prime field such that there exist rational points of the desired degree in the curve, and achieves speedup factors for large degrees of up to 1.74, 2.54 and 3.44 for 2, 4 and 8 cores, respectively, even for medium-sized isogenies.

Assumptions and generalizations. For the purposes of our implementation, we have searched for a 1792-bit prime such that $p + 1$ contains evenly-spaced prime factors ranging from 19 to 321193 (the degree of the isogeny in the attack of [5]). Isogenies are performed over the base prime field, and two points are pushed through each isogeny. These choices were made for concreteness and closely mirror the scenario in CSIDH. On the other hand, they do not adhere exactly to the case of SQISign (which works in a quadratic field extension) nor that of the attack in [5] (which performs isogenies in a much larger field extension). Nevertheless, the techniques that we describe save on a fixed number of field multiplications, which are the dominating part of the total cost. Therefore we expect that similar speedups in percentage would be obtained in those other cases, with the caveat that the savings could even be slightly larger as the parallelization overhead becomes more negligible with respect to the arithmetic cost. As for the theoretical savings, our costs are expressed in terms of the total number of field multiplications and remain valid for any choice of base field.

Related Work. Different forms of parallelism have been explored at various layers of isogeny-based protocols. For instance, the use of vectorization through Intel's Advanced Vector Extensions (AVX-512) has been exploited for the finite field arithmetic layer, both in the context of CSIDH [24] and of the now obsolete SIKE protocol [25,26], obtaining speedup factors in the order of 1.5. Additionally, it has also been proposed to use AVX-512 to batch multiple evaluations of the protocols, leading to increases in throughput of up to 3.6 for CSIDH [24] and 4.6 in SIKE [26]. When latency is the focus, vectorization can also be used to push multiple points through an isogeny, leading to the concept of parallel-friendly evaluation strategies which favour pushing points over point multiplications [26–28]. These parallel strategies can accelerate the evaluation of large composite-degree isogenies, but do not apply for isogenies of a large prime degree.

This work focuses exclusively on the multi-core optimization for the isolated evaluation of a large prime-degree isogeny, which has received much less attention. To the best of our knowledge,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

there are only simple parallelization strategies that have been proposed [29] for the linear-complexity Vélu formulas, not exploiting the improved complexity methods from [22]. We also do not consider vectorization: if it was to be used, we expect that the best way to exploit it would be at the field arithmetic layer. This would lead to improvements similar to those from [24–26] which are well documented and are largely parallel to our multi-core improvements.

Outline. The remainder of the manuscript is organized as follows. In §2 we give the basic background and description of the original square-root Vélu formulas with their three main building blocks: `KPS`, `xISOG` and `xEVAL`, and derive their expected cost function. In §3 we introduce our new framework, explaining the need for a new indexing system in §3.1 and then detailing the new algorithms in §3.2 and deriving their expected cost function in §3.3. Finally, in §4, we present our experimental results, comparing the performance to the theoretical savings.

2. Background

We denote by $\mathbb{F}_q$ the finite field with $q = p^n$ elements, and p a prime number. We focus on supersingular Montgomery curves, E, given by

$$E\colon y^2 = x^3 + Ax^2 + x, \quad A \in \mathbb{F}_q \setminus \{\pm 2\}.$$

Remark 1. *In general, a Montgomery curve is defined by the equation $E_{A,B} : By^2 = x^3 + Ax^2 + x$ such that $B \neq 0$ and $A \neq \pm 2$, but all choices of B are equivalent up to isomorphism. For the sake of simplicity, we write E_A instead of $E_{A,1}$ and omit the constant B.*

Viewing the curve as a projective surface, the set $E(\mathbb{F}_q)$ of $\mathbb{F}_q$-rational points in E forms a group where the point at infinity ∞ acts as the group identity. An order-d point P on E is a point on the curve such that d is the smallest positive integer satisfying

$$[d]P := \sum_{i=1}^{d} P = \infty.$$

We write $E[d]$ to refer to the d-torsion subgroup $\{P \in E(\overline{\mathbb{F}_q}) \mid [d]P = \infty\}$ of E.

Isogenies. An isogeny $\phi\colon E \to E'$ is a surjective morphism between curves with a finite kernel such that $\phi(\infty_E) = \infty_{E'}$. Such a map is always also a group homomorphism. Two curves E and E' are isogenous over $\mathbb{F}_q$ if there exists such ϕ connecting them, or equivalently if $\#E(\mathbb{F}_q) = \#E'(\mathbb{F}_q)$. The kernel of ϕ is $\{P \in E(\mathbb{F}_q) \mid \phi(P) = \infty\}$, denoted by $\ker\phi$. When restricting only to separable maps, the degree of an isogeny is equal to the size of its kernel, and an isogeny is uniquely determined by its kernel up to an isomorphism. An isogeny $\phi : E \to E'$ of degree ℓ is referred to as a ℓ-isogeny, and it has a unique dual (up to isomorphism) $\hat{\phi} : E' \to E$ of the same degree such that $\phi \circ \hat{\phi}$ and $\hat{\phi} \circ \phi$ are equivalent to the multiplication-by-ℓ map on E and E', respectively.

2.1. Computation of ℓ-Isogenies

Let E_A be a Montgomery curve defined over $\mathbb{F}_q$ and $P \in E_A(\mathbb{F}_q)$ a point of order ℓ. If $\phi : E_A \to E_{A'}$ is an isogeny with $\ker\phi = \langle P \rangle$, then it is possible to find polynomials $g(x), h(x) \in \mathbb{F}_q[x]$ such that

$$\phi(x,y) = \left(\frac{g(x)}{h(x)}, y\left(\frac{g(x)}{h(x)}\right)'\right).$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

The polynomials $g(x)$ and $h(x)$ are related by a formula stated by Elkies [30], so the main task for computing an ℓ-isogeny is that of obtaining $h(x)$ from P. In the particular case of Montgomery curves, from the formulas of [31, Theorem 1] the coefficient of the Montgomery curve $E_{A'}$ can be obtained by,

$$A' = \frac{2(1+d)}{1-d}, \quad d = \left(\frac{A-2}{A+2}\right)^{\ell} \left(\frac{h_S(1)}{h_S(-1)}\right)^{8}, \tag{1}$$

and the x-coordinate $\phi_x(x(Q))$ of the point $\phi(Q)$, by

$$\phi_x(x(Q)) = \left(\frac{x(Q)^{\ell} h_S(1/x(Q))}{h_S(x(Q))}\right)^{2}, \tag{2}$$

where $h_S(X) = \prod_{s \in S}(X - x([s]P))$ and $S = \{1, 3, \ldots, \ell - 2\}$.

In 2020, Bernstein, De Feo, Leroux and Smith [22] proposed an important improvement in the computation of isogenies. Their key idea exploits the fact that for isogeny evaluations and constructions, it is not required to compute the polynomial h itself, but only its evaluation at a given set of points. We briefly describe their method for evaluating (1) and (2), starting by pointing out that, although the map $Q \mapsto x(Q)$ is not holomorphic, an algebraic relation exists between the values. More precisely,

Lemma 1 ([22] Lemma 4.3). *Let $E/\mathbb{F}_q$ be an elliptic curve, where q is a prime power. There exist biquadratic polynomials F_0, F_1, and F_2 in $\mathbb{F}_q[X_1, X_2]$ such that*

$$(X - x(P+Q))\,(X - x(P-Q)) = X^2 + \frac{F_1(x(P), x(Q))}{F_0(x(P), x(Q))} + \frac{F_2(x(P), x(Q))}{F_0(x(P), x(Q))},$$

for all P and Q in E such that $0 \notin \{P, Q, P+Q, P-Q\}$.

This allows us to rearrange $h_S(X)$ in a baby-step giant-step style:

$$h(X) = \left(\prod_{i \in I}\prod_{j \in J}(X - x([i+j]P))(X - x([i-j]P))\right)\left(\prod_{k \in K}(X - x([k]P))\right), \tag{3}$$

where the index system $\{I, J\}$ is chosen such that $S = (I+J) \cup (I-J) \cup K$, with $K = S \setminus (I+J) \cup (I-J)$ and satisfying the following definition:

Definition 1. *Let S, I and J be finite sets of integers. The (I, J) is an index system of S if:*

1. *the maps $I \times J \to \mathbb{Z}$ defined by $(i, j) \longmapsto i + j$ and $(i, j) \longmapsto i - j$ are both injective and have disjoint images.*
2. *$I + J$ and $I - J$ are both contained in S.*

From here, they show that the factor of $h(X)$ that is related to I and J can be computed efficiently as a resultant. More precisely, letting $I \pm J$ be the union of the sets $I + J$ and $I - J$, the following result is proven.

Lemma 2 ([22] Lemma 4.9). *Let $E/\mathbb{F}_q$ be an elliptic curve, where q is a prime power. Let P be an order-n element of $E(\mathbb{F}_q)$. Let (I, J) be an index system such that I, J, $I + J$, and $I - J$ do not contain any elements of $n\mathbb{Z}$. Then*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

$$h_{I\pm J}(X) = \frac{1}{\Delta_{I,J}} Res_Z(h_I(Z), E_J(X,Z))$$

where

$$E_J(X,Z) := \prod_{i\in J} \Big(F_0(Z, x([j]P))X^2 + F_1(Z, x([j]P))X + F_2(Z, x([j]P))\Big)$$

and $\Delta_{I,J} := Res_Z(h_I(Z); D_J(Z))$ *where* $D_J(Z) := \prod_{i\in J} F_0(Z, x([j]P))$.

Remark 2. *Note that* $\Delta_{I,J}$ *in Lemma 2 does not depend on the variable* X. *The formulas (2) and (1) for computing the codomain curve and point evaluation respectively, include quotients of the polynomial* h, *so in practice,* $\Delta_{I,J}$ *in Lemma 2 does not need to be computed.*

Notice that $h_{I\pm J}$ is precisely the first parenthesis that appears in (3), which can be computed quadratically faster as a resultant, while the second parenthesis (corresponding to $h_K(X)$) is a leftover product that is still computed in linear time. Based on this, Adj, Chi-Domínguez, and Rodríguez-Henríquez [23] introduced three explicit algorithms for evaluating ℓ-isogenies: `KPS` (which computes the necessary multiples of the kernel generator), `xISOG` (which obtains the coefficient of the image curve) and `xEVAL` (which obtains the image of a point). These algorithms are reproduced here as Algorithm 1, Algorithm 2 and Algorithm 3.

Algorithm 1 `KPS`

Inputs: An elliptic curve $E_A/\mathbb{F}_q$; $P \in E_A(\mathbb{F}_q)$ of order an odd prime ℓ
Output: $\mathcal{I}, \mathcal{J}, \mathcal{K}$ such that (I, J) is an index system for S

1: $b \leftarrow \lfloor \sqrt{(\ell-1)}/2 \rfloor$
2: $b' \leftarrow \lfloor (\ell-1)/4b \rfloor$
3: $J \leftarrow \{2j+1 : 0 \le j < b\}$
4: $I \leftarrow \{2b(2i+1) : 0 \le i < b'\}$
5: $K \leftarrow S \backslash (I \pm J)$
6: $\mathcal{J} \leftarrow \{x([j]P) : j \in J\}$
7: $\mathcal{I} \leftarrow \{x([i]P) : i \in I\}$
8: $\mathcal{K} \leftarrow \{x([k]P) : k \in K\}$ v
9: **return** $\mathcal{I}$, $\mathcal{J}$ and $\mathcal{K}$

Algorithm 2 `xISOG`

Inputs: An elliptic curve $E_A/\mathbb{F}_q : y^2 = x^3 + Ax^2 + x$; $P \in E_A(\mathbb{F}_q)$ of order an odd prime ℓ; and $\mathcal{I}$,$\mathcal{J}$,$\mathcal{K}$ the output from `KPS`(P)
Output: $A' \in \mathbb{F}_q$ such that $E_A/\mathbb{F}_q : y^2 = x^3 + A'x^2 + x$ is the image curve of a separable isogeny with kernel $\langle P \rangle$

1: $h_I \leftarrow \prod_{x_i\in\mathcal{I}} (Z - x_i) \in \mathbb{F}_q[Z]$
2: $E_{0,J} \leftarrow \prod_{x_j\in\mathcal{J}} (F_0(Z, x_j) + F_1(Z, x_j) + F_2(Z, x_j)) \in \mathbb{F}_q[Z]$
3: $E_{1,J} \leftarrow \prod_{x_j\in\mathcal{J}} (F_0(Z, x_j) - F_1(Z, x_j) + F_2(Z, x_j)) \in \mathbb{F}_q[Z]$
4: $R_0 \leftarrow Res_Z(h_I, E_{0,J}) \in \mathbb{F}_q$
5: $R_1 \leftarrow Res_Z(h_I, E_{1,J}) \in \mathbb{F}_q$
6: $M_0 \leftarrow \prod_{x_k\in\mathcal{K}} (1 - x_k) \in \mathbb{F}_q$
7: $M_1 \leftarrow \prod_{x_k\in\mathcal{K}} (-1 - x_k) \in \mathbb{F}_q$
8: $d \leftarrow \left(\frac{A-2}{A+2}\right)^{\ell} \left(\frac{M_0 R_0}{M_1 R_1}\right)^8$
9: **return** $\frac{2(1+d)}{1-d}$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

Algorithm 3 `xEVAL`

Inputs: An elliptic curve $E_A/\mathbb{F}_q : y^2 = x^3 + Ax^2 + x$; $P \in E_A(\mathbb{F}_q)$ of order an odd prime ℓ; $\alpha = x(Q) = [Q_x : Q_z] \neq 0$ of a point $Q \in E_A(\mathbb{F}_q) \setminus \langle P \rangle$ and $\mathcal{I}$,$\mathcal{J}$,$\mathcal{K}$ the output from $\texttt{KPS}(P)$

Output: The image of $x(\phi(Q))$ by w, where ϕ is a separable isogeny of kernel $\langle P \rangle$.

1: $h_I \leftarrow \prod_{x_i \in \mathcal{I}}(Z - x_i) \in \mathbb{F}_q[Z]$
2: $E_{0,J} \leftarrow \prod_{x_j \in \mathcal{J}}(F_0(Z, x_j)/\alpha^2 + F_1(Z, x_j)/\alpha + F_2(Z, x_j)) \in \mathbb{F}_q[Z]$
3: $E_{1,J} \leftarrow \prod_{x_j \in \mathcal{J}}(F_0(Z, x_j)\alpha^2 + F_1(Z, x_j)\alpha + F_2(Z, x_j)) \in \mathbb{F}_q[Z]$
4: $R_0 \leftarrow Res_Z(h_I, E_{0,J}) \in \mathbb{F}_q$
5: $R_1 \leftarrow Res_Z(h_I, E_{1,J}) \in \mathbb{F}_q$
6: $M_0 \leftarrow \prod_{x_k \in \mathcal{K}}(1/\alpha - x_k) \in \mathbb{F}_q$
7: $M_1 \leftarrow \prod_{x_k \in \mathcal{K}}(\alpha - x_k) \in \mathbb{F}_q$
8: **return** $(M_0 R_0 Q_x)^2/(M_1 R_1 Q_z)^2$

2.2. Computing the Resultants

The main computational task that is required for the square-root Vélu algorithm is the evaluation of the resultants $Res_Z(h_I, E_{i,J})$, which can be achieved in constant-time via a remainder tree approach as described in [23]. This task accounts for approximately 85% of the total computation time, and depends on the size of the sets I, J, and K. For the following discussion, we set $\#J = b$, $\#I = b'$ and $\#K = \frac{\ell-1}{2} - 2bb'$. As a remainder, if $f(Z) \in \mathbb{F}_q[Z]$ splits into linear factors as $f(Z) = a\prod_{0 \leq i < b}(Z - x_i)$ and $g(Z) \in \mathbb{F}_q[Z]$, then the resultant is given by

$$Res_Z(f(Z), g(Z)) = a^b \prod_{0 \leq i < b} g(x_i). \tag{4}$$

For our purposes, the polynomials f and g are provided as a product of b' linear polynomials and b quadratic polynomials, respectively. Since the x_i values are readily available, one could compute (4) directly by evaluating g multiple times, but this would lead to a complexity of $\mathcal{O}(bb') = \mathcal{O}(\ell)$.

Instead, we begin by obtaining the polynomials f and g as a list of coefficients by multiplying together all the terms following a *product tree* approach, as shown in Figure 1.

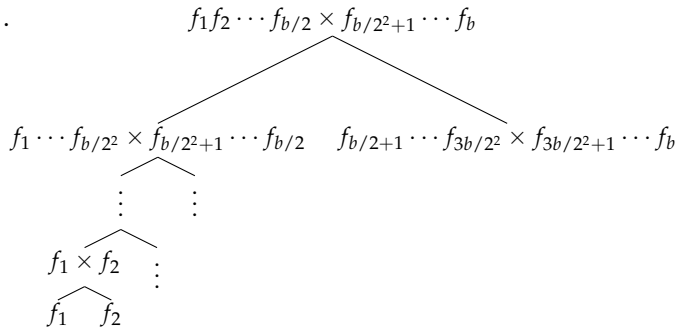

Figure 1. Diagram of the product tree for computing $f_i = (Z - x_i)$.

After the product tree for f has been built, we compute a corresponding *reciprocal tree*. Unlike the product tree, the reciprocal tree is computed from the root down. If a node in the product tree contains a polynomial F of degree m, then the corresponding node in the reciprocal tree contains $(\mathfrak{F}, c)$, where $\mathfrak{F}$ is a polynomial of degree m and c is a constant such that $rev_m(F) \cdot \mathfrak{F} = c \mod x^m$. Here, $rev_m(\cdot)$ denotes the polynomial with the list of coefficients in reverse order: if $F = \sum_{i=0}^{m} F_i x^i$, then $rev_m(F) = \sum_{i=0}^{m} F_{n-i} x^i$. The reciprocal tree is depicted in Figure 2, and is used as an auxiliary tool to construct one last tree called the *residue tree*.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

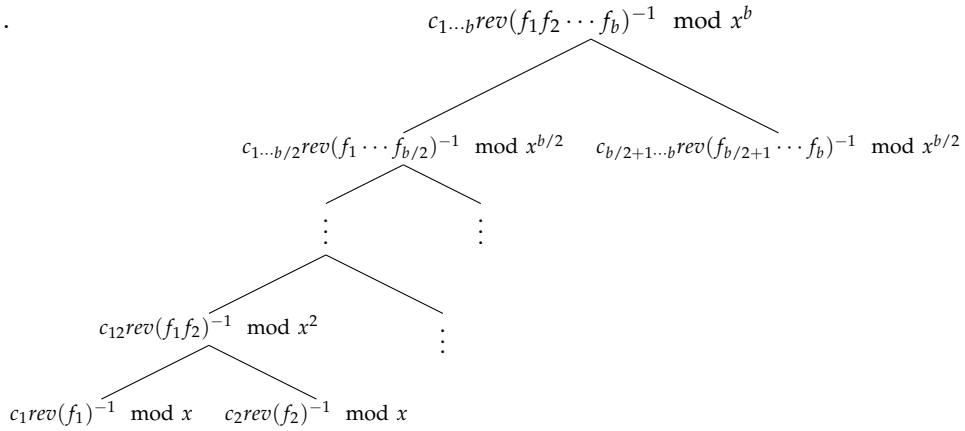

Figure 2. Diagram of the reciprocal tree, where $f_i = (Z - x_i)$, $rev(.)$ and c_i are as mentioned above.

The residue tree is also built from the root down, and as depicted in Figure 3, each of its nodes contains $cg \mod F$, where F is the polynomial in the corresponding node of the product tree of f and c the constant from the reciprocal tree. The leaves of the remainder tree contain a multiple of $g(Z) \mod (Z - x_i) = g(x_i)$, so a multiple of the resultant is readily obtained by multiplying all the leaves together. The multiple cancels out when taking the ratios at the last step of Algorithm 3 and Algorithm 2. Remarkably, this method allows us to compute (a multiple of) all of the $g(x_i)$ with a complexity of just $\tilde{\mathcal{O}}(b + b') = \tilde{\mathcal{O}}(\sqrt{\ell})$. A detailed description for the cost of the product tree, reciprocal tree, and remainder tree is presented in the Appendix A.

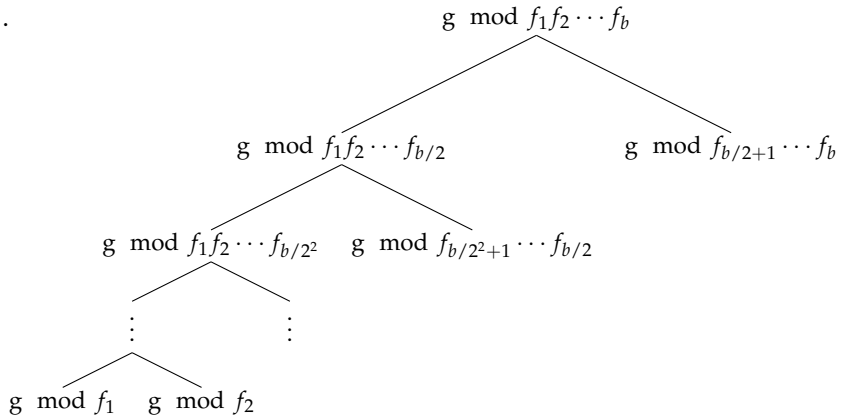

Figure 3. Diagram of the remainder tree for computing $Res_Z(f(Z), g(Z))$.

2.3. Cost model for the Sequential Square-Root Vélu

An expected cost function for `KPS`, `xISOG`, and `xEVAL` has been presented in [23], but it takes into account some redundant computations and also fails to consider the cost of the remainder tree. We now present a more accurate and optimal cost function.

As in [23], we begin by noting that a better performance is achieved when $\#K$ is as small as possible and $\#I, \#J$ are similar in size. Since we need to have $2(\#I)(\#J) + \#K = (\ell - 1)/2$, we set the size of $\#J$ to be $b := \lceil \sqrt{(\ell - 1)/2} \rceil$ and then perform division with remainder to obtain $\#I = \lfloor (\ell - 1)/(4b) \rfloor$ and $\#K \in [0, 2b]$. For simplicity, from now on we ignore rounding errors and assume the average value for $\#K$, so that $\#I \approx \#J \approx \#K \approx b \approx \sqrt{(\ell - 1)/2}$.

Next, we provide explicit formulas for the number of field multiplications in each of the three procedures as a function of b. This cost model neglects the cost of field additions and subtractions, as well as the savings that a field squaring may have over a regular field multiplication. We consider this a fair compromise, since the total number of squarings is small and the benefit of having a single

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

metric far outweighs the importance of the small error. The cost functions are expressed in terms of the cost of the various tree procedures, which are detailed in the Appendix A.

Cost function for KPS. Notice that the computation of h_I is performed in both Algorithm 3 and Algorithm 2, so instead we delegate this product tree and the corresponding reciprocal tree to KPS and perform it only once. The total cost of KPS is then reflected by obtaining b multiples of an elliptic curve point (at a cost of $6b$ field multiplications each) for each of $\mathcal{I}$, $\mathcal{J}$ and $\mathcal{K}$, and the cost of a product tree and reciprocal tree for b linear polynomials. We obtain

$$Cost_{\texttt{KPS}}(b) = 18b + ProductTree_1(b) + ReciprocalTree(b) = 4b^{\log_2(3)} + 2\log_2(b) + 16b - 2.$$

Cost function for xISOG. The degree-2 factors in $E_{0,J}$ and $E_{1,J}$ can be obtained at the cost of $5b$ field multiplications, and the complete polynomials are then obtained from two product trees of quadratic leaves. We then perform 2 residue trees to compute the two resultants of Algorithm 2, and another $2 * (b-1)$ multiplications to compute M_0 and M_1. Algorithm 2 also requires an ℓ-th power exponentiation at an average cost of $1.5\log\ell \approx 3(\log_2(b)+1)$ multiplications, which must be performed separately on the numerator and denominator when working in projective coordinates. Finally, there are another 10 multiplications for the final two steps. We obtain

$$\begin{aligned} Cost_{\texttt{xISOG}}(b) =& 5b + 2 \times ProductTree_2(b) + 2 \times ResidueTree(b) \\ &+ 2(b-1) + 2 \times 3(\log_2(b)+1) + 10, \end{aligned}$$

$$Cost_{\texttt{xISOG}}(b) = 18b^{\log_2(3)} + 6\log_2(b) - 5b + 12.$$

Cost function for xEVAL. In this case, computing the factors for $E_{0,J}$ requires $10b$ field multiplications. However, only one product tree is required since $E_{1,J}$ can be obtained by inverting the coefficient list of $E_{0,J}$, as noted in [23]. We then need two residue trees to compute the resultants in Algorithm 3, $2(2b-1)$ multiplications to compute M_0 and M_1, and 6 more multiplications for the final step. The total is

$$Cost_{\texttt{xEVAL}}(b) = 10b + ProductTree_2(b) + 2 \times ResidueTree(b) + 2(2b-1) + 6$$

$$Cost_{\texttt{xEVAL}}(b) = 15b^{\log_2(3)} + 5b + 2.$$

Total cost function. The total cost function, assuming that two points need to be pushed through the isogeny as is the case in CSIDH and SQISign, is

$$\begin{aligned} Cost_{\texttt{Vélu}}(b) =& Cost_{\texttt{KPS}}(b) + Cost_{\texttt{xISOG}}(b) + 2 \times Cost_{\texttt{xEVAL}}(b) \\ =& 52b^{\log_2(3)} + 8\log_2(b) + 21b + 14. \end{aligned} \tag{5}$$

3. Parallelizing Square-Root Vélu Formulas

In this section, we present our main results which focuses on the problem of parallelizing Vélu's formulas for isogeny computations.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

An immediate observation is that the two resultants that appear in each of Algorithm 2 and Algorithm 3, which represent the heaviest part of the computation, are completely independent of each other. Therefore, there is a simple way to parallelize the two algorithms in a 2-core implementation by assigning one resultant to each core. We show that these algorithms exhibit a good degree of parallelizability even beyond two cores, by assigning more than one core per resultant. However, doing so requires a new indexing system that allows for a clear separation of the computation between cores, which we introduce in the first subsection.

Additionally, other parts of the algorithms are also apt for parallelization, such as the computation of M_0 and M_1 where a partial product can be assigned to each core, or the computation of product trees where a subtree can be assigned to each core after a small bit of sequential work. We show also that the computation of point multiples in KPS can be parallelized with our new index system.

After detailing the new index system, provide explicit new algorithms for the square-root Vélu formulas which can be computed by assuming the availability of n-cores, where $n > 1$ is a power of two.

3.1. Construction of a New Index System

The main observation that allows us to parallelize the computation of resultants beyond two cores is that they exhibit a multiplicative property. More precisely, if I can be written as the disjoint union

$$I = I_0 \cup I_1 \cup \ldots \cup I_{n/2-1},$$

then

$$Res_Z(h_I, E_{i,J}) = Res_Z(h_{I_0}, E_{i,J}) \times Res_Z(h_{I_1}, E_{i,J}) \times \ldots \times Res_Z(h_{I_{n/2-1}}, E_{i,J}),$$

so we can assign one subset I_t to each of the cores, have them compute one subresultant each, and then multiply them all together. Doing so will require us to modify the sizes of I and J: since each resultant computation should have balanced sizes, we now require $\#J \approx \#I/(n/2)$. Accordingly, we now need $\#J \approx \sqrt{(\ell-1)/2n}$ and $\#I \approx \sqrt{(\ell-1)n/4}$. We will design such an indexing system according to the following lemma.

Lemma 3. *Let $n > 1$ be a power of two, ℓ be an odd positive integer and consider the set $S = \{1, 3, ..., \ell\}$. If $b = \lfloor \sqrt{(\ell-1)/2n} \rfloor \geq 1$ and $b' = \lfloor (\ell-1)/2nb \rfloor$, then (I, J) is an index system for S, where*

$$J := \{2j + 1 : 0 \leq j < b\},$$

$$I = \bigcup_{t=0}^{n/2-1} I_t, \text{ and}$$

$$I_t := \{2b(2t+1) + 2bni : 0 \leq i < b'\} \quad \text{for } 0 \leq t < n/2.$$

Moreover, $I_t \cap I_{\tilde{t}} = \emptyset$ if and only if $t \neq \tilde{t}$.

Proof of Lemma 3. Suppose that

$$(2b(2t+1) + 2bni_1)) - (2b(2\tilde{t}+1) + 2bni_2)) = (2j_1 + 1) - (2j_2 + 1),$$

where $0 \leq j_i, j_2 < b$, $0 \leq i_1, i_2 < b'$ and $j_1 \neq j_2$ and $i_1 \neq i_2$ (t and $\tilde{t}$ can be equal). Then,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

$$4b(t-\tilde{t})+2bn(i_1-i_2)=2(j_1-j_2),$$

so

$$2b(t-\widetilde{t}+n(i_1-i_2)/2)=j_1-j_2$$

but this is impossible since $0<|t-\tilde{t}|<n$ and thus $0<|j_1-j_2|<b<2b|(t-\tilde{t}+n(i_1-i_2)/2)|$. Therefore, the map $I\times J\to\mathbb{Z}$ defined by $(i,j)\longmapsto i+j$ is injective. Similarly, the map $(i,j)\longmapsto i-j$ is injective and therefore, both have disjoint images.

On the other hand, the sets $I+J$ and $I-J$ are both contained in S. It is clear that the elements in these sets are odd integers. Since for all $0\leq t<n/2$, $0\leq j<b$ and $0\leq i<b'$ we have the following:

$$(2b(2t+1)+2bni)+(2j+1)\leq 2b(2(n/2-1)+1+ni)+(2b+1)=2nbi+2nb+1$$

$$(2b(2t+1)+2bni)+(2j+1)\leq 2nb\left(\frac{(\ell-1)}{2nb}-1\right)+2nb+1=\ell.$$

Similarly, $1\leq(2b(2t+1)+2bni)-(2j+1)$ for all $0\leq t<n/2$, $0\leq j<b$ and $0\leq i<b'$ since the minimum of I is $(2b(2t+1)+2bni)=2b$ when $t=i=0$ and the maximum of J is $2j+1=2b-1$ when $j=b-1$. □

Note that we have chosen to partition I into only $n/2$ subsets because, when n cores are available, we assign $n/2$ cores to each of the two resultants. However, when computing point multiples during KPS we do have all n cores available for working on the $n/2$ subsets. Therefore, it will be convenient to further divide each I_t into two subsets to obtain a total of n half-sized subsets. Note that some subsets will have fewer elements than others because $\#I$ need not be a multiple of n. In practice, this is problematic because it is crucial for each core to perform exactly analogous tasks to guarantee performance. In order to achieve a balanced computation, we will add additional elements to the subsets that are lacking, ensuring that each subset contains the same number of elements, while ignoring redundant multiples in future steps. This idea is inspired by [22, Example 3.7] and the parallelization technique proposed by Costello in [32] to compute the set $[i]P_{1\leq i\leq d}$. We provide an example to illustrate this approach.

Example 1. *Let $\ell=89$ and $n=4$. From the Lemma 3 we get*

$$J:=\{2j+1:0\leq j<3\},\quad I=\bigcup_{t=0}^{1}I_t,$$

$$I_t:=\{2b((2t+1)+ni):0\leq i<3\},\qquad \textit{for all } 0\leq t<2.$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

So $I_0 := \{2b, 10b, 18b\}$ *and* $I_1 := \{6b, 14b, 22b\}$. *Instead of this, we can consider the following sets:*

$$II_t := \{2b((2t+1)+2ni) : 0 \leq i < 2\} \quad \text{for all } 0 \leq t < 4.$$

So $II_0 := \{2b, 18b\}$, $II_1 := \{6b, 22b\}$, $II_2 := \{10b, 26b\}$ *and* $II_3 := \{14b, 30b\}$.

We will now demonstrate that the partitioning of the set I, without taking into account the additional elements added to ensure equal-sized subsets, still forms an index system.

Proposition 1. *Let us assume the same conditions of the Lemma 3. Let* $c = \lceil b'/2 \rceil > 2$. *If*

$$I' := \bigcup_{t=0}^{n-1} II_t \setminus \bigcup_{t=n/2}^{n-1} \{w_t\} \quad \text{with } w_t := 2b(2t+1+2n(c-1))(b' \bmod 2),$$
$$\text{and } II_t := \{2b(2t+1)+bn(2i) : 0 \leq i < c\}$$
$$\text{for all } 0 \leq t < n,$$

$$II_t := \{2b(2t+1)+bn(2i) : 0 \leq i < c\}, \quad \text{for all } 0 \leq t < n,$$

$$I' := \bigcup_{t=0}^{n-1} II_t \setminus \bigcup_{t=n/2}^{n-1} \{w_t\}, \quad \text{with } w_t := 2b(2t+1+2n(c-1))(b' \bmod 2),$$

then $I' = I$. *In particular,* (I', J) *is an index system for* S.

Proof of Proposition 1. If b' is an even integer, it is clear that $II_t \subset I$, for all $0 \leq t < n$ and thus $\bigcup_{t=n/2}^{n-1} \{w_t\} = \{0\}$. Note that $0 \leq 2i \leq 2c = b'$ and considering that $\bigcup_{t=0}^{n-1} II_t$ and I have the same cardinality, we conclude that they are equal. If b' is an odd integer, all $0 \leq t < n/2$ we have that $II_t \subset I$. When $n/2 \leq t < n$, dropping the last element $w_t := 2b(2t+1+2n(c-1))$ in each $II_{n/2}, ..., II_{n-1}$, we have $II_t \subset I$. Note that the cardinality of $\bigcup_{t=n/2}^{n-1} \{w_t\}$ is $n/2$ and the cardinality of $\bigcup_{t=0}^{n-1} II_t$ is $nc = n((b'+1)/2) = nb' + n/2 = \#I + n/2$. □

Similarly, we consider a partition of $K = S \setminus (I \pm J) = \{2bb'n+1, ..., \ell-2, \ell\}$ into n sets $K_0, .., K_{n-1}$ with $d = \lceil \#K/n \rceil$ elements,

$$K_t = \{2bb'n + 2(t+1) + 2nk : 0 \leq k < d\}, \quad 0 \leq t < n$$

However, when $\#K$ is not divisible by n, we will add one additional element to each set $K_{d \bmod (n)}, ..., K_{n-1}$, ensuring that each subset contains the same number of elements.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

The additional elements that were added to balance the size of the sets are not taken into account when calculating the corresponding products for each set.

3.2. Parallelized Algorithms

We now describe the construction of parallelized versions of algorithms `KPS`, `xISOG`, and `xEVAL` assuming that n cores are available, which we refer to as `KPS-Parallel`, `xISOG-Parallel`, and `xEVAL-Parallel`.

The `KPS-Parallel` algorithm, presented as Algorithm 4, is based on the new index system from Proposition 1: instead of computing the sets $\mathcal{I}$ and $\mathcal{K}$ as in the original algorithm, `KPS-Parallel` computes the sets $\mathcal{II}_t$ and $\mathcal{K}_t$, as well as the polynomial product tree for h_{II_t}, by assigning one core to each value of $0 \leq t < n$. Note that the computation of the resultants requires a reciprocal tree for each of the h_{I_t}, so the polynomials $\{h_{II_t},\ 0 \leq t < n\}$ are multiplied pair-wise to obtain the polynomials $\{h_{I_t},\ 0 \leq t < n/2\}$. We then compute the residue trees, which are built from the root down: the root is computed for each tree using $n/2$ cores, and then the two subtrees of each tree are computed in parallel, using two cores per tree, for a total of n cores. Note that although the computations for the root of the product and reciprocal trees, corresponding to lines 11 and 12 of 4, should only be ran with $n/2$ cores, we instead have all cores running with half of them repeating the work over dummy arrays $\mathcal{I}_t$ for $t > n/2$. These computations are of course redundant, but ensure that the workload is balanced.

Algorithm 4 `KPS-Parallel`

Inputs: An elliptic curve $E_A/\mathbb{F}_q$; $P \in E_A(\mathbb{F}_q)$ of order ℓ an odd prime and a number of cores n which is a power of 2

Output: b, d integers, $\mathcal{J}$, $\mathcal{K}$ arrays of field elements, $h_{I_0}, ..., h_{I_{n/2-1}}$ polynomials and $rtree_{h_{I_0}}, ..., rtree_{h_{I_{n/2-1}}}$ reciprocal trees

1: $b \leftarrow \lfloor \sqrt{(\ell-1)/2n} \rfloor$; $b' \leftarrow \lfloor (\ell-1)/2nb \rfloor$; $d \leftarrow (\ell-1)/2 - nbb'$;
2: $b' \leftarrow \lceil b'/2 \rceil$; $d \leftarrow \lceil d/n \rceil$ // $\lceil d/n \rceil > 2$ and $\lceil b'/2 \rceil > 1$
3: $\mathcal{J} \leftarrow \{x([m]P) : m \in \{2j+1 : 0 \leq j < b\}\}$
4: $B_{\mathcal{I}} \leftarrow \{x([m]P) : m \in \{2b(2t+1) : 0 \leq t < n\}\}$
5: $B_{\mathcal{K}} \leftarrow \{x([m]P) : m \in \{2(t+1) : 0 \leq t < n\}\}$
6: **for** $t \in \{0, 1, ..., n-1\}$ **in parallel do**
7: $\quad \mathcal{II}_t \leftarrow \{B_{\mathcal{I}}[t] + x([m]P) : m \in \{2b(2ni) : 0 \leq i < b'\}\}$
8: $\quad \mathcal{K}_t \leftarrow \{B_{\mathcal{K}}[t] + x([m]P) : m \in \{2nk : 0 \leq k < d\}\}$
9: $\quad h_{II_t} \leftarrow \prod_{x_i \in \mathcal{II}_t} (Z - x_i)$.
10: $\quad$ **barrier** // Synchronize cores
11: $\quad h_{I_t} \leftarrow h_{II_t} \cdot h_{II_{(t+n/2)}}$
12: $\quad rtree_{h_{I_t}} \leftarrow ReciprocalTreeRoot(h_{I_t})$ // Fills in only the root node
13: $\quad$ **if** $t < n/2$ **then**
14: $\quad\quad rtree_{h_{I_t}} \leftarrow ReciprocalLeftSubtree(h_{I_t})$ // Fills in only the left subtree
15: $\quad$ **else**
16: $\quad\quad rtree_{h_{I_{t-n/2}}} \leftarrow ReciprocalRightSubtree(h_{I_{t-n/2}})$ // Fills in only the right subtree
17: $\quad$ **end if**
18: **end for**
19: $\mathcal{K} \leftarrow \mathcal{K}_0 \cup \ldots \cup \mathcal{K}_{n-1}$
20: **return** $b, d, \mathcal{J}, \mathcal{K}, h_{I_0}, \ldots, h_{I_{n/2-1}}$ and $rtree_{h_{I_0}}, ..., rtree_{h_{I_{n/2-1}}}$

For the algorithms `xISOG-Parallel` and `xEVAL-Parallel`, the set J is partitioned into n subsets J_t, and each core computes the polynomial product tree corresponding to one subset. The roots of the n trees are then multiplied together sequentially to obtain the full polynomial $E_{i,J}$. Next, each core computes one of the subresultants $Res_Z(h_{I_t}, E_{i,J})$ using the reciprocal trees from `KPS-Parallel`,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

and the subresultants are multiplied sequentially at the end to obtain the two principal resultants. As for the computations related to $\mathcal{K}$, it is also split into subsets $\mathcal{K}_t$ so that each core can compute a subproduct of M_i, and the subproducts are multiplied together at the end.

Algorithm 5 `xISOG-Parallel`

Inputs: An elliptic curve $E_A/\mathbb{F}_q : y^2 = x^3 + Ax^2 + x$; $P \in E_A(\mathbb{F}_q)$ of order an odd prime ℓ; b, d, $\mathcal{J}$, $\mathcal{K}$, $h_{I_0}, ..., h_{I_{n/2-1}}$ and $rtree_{h_{I_0}}, ..., rtree_{h_{I_{n/2-1}}}$ from `KPS-Parallel`; and a number of cores n which is a power of 2.

Output: $A' \in \mathbb{F}_q$ such that $E_A/\mathbb{F}_q : y^2 = x^3 + A'x^2 + x$ is the image curve of a separable isogeny with kernel $\langle P \rangle$

for $t \in \{0, 1, ..., n-1\}$ **in parallel do**
  $\mathcal{J}_t \leftarrow \{x([j]P) \in \mathcal{J} : t\lceil b/n \rceil \leq j < \lceil b/n \rceil + t\lceil b/n \rceil\}$
  $E_{0,J_t} \leftarrow \prod_{x_j \in \mathcal{J}_t}(F_0(Z, x_j) + F_1(Z, x_j) + F_2(Z, x_j))$
  $E_{1,J_t} \leftarrow \prod_{x_j \in \mathcal{J}_t}(F_0(Z, x_j) - F_1(Z, x_j) + F_2(Z, x_j))$
  $\mathcal{K}_t \leftarrow \{x([k]P) \in \mathcal{K} : td \leq k < d + td\}$
  $M_{0,t} \leftarrow \prod_{x_k \in \mathcal{K}_t}(1 - x_k)$
  $M_{1,t} \leftarrow \prod_{x_k \in \mathcal{K}_t}(-1 - x_k)$
end for
$E_{0,J} \leftarrow E_{0,J_0} \times \ldots \times E_{0,J_{n-1}} \in \mathbb{F}_q[Z]$
$E_{1,J} \leftarrow E_{1,J_0} \times \ldots \times E_{1,J_{n-1}} \in \mathbb{F}_q[Z]$
$M_0 \leftarrow M_{0,0} \times \ldots \times M_{0,n-1} \in \mathbb{F}_q$
$M_1 \leftarrow M_{1,0} \times \ldots \times M_{1,n-1} \in \mathbb{F}_q$
for $t \in \{0, 1, ..., n-1\}$ **in parallel do**
  if $t < n/2$ **then**
    $R_{0,t} \leftarrow Res_Z(h_{I_t}, E_{0,J}, rtree_{I_t})$
  else
    $R_{1,(t-n/2)} \leftarrow Res_Z(h_{I_{t-n/2}}, E_{1,J}, rtree_{I_{t-n/2}})$
  end if
end for
$R_0 \leftarrow R_{0,0} \times \ldots \times R_{0,n/2-1} \in \mathbb{F}_q$
$R_1 \leftarrow R_{1,0} \times \ldots \times R_{1,n/2-1} \in \mathbb{F}_q$
$d \leftarrow \left(\frac{A-2}{A+2}\right)^{\ell} \left(\frac{M_0 R_0}{M_1 R_1}\right)^8$
return $2(1+d)/(1-d)$

In the next subsection, we present a detailed analysis that estimates the cost of computing the image curve of an isogeny and the evaluation of a point using the procedures `KPS-Parallel`+ `xISOG-Parallel`+ `xEVAL-Parallel`.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

Algorithm 6 `xEVAL-Parallel`

Inputs: An elliptic curve $E_A/\mathbb{F}_q : y^2 = x^3 + Ax^2 + x$; $P \in E_A(\mathbb{F}_q)$ of order an odd prime ℓ; $\alpha = x(Q) \neq 0$ for a point $Q \in E_A(\mathbb{F}_q) \setminus \langle P \rangle$; b, d, $\mathcal{J}$, $\mathcal{K}$, $h_{I_0}, ..., h_{I_{n/2-1}}$ and $rtree_{h_{I_0}}, ..., rtree_{h_{I_{n/2-1}}}$ from `KPS-Parallel`; and a number of cores n which is a power of 2.

Output: $x(\phi(Q))$, where ϕ is a separable isogeny of kernel $\langle P \rangle$

for $t \in \{0, 1, ..., n-1\}$ **in parallel do**
  $\mathcal{J}_t \leftarrow \{x([j]P) \in \mathcal{J} : t\lceil b/n \rceil \leq j < \lceil b/n \rceil + t\lceil b/n \rceil\}$
  $E_{0,J_t} \leftarrow \prod_{x_j \in \mathcal{J}_t} (F_0(Z, x_j)\alpha^2 + F_1(Z, x_j)\alpha + F_2(Z, x_j))$
  $E_{1,J_t} \leftarrow \prod_{x_j \in \mathcal{J}_t} (F_0(Z, x_j)\alpha^{-2} + F_1(Z, x_j)\alpha^{-1} + F_2(Z, x_j))$
  $K_t \leftarrow \{x([k]P) \in \mathcal{K} : td \leq k < d + td\}$
  $M_{0,t} \leftarrow \prod_{x_k \in \mathcal{K}_t} (\alpha - x_k)$
  $M_{1,t} \leftarrow \prod_{x_k \in \mathcal{K}_t} (\alpha^{-1} - x_k)$
end for
$E_{0,J} \leftarrow E_{0,J_0} \times \ldots \times E_{0,J_{n-1}} \in \mathbb{F}_q[Z]$
$E_{1,J} \leftarrow E_{1,J_0} \times \ldots \times E_{1,J_{n-1}} \in \mathbb{F}_q[Z]$
$M_0 \leftarrow M_{0,0} \times \ldots \times M_{0,n-1} \in \mathbb{F}_q$
$M_1 \leftarrow M_{1,0} \times \ldots \times M_{1,n-1} \in \mathbb{F}_q$
for $t \in \{0, 1, ..., n-1\}$ **in parallel do**
  if $0 \leq t < n/2$ **then**
    $R_{0,t} \leftarrow Res_Z(h_{I_t}, E_{0,J}, rtree_{I_t})$
  else
    $R_{1,(t-r)} \leftarrow Res_Z(h_{I_{t-n/2}}, E_{1,J}, rtree_{I_{t-n/2}})$
  end if
end for
$R_0 \leftarrow R_{0,0} \times \ldots \times R_{0,n/2-1} \in \mathbb{F}_q$
$R_1 \leftarrow R_{1,0} \times \ldots \times R_{1,n/2-1} \in \mathbb{F}_q$
return $(M_0 R_0 Q_x)^2 \,/\, (M_1 R_1 Q_z)^2$

3.3. Cost Analysis of the Parallel Square-Root Vélu

In this section, we will provide an estimation of the cost that is expected when performing the `KPS-Parallel`, `xISOG-Parallel`, and `xEVAL-Parallel` procedures, which is analogous to the one presented in subsection 2.3 for the sequential algorithms. Recall that for our new index system from subsection 3.1 we have $\#J = \lfloor \sqrt{(\ell-1)/(2n)} \rfloor$, where ℓ is the degree of the isogeny and n represents the number of cores available, while each of the I_t subsets has a size of $b' := \lfloor (\ell-1)/(2n(\#J)) \rfloor$ for a total size of $\#I = (n/2)\lfloor (\ell-1)/(2n(\#J)) \rfloor$. It follows that

$$\#K := \frac{\ell-1}{2} - 2(\#I)(\#J),$$

$$\#K \leq \frac{\ell-1}{2} - 2\left(\sqrt{\frac{\ell-1}{2n}} - 1\right)\left(\frac{n}{2}\left(\sqrt{\frac{\ell-1}{2n}} - 1\right)\right),$$

$$\#K = \sqrt{2n(\ell-1)} - n.$$

As before, we will ignore rounding errors and assume that $\#K$ takes the middle value of this range (neglecting the second term), so that $\#I \approx nb/2$, $\#J \approx b$, and $\#K \approx nb$ for $b = \sqrt{(\ell-1)/(2n)}$.

Proposition 2. *The cost of computing the parallel Algorithm* `KPS-Parallel` *with n cores is*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

$$Cost_{\texttt{KPS-Parallel}}(b,n) = \frac{8}{3}b^{\log_2(3)} + 4\log_2(b) + 15b + 12n - 18,$$

field multiplications.

Proof of Proposition 2. In order to find all the point multiples in K, we first compute the first n multiples sequentially and then core t start from the t-th multiple and take steps of size n. This leads to wall-clock time of $n + \#K/n - 1$ point operations, which involve 6 field multiplications each. A similar approach is taken for computing all the multiples in the $\mathcal{II}_t$ sets, while the ones in $\mathcal{J}$ are computed sequentially as this is the smallest of the sets. A product tree is constructed for each of the h_{II_t} in parallel, and line 11 involves multiplying two polynomials of degree $b/2$. As for the reciprocal trees, we incur the cost of the root node and then just half of the remainder cost since there is one core working on each subtree, so its cost is $(Reciprocal(b) + ReciprocalTree(b))/2$, where $Reciprocal(b) = b^{\log_2(3)} + b + 2\log_2(b) - 2$ is the cost of the root node alone. The total cost is then

$$\begin{aligned} Cost_{\texttt{KPS-Parallel}}(b,n) =& 6(n+b-1) + 6(n+b/2-1) + 6b \\ &+ ProductTree_1(b/2) + PolyMul(b/2) \\ &+ (Reciprocal(b) + ReciprocalTree(b))/2, \end{aligned}$$

and the proposition follows after considering the cost functions in the Appendix A.

□

We now focus on Algorithm 5:`xISOG-Parallel` which computes the coefficient of the image curve, and on Algorithm 6:`xEVAL-Parallel` which computes the isogeny evaluation at a specific point.

Proposition 3. *The cost of computing the parallel Algorithm* `xISOG-Parallel` *with n cores is*

$$\begin{aligned} &Cost_{\texttt{xISOG-Parallel}}(b,n) = \\ &\quad 6b^{\log_2(3)} + 3(n^2 - n + 2)\left(\frac{b}{n}\right)^{\log_2(3)} + 3\log_2(b\sqrt{n}) - b - \frac{b}{n} + \frac{5}{2}n + \frac{11}{2}, \end{aligned}$$

field multiplications.

Proof of Proposition 3. We begin by considering the cost of computing the polynomials $E_{0,J}$ and $E_{1,J}$ in Algorithm 5, where $\mathcal{J}$ comes from `KPS-Parallel`. We partition $\mathcal{J}$ into the subsets $\mathcal{J}_0, \ldots, \mathcal{J}_{n-1}$ of size (b/n) each, and compute one sub-product tree E_{0,J_t} per subset J_t concurrently. The cost of computing the factors of each subset is given by $5(b/n)$ field multiplications, then a product tree is computed for b/n quadratic polynomials on each core, and the tree roots are multiplied together sequentially to obtain the complete polynomial $E_{0,J}$. This last step involves multiplying together n polynomials of degree $2b/n$ each, which we denote as $LinearProduct_{2b/n}(n)$. An identical procedure is used to compute $E_{1,J}$ as well.

Using the residue trees $rtree_{h_{I_0}}, ..., rtree_{h_{I_{n/2-1}}}$ from `KPS-Parallel`, each core then compute a subresultant using a residue tree of size b/n, and the subresultants are multiplied sequentially at a cost of $n/2 - 1$ multiplications per resultant.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

For each of the M_i, each core computes a subproduct of size b at a cost of $b-1$ multiplications, and then combining the subproducts takes another $n-1$ multiplications.

Finally, we use two cores to compute the ℓ-th power exponentiation in Algorithm 5 for the numerator and denominator concurrently at a cost of about $1.5\log\ell \approx 3/2 + 3\log_2(b) + 3\log(n)/2$, and the last few products take 10 more multiplications.

The total cost is then

$$\begin{aligned} Cost_{\texttt{xISOG-Parallel}}(b,n) =& 5b/n + 2\times ProducTree_2(b/n) + 2\times LinearProduct_n(2b/n) \\ &+ ResidueTree(b) + n/2 - 1 \\ &+ 2\times(b+n-2) \\ &+ 3/2 + 3\log_2(b) + 3\log(n)/2 + 10, \end{aligned}$$

and the proposition follows after considering the cost functions in the Appendix A. □

Proposition 4. *The cost of computing the parallel Algorithm* `xEVAL-Parallel` *with n cores is*

$$Cost_{\texttt{xEVAL-Parallel}}(b,n) = \qquad 6b^{\log_2(3)} + \frac{3}{2}(n^2-n+2)\left(\frac{b}{n}\right)^{\log_2(3)} + b + \frac{7b}{n} + \frac{5}{2}n.$$

field multiplications.

Proof. The proof of the current proposition follows a similar structure to the previous proposition, with the main distinction lying in the polynomials $E_{0,J}$ $E_{1,J}$ $M_{0,t}$ and $M_{1,t}$. As before, the factors of $E_{0,J}$ can be computed with a cost of $10(b/n)$, and only one product tree is needed since $E_{1,J}$ is obtained at no additional cost. For each of M_0 and M_1, each core is still working with a subset of size b, but now there is a cost of b scalar multiplications for computing the factors in a subset, $b-1$ for the subproduct of each subset, and $n-1$ for combining the subproducts. There is no exponentiation in this case, and the final steps require only 6 additional multiplications. The total cost is then

$$\begin{aligned} Cost_{\texttt{xEVAL-Parallel}}(b,n) =& 10b/n + ProducTree_2(b/n) + LinearProduct_n(2b/n) \\ &+ ResidueTree(b) + n/2 - 1 \\ &+ 2(2b+n-2) \\ &+ 6, \end{aligned}$$

and the proposition follows after considering the cost functions in the Appendix. □

The next theorem summarizes our cost analyzis, and follows immediately from the previous propositions.

Theorem 1. *The expected total cost of the algorithms* `KPS-Parallel`, `xISOG-Parallel` *and twice* `xEVAL-Parallel` *expressed in field multiplications, assuming that two points need to be pushed through the isogeny, is given by*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

$$\begin{aligned} Cost_{\texttt{Vélu-Parallel}}(b,n) =& Cost_{\texttt{KPS-Parallel}}(b,n) + Cost_{\texttt{xISOG-Parallel}}(b,n) \\ &+ 2 \times Cost_{\texttt{xEVAL-Parallel}}(b,n) \\ =& \frac{62}{3} b^{\log_2(3)} + 6(n^2 - n + 2)\left(\frac{b}{n}\right)^{\log_2(3)} + 7\log_2(b) \\ &+ \frac{13b}{n} + 16b + \frac{3}{2}\log_2(n) + \frac{39}{2}n - \frac{25}{2}. \end{aligned} \tag{6}$$

Remark 3. *While it may be tempting to compare (6) directly to (5), a direct comparison would be incorrect since the definitions of b (the size of the set J) are different in each case. The fair comparison would be in terms of ℓ, where (5) has $b = \sqrt{(\ell-1)}/2$, whereas (6) has $b = \sqrt{(\ell-1)/(2n)}$.*

4. Experimental Results

We implemented our parallel version of the square-root Vélu algorithm in the C programming language, leveraging OpenMP for parallelization. This implementation has been integrated into a fork of the SQALE'd CSIDH library [12], where we can measure the savings due to parallelism for isogenies of each prime degree $\ell | p+1$. In order to visualize the progression of the performance with the isogeny degree, we have incorporated a new 1792-bit prime

$$\begin{aligned} p_{test} := \ & \texttt{0x4FA4E8C57C4EFF02D0650FE3AFC0413536E72253101645B1387DA2DD519C17FBEC}\ldots \\ & \texttt{69843C04DEAEA2DB59CDDDA7876B514C101A1DF0D96778BD72A3C51844BB0196}\ldots \\ & \texttt{F73F1DDBFEC980A4BB3B200A4E618C54621ADB35B5E4B0545F5BE025D63}\ldots \\ & \texttt{BC914AB11A882AD78B6203C57A31031B98B6C104DC99AC9A4532DEC0C293}\ldots \\ & \texttt{0F8AE51B008E4BA6D26E56C736D3C067C8F2DFDF7F8206B444A42D39E0F4D82FF}\ldots \\ & \texttt{3FD0EB1DF44B31DDCDE876E658489E1CA359DAF5868A6C22E8455B4A4F7679F6}\ldots \\ & \texttt{2B0C30D8883D2B79931C19E4737C3CC33056461E9C96A175D94B594B2A9EAB4B6B6303}, \end{aligned}$$

where $p_{test} + 1 = 4\prod_{i=0}^{107} \ell_i$ for odd primes ℓ_i that are roughly evenly spaced between $\ell_0 = 19$ and $\ell_{107} = 321193$ (the degree of the isogeny in the attack of [5]). This prime was chosen so that isogenies of each degree ℓ_i can be evaluated using only $\mathbb{F}_{p_{test}}$ arithmetic (for which we also provide an assembly code implementation). As previously mentioned, this model most closely resembles the scenario in CSIDH as opposed to applications where isogenies are performed over quadratic fields or larger extensions. However, we make this choice for concreteness while still expecting our results to extrapolate well to other scenarios since the main savings come from reducing the number of field multiplications by a factor that is agnostic to the field itself. Our library can be found at https://github.com/TheParallelSquarerootVeluTeam/Parallel-Squareroot-Velu. The benchmark results presented in this section are from experiments conducted using an Intel(R) Xeon(R) Gold 6230R processor equipped with 26 physical cores, operating at 2.10 GHz. Turbo boost and Hyper-threading technologies were disabled during these experiments.

Our experimental results are illustrated in Figure 4. We measured the total computational time for executing `KPS-Parallel`, `xISOG-Parallel`, and two iterations of `xEVAL-Parallel` for each of the prime degrees ℓ_i using 2, 4 and 8 cores, and compared to the computational time of the original sequential implementation of [12]. In Figure 4 we compare the observed speedup factors against the theoretical speedup using (6) and (5). We observe there is a more important difference for the eight-core implementation. Moreover, we notice that in situations where the degree of the isogeny is not sufficiently large, the overhead associated with synchronization and the computational steps involved tends to offset any potential performance gains achieved by employing eight cores. However, the speedup factor can be seen to achieve levels close to the asymptotic value starting from $\ell \approx 10000$.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

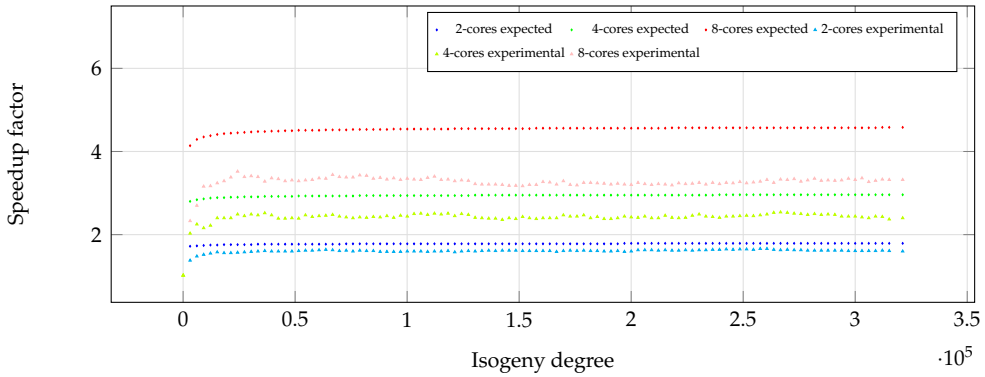

Figure 4. Speedup factors of timings of the joint cost `KPS-Parallel` + `xISOG-Parallel` + 2`xEVAL-Parallel` over $\mathbb{F}_{\text{ptest}}$ for each odd prime degree $\ell \mid (p_{test}+1)$. Experimental timings correspond to an average of 100 runs, whereas the expected savings are obtained from the estimated number of field multiplications from (6) and (5).

To showcase our results in a more practical setting, we also implemented and benchmarked our parallelized square-root Vélu algorithms in the context of SQALE'd CSIDH-9216. In its dummy-free variant, SQALE'd CSIDH-9216 performs the operations corresponding to `KPS` + `xISOG` + 2`xEVAL` exactly once for each of the 333 smallest odd primes (excluding 263), over a 9216-bit prime field. The experimental timings for each of the three routines, summed over all the degrees, are shown in Table 1. Our two-core implementation achieves speedup factors of 1.23, 1.70, and 1.92 for `KPS-Parallel`, `xISOG-Parallel`, and `xEVAL-Parallel`, respectively, while for the four-core implementation we obtained acceleration factors of 1.42, 2.27, and 3.02. While these accelerations span the entirety of the Vélu-related computations in the protocol, it is important to stress that they cannot be translated easily into an acceleration for the protocol as a whole, since the protocol includes other non-negligible computations (most notably elliptic curve point scalar multiplications).

Table 1. Aggregate computational time (in gigacycles[1]) measured for the single-core square-root Vélu procedures `KPS`, `xISOG`, and `xEVAL` described in §2.1, and of the two-core and four-core implementations of the parallel analogues described in §3.2. The costs are summed over each odd prime degree $\ell \mid (p_{9216}+1)$, and the `Total` column corresponds to `KPS` + `xISOG` + 2`xEVAL`. Timings correspond with the average of 100 runs. (Gcc) is the number of gigacycles and (SF) is the experimental speedup factor considering the single core implementation as a baseline.

Algorithm	Single core	Algorithm	Two cores		Four cores	
	Gcc		Gcc	SF	Gcc	SF
`KPS`	23.6	`KPS-Parallel`	19.2	1.23	16.6	1.42
`xISOG`	58.5	`xISOG-Parallel`	34.5	1.70	25.8	2.27
`xEVAL`	59.5	`xEVAL-Parallel`	31.0	1.92	19.7	3.02
`Total`	201.1		115.7	1.74	81.8	2.46

[1] One gigacycle is one billion clock cycles.

5. Discussion

Since `KPS-Parallel` performs some of its computational tasks sequentially, while others only benefit from half of the total number of cores, it is not surprising that its speedup factor is the lowest out of the three square-root Vélu routines. Nonetheless, as illustrated by Table 1, this is not too important as its total cost is comparatively small. Fortunately, the largest accelerations come from the `xEVAL-Parallel`, which apart from being the most expensive of the three is often required to be evaluated multiple times for different points.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

Both our experimental and theoretical results show that parallelizing with at least two cores yields strong improvements, which are important considering that they can be combined with other improvements such as vectorized field arithmetic. On the downside, it is also noticeable that the speedup factors in our implementations do not scale well when transitioning to four- and eight-core implementations. From eight cores onward, even the maximum speedup derived from the theoretic estimates suggests that the utilization that can be achieved may not be attractive for practical applications. The intuition behind these diminishing returns on the parallelization effectiveness is easy to explain, as the size of the resultants in algorithm 5 and 6 only decreases as $1/\sqrt{n}$.

Acknowledgments: This work started when Chávez-Saab, J. and Ortega, O. were doing an internship at the Technology Innovation Institute (TII) under the guidance of Prof. Dr. Rodríguez-Henríquez F. We thank TII for sponsoring such an internship. We thank ANID for the study scholarship to Ortega, O. grant number 21190301. We also thank Dr. Chi-Domínguez J. and Dr. Zamarripa-Rivera L. for valuable discussion on an early version of this manuscript. Additionally, this work has received partial funding to facilitate the use of a server in CINVESTAV-IPN in Mexico which was used for our tests.

Appendix A. Cost Functions for Polynomial Operations

In this section, we detail the costs in field multiplications associated with various operations in both the sequential and parallel versions of square-root Vélu.

Polynomial Multiplication. The basic step is the multiplication of two polynomials of equal degree d, which we denote as $PolyMul(d)$. We use a Karatsuba strategy for polynomial multiplication, so the cost in field multiplications can be expressed as in [33]

$$PolyMul(d) = d^{\log_2(3)}.$$

Multiplication of Multiple Polynomials. Suppose we want to multiply together m polynomials of degree d each. If m is small, we opt for multiplying the polynomials one by one: in step 1 we are multiplying together 2 polynomials of degree d, and in step t for $1 < t < m$ we are multiplying a polynomial of degree td by another of degree d. By partitioning the larger polynomial, we can perform this multiplication via t polynomial multiplications of degree d by d, so the total cost is

$$LinearProduct_m(d) = \sum_{t=1}^{m-1} t \cdot PolyMul(d) = \frac{m(m-1)}{2} PolyMul(d) = \frac{m(m-1)d^{log_2(3)}}{2}.$$

On the other hand, if m is large then we opt for a product tree strategy, where the input polynomials are placed at the leaves of the tree and each node contains the product of its two child nodes. We assume for simplicity that m is a power of 2. At level i (where i=0 corresponds to the root), we need to compute 2^i nodes, each of which is a product of two polynomials of degree $md/2^{i+1}$. The total cost is therefore

$$\begin{aligned} ProductTree_d(m) &= \sum_{i=0}^{\log_2(m)-1} 2^i \times PolyMul(md/2^{i+1}) = \sum_{i=0}^{\log_2(m)-1} 2^i \left(\frac{md}{2^{i+1}}\right)^{\log_2(3)}, \\ &= (md)^{\log_2(3)} - md^{\log_2(3)}. \end{aligned}$$

We only use product trees for computing h_I and $E_{i,J}$ which have leaves of degree $d = 1$ and $d = 2$, respectively, so we are only interested in the special cases

$$ProductTree_1(m) = m^{\log_2(3)} - m, \qquad ProductTree_2(m) = 3m^{\log_2(3)} - 3m.$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

Reciprocal tree. Computing the resultants requires the aid of a reciprocal tree, which is built from the root down using the product tree. If a node in the product tree contains a polynomial F of degree m, then the corresponding node in the reciprocal tree contains $(\mathfrak{F}, c)$, where $\mathfrak{F}$ is a polynomial of degree m and c is a constant such that $rev_m(F) \cdot \mathfrak{F} = c \mod x^m$. Here, $rev_m(\cdot)$ denotes the polynomial with the list of coefficients in reverse order: if $F = \sum_{i=0}^{m} F_i x^i$, then $rev_m(F) = \sum_{i=0}^{m} F_{m-i} x^i$.

We only use the case for the product tree of a polynomial f that has b leaves of degree 1. For the root of the tree, we need to know the cost of obtaining from scratch $rev_b(f)^{-1} \mod x^b$, where f has degree b. As pointed out in [23, Section 6.4], if we have already computed a reciprocal $(\mathfrak{f}_0, c_0)$ of half the degree (that is, $rev_b(f) \cdot \mathfrak{f}_0 = c_0 \mod x^{b/2}$), then the inverse modulus x^b can be obtained as $(\mathfrak{f}, c)$ with

$$\mathfrak{f} = c_0\mathfrak{f}_0 - (rev_b(f) \cdot \mathfrak{f}_0 - c_0)\mathfrak{f}_0 \mod x^b,$$

$$c = c_0^2.$$

These equations can be evaluated at the cost of $b/2 + 1$ field multiplications (for multiplying f_0 by the scalar c_0), plus the cost of 2 polynomial multiplications modulus $x^{b/2}$ (because the term in parenthesis is known to have null coefficients for all powers less than $b/2$), plus one squaring. This leads to the recursive formula for the cost of the reciprocal

$$Reciprocal(b) = Reciprocal(b/2) + b/2 + 1 + 2 \times PolyMul(b/2) + 1$$

$$= Reciprocal(b/2) + b/2 + 2(b/2)^{log_2(3)} + 2.$$

The base case (finding a multiple of the reciprocal of a constant) is trivial, so we obtain

$$Reciprocal(b) = \sum_{i=1}^{\log_2(b)} \left(\frac{b}{2^i} + 2\left(\frac{b}{2^i}\right)^{\log_2(3)} + 2 \right) = \qquad b^{\log_2(3)} + b + 2\log_2(b) - 2.$$

The child nodes in the reciprocal tree can then be computed from the root. Let f_1, f_2 be polynomials of degree d corresponding to sibling nodes of the product tree, and assume that we have already computed a reciprocal for their parent node $F = f_1 \cdot f_2$. That is, we already have $(\mathfrak{F}, C)$ such that $rev_{2d}(F) \cdot \mathfrak{F} = C \mod x^{2d}$. It follows that $rev_d(f_1) \cdot (rev_d(f_2) \cdot \mathfrak{F}) = C \mod x^d$, so we can compute a reciprocal for f_1 via the modular multiplication $rev_d(f_2) \cdot \mathfrak{F} \mod x^d$. The total cost of the reciprocal tree is then

$$ReciprocalTree(b) = Reciprocal(b) + \sum_{i=1}^{\log_2(b)} 2^i \times PolyMul(b/2^i)$$

$$= b^{log_2(3)} + b + 2\log_2(b) - 2 + \sum_{i=1}^{\log_2(b)} 2^i \left(\frac{b}{2^i}\right)^{\log_2(3)}$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 January 2024 doi:10.20944/preprints202401.1366.v1

$$=3b^{\log_2(3)} - b + 2\log_2(b) - 2.$$

Residue tree. Recall that we can compute the resultant $Res_Z(f,g)$ for monic f as

$$Res_Z(f,g) = \prod_i g(x_i) = \prod_i (g \mod f_i),$$

where x_i are the roots of f and $f_i = Z - x_i$ are its linear factors. In the context of 2 and 3, (resp. the parallel versions 5 and 6), f is a polynomial of degree b corresponding to h_I (resp. h_{I_t}) for which we have already computed the product tree with leaves f_i as well as the reciprocal tree, and g is a polynomial of degree $2b$ corresponding to $E_{i,J}$ for which we have computed the product tree with leaves of degree 2. We obtain the values $g \mod f_i$ by building a residue tree from the root down. Each node of the residue tree contains $R := cF \mod G$, where F and G are the polynomials in the corresponding nodes of the product trees of f and g, respectively, and c is the constant of the reciprocal tree.

As pointed out in [34, Section 17.4], the computation of $F \mod G$ for $deg(G) = 2deg(F) = 2d$ can be achieved with the aid of $(\mathfrak{F}, c)$ from the reciprocal tree via

$$F \mod G = G - rev_d(\mathfrak{F} \cdot rev_{2d}(G) \mod x^d) \cdot F \mod x^d,$$

at the cost of an additional two polynomial multiplications modulus x^d, so the total cost of the tree (including the cost to multiply together all of the leaves) is

$$ResidueTree(b) = b - 1 + 2\sum_{i=0}^{\log_2(b)} 2^i \times PolyMul(b/2^i)$$

$$= b - 1 + 2\sum_{i=0}^{\log_2(b)} 2^i \left(\frac{b}{2^i}\right)^{\log_2(3)}$$

$$= 6b^{\log_2(3)} - 3b - 1.$$

Note that the product of all the leaves gives us a multiple of the resultant (the multiple being the product of all the c constants at the leaves of the reciprocal tree), but these cancel out when taking ratios in the actual algorithms.

References

1. De Feo, L.; Jao, D. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. Post-Quantum Cryptography; Yang, B.Y., Ed.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2011; pp. 19–34.
2. De Feo, L.; Jao, D.; Plût, J. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Mathematical Cryptology* **2014**, *8*, 209–247.
3. SIKE - Supersingular Isogeny Key Encapsulation. Available online: https://sike.org/, 2023. Accessed: 6-12-2023.
4. Castryck, W.; Decru, T. An Efficient key recovery attack on SIDH. Advances in Cryptology – EUROCRYPT 2023; Hazay, C.; Stam, M., Eds.; Springer Nature Switzerland: Cham, 2023; pp. 423–447.

5. Maino, L.; Martindale, C.; Panny, L.; Pope, G.; Wesolowski, B. A direct key recovery attack on SIDH. Advances in Cryptology – EUROCRYPT 2023; Hazay, C.; Stam, M., Eds.; Springer Nature Switzerland: Cham, 2023; pp. 448–471.
6. Robert, D. Breaking SIDH in polynomial time. Advances in Cryptology – EUROCRYPT 2023; Hazay, C.; Stam, M., Eds.; Springer Nature Switzerland: Cham, 2023; pp. 472–503.
7. Castryck, W.; Lange, T.; Martindale, C.; Panny, L.; Renes, J. CSIDH: An efficient post-quantum commutative group action. Advances in Cryptology – ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part III; Springer-Verlag: Berlin, Heidelberg, 2018; p. 395–427.
8. De Feo, L.; Galbraith, S.D. SeaSign: compact isogeny signatures from class group actions. Advances in Cryptology – EUROCRYPT 2019; Ishai, Y.; Rijmen, V., Eds.; Springer International Publishing: Cham, 2019; pp. 759–789.
9. Beullens, W.; Kleinjung, T.; Vercauteren, F. CSI-FiSh: Efficient isogeny based signatures through class group computations. Advances in Cryptology – ASIACRYPT 2019; Galbraith, S.D.; Moriai, S., Eds.; Springer International Publishing: Cham, 2019; pp. 227–247.
10. De Feo, L.; Kohel, D.; Leroux, A.; Petit, C.; Wesolowski, B. SQISign: Compact post-quantum signatures from quaternions and isogenies. Advances in Cryptology – ASIACRYPT 2020; Moriai, S.; Wang, H., Eds.; Springer International Publishing: Cham, 2020; pp. 64–93.
11. National Institute of Standards and Technology NIST. Available online: https://csrc.nist.gov/news/2023/additional-pqc-digital-signature-candidates, 2023. Accessed: 6-12-2023.
12. Chávez-Saab, J.; Chi-Domínguez, J.; Jaques, S.; Rodríguez-Henríquez, F. The SQALE of CSIDH: sublinear Vélu quantum-resistant isogeny action with low exponents. *J. Cryptogr. Eng.* **2022**, *12*, 349–368.
13. SQIsign: Algorithm specifications and supporting documentation. Available online: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/sqisign-spec-web.pdf, 2023. Accessed: 6-12-2023.
14. Basso, A.; Maino, L.; Pope, G. FESTA: Fast encryption from supersingular torsion attacks. Cryptology ePrint Archive, Paper 2023/660, 2023.
15. Nakagawa, K.; Onuki, H. QFESTA: Efficient algorithms and parameters for FESTA using quaternion algebras. Cryptology ePrint Archive, Report 2023/1468, 2023.
16. Moriya, T. IS-CUBE: An isogeny-based compact KEM using a boxed SIDH diagram. Cryptology ePrint Archive, Report 2023/1506, 2023.
17. Decru, T.; Maino, L.; Sanso, A. Towards a Quantum-resistant Weak Verifiable Delay Function. Progress in Cryptology - LATINCRYPT 2023 - 8th International Conference on Cryptology and Information Security in Latin America; Aly, A.; Tibouchi, M., Eds. Springer, 2023, Vol. 14168, *LNCS*, pp. 149–168.
18. Leroux, A. Verifiable random function from the Deuring correspondence and higher dimensional isogenies. Cryptology ePrint Archive, Report 2023/1251, 2023.
19. Vélu, J. Isogénies entre courbes elliptiques. *Comptes-Rendus de l'Académie des Sciences, Série I* **1971**, *273*, 238–241.
20. Kohel, D.R. Endomorphism rings of elliptic curves over finite fields. PhD thesis, University of California at Berkeley, The address of the publisher, 1996. Available at:http://iml.univ-mrs.fr/~kohel/pub/thesis.pdf.
21. Washington, L. *Elliptic curves: number theory and cryptography, Second Edition*, 2 ed.; Chapman & Hall/CRC, 2008.
22. Bernstein, D.J.; Feo, L.D.; Leroux, A.; Smith, B. Faster computation of isogenies of large prime degree. In: ANTS XIV. The Open Book Series, vol. 4(1), pp. 39–55 (2020), 2020.
23. Adj, G.; Chi-Domínguez, J.; Rodríguez-Henríquez, F. Karatsuba-based square-root Vélu's formulas applied to two isogeny-based protocols. *Journal of Cryptographic Engineering* **2022**.
24. Cheng, H.; Fotiadis, G.; Großschädl, J.; Ryan, P.Y.A.; Rønne, P.B. Batching CSIDH group actions using AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2021**, *2021*, 618–649.
25. Orisaka, G.; López-Hernández, J.C.; Aranha, D.F. Finite field arithmetic using AVX-512 for isogeny-based cryptography. 18th Brazilian Symposium on Information and Computer Systems Security (SBSeg), 2018, pp. 49–56.
26. Cheng, H.; Fotiadis, G.; Großschädl, J.; Ryan, P.Y.A. Highly Vectorized SIKE for AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2022**, *2022*, 41–68.

27. Phalakarn, K.; Suppakitpaisarn, V.; Rodríguez-Henríquez, F.; Hasan, M.A. Vectorized and parallel computation of large smooth-Degree isogenies using precedence-constrained scheduling. Cryptology ePrint Archive, Paper 2023/885, 2023.
28. Phalakarn, K.; Suppakitpaisarn, V.; Hasan, M.A. Speeding-up parallel computation of large smooth-degree isogeny using precedence-constrained scheduling. Information Security and Privacy; Nguyen, K.; Yang, G.; Guo, F.; Susilo, W., Eds.; Springer International Publishing: Cham, 2022; pp. 309–331.
29. Kato, G.; Suzuki, K. Speeding up CSIDH using parallel computation of isogeny. 2020 7th International Conference on Advance Informatics: Concepts, Theory and Applications (ICAICTA), 2020, pp. 1–6.
30. Elkies, N.D.; others. Elliptic and modular curves over finite fields and related computational issues. *AMS IP STUDIES IN ADVANCED MATHEMATICS* **1998**, *7*, 21–76.
31. Costello, C.; Hisil, H. A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies. Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II; Takagi, T.; Peyrin, T., Eds. Springer, 2017, Vol. 10625, *Lecture Notes in Computer Science*, pp. 303–329.
32. Costello, C. B-SIDH: Supersingular Isogeny Diffie-Hellman Using Twisted Torsion. Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part II; Moriai, S.; Wang, H., Eds. Springer, 2020, Vol. 12492, *Lecture Notes in Computer Science*, pp. 440–463.
33. Karatsuba, A.; Ofman, Y. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady* **1962**, *7*, 595.
34. Bernstein, D.J., Fast multiplication and its applications. In *Algorithmic Number Theory*; Buhler, J.; Stevenhagen, P., Eds.; Mathematical Sciences Research Institute Publications, Cambridge University Press: United Kingdom, 2008; pp. 325–384.