IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–23. DOI:XXXXXXXX

Vectorized and Parallel Computation of Large Smooth-Degree Isogenies using Precedence-Constrained Scheduling

Kittiphon Phalakarn[1], Vorapong Suppakitpaisarn[2], Francisco Rodríguez-Henríquez[3,4] and M. Anwar Hasan[1]

[1] University of Waterloo, Waterloo, Canada, {kphalakarn,ahasan}@uwaterloo.ca
[2] The University of Tokyo, Tokyo, Japan, vorapong@is.s.u-tokyo.ac.jp
[3] CINVESTAV-IPN, Mexico City, Mexico, francisco@cs.cinvestav.mx
[4] Technology Innovation Institute, Abu Dhabi, UAE, francisco.rodriguez@tii.ae

Abstract. *Strategies* and their *evaluations* play important roles in speeding up the computation of large smooth-degree isogenies. The concept of *optimal strategies* for such computation was introduced by De Feo et al., and virtually all implementations of isogeny-based protocols have adopted this approach, which is provably optimal for single-core platforms. In spite of its inherent sequential nature, several recent works have studied ways of speeding up this isogeny computation by exploiting the rich parallelism available in vectorized and multi-core platforms. One obstacle to taking full advantage of this parallelism, however, is that De Feo et al.'s strategies are not necessarily optimal in multi-core environments. To illustrate how the speed of vectorized and parallel isogeny computation can be improved at the strategy-level, we present two novel software implementations that utilize a state-of-the-art evaluation technique, called *precedence-constrained scheduling (PCS)*, presented by Phalakarn et al., with our proposed strategies crafted for these environments. Our first implementation relies only on the parallelism provided by multi-core processors. The second implementation targets multi-core processors supporting the latest generation of the Intel's Advanced Vector eXtensions (AVX) technology, commonly known as AVX-512IFMA instructions. To better handle the computational concurrency associated with PCS, we equip both implementations with extensive synchronization techniques. Our first implementation outperforms the implementation of Cervantes-Vázquez et al. by yielding up to 14.36% reduction in the execution time, when targeting platforms with two- to four-core processors. Our second implementation, equipped with four cores, achieves up to 34.05% reduction in the execution time compared to the single-core implementation of Cheng et al. of CHES 2022.

Keywords: Isogeny-based cryptography · Isogeny computation · Software optimization · Vectorization · Parallel computing · Precedence-constrained scheduling

1 Introduction

Isogeny-based cryptography is much newer than other post-quantum cryptographic schemes, specifically, those based on codes and lattices. Some of the isogeny-based cryptographic schemes include hash functions [CLG09, DPB17], the Supersingular Isogeny Diffie-Hellman (SIDH) key exchange [JD11], the Supersingular Isogeny Key Encapsulation (SIKE) mechanism [JAC+20], and the Commutative SIDH (CSIDH) [CLM+18]. Recently, other isogeny-based schemes have been proposed, notably Verifiable Delay Functions (VDFs) [DMPS19, CSRT22] and the Short Quaternion and Isogeny Signature (SQISign) scheme

[DKL$^+$20]. We note that a key-recovery attack exploiting the auxiliary elliptic-curve points of SIDH/SIKE has recently rendered SIDH/SIKE completely insecure [CD22, MM22, Rob22], but there is no known way to apply similar attacks to the general isogeny problem.

The computation to find the curve and point images of isogenies, required by many cryptosystems, is time-consuming. There have been various proposals (see below) to speed-up the computation to obtain low-latency implementations of those protocols. Also, the speed-up is of interest for VDFs. A VDF is a function that cannot be computed in less time than a prescribed delay. Thus, the function must crucially be as sequential as possible, in the sense that there should not exist any effective parallelization technique that yields a significant acceleration in its computation. In this work, we push the limits of the amount of effective parallelization obtained from isogeny computations, which can be useful for a parameter selection in isogeny-based VDFs. Our focus is on the computation of large smooth-degree isogenies with degree ℓ^e, where ℓ is typically a small prime.

It is known that the best way of performing this task is through the computation of a sequence of degree-ℓ isogenies using Vélu-like formulas and point multiplications by $[\ell]$. The first work which considered this problem is by De Feo et al. [FJP14]. The authors started with an abstraction of the computation called a *strategy* and associated a *cost* with it. The cost of a strategy theoretically represents the execution time of the isogeny computation corresponding to that strategy when it is implemented. In their paper, a dynamic programming equation for mathematically constructing a strategy with the least cost, called an *optimal strategy*, is proposed. These optimal strategies are then utilized in many implementations to reduce execution times [KJA$^+$16, Mic17]. Apart from this, several techniques were introduced to speed-up isogeny computation taking into account various arithmetic aspects of the underlying field [CLN16, ALJK18, FLOR18, SLLH18].

In order to speed-up the computation, one can also adopt a technology especially designed for exploiting paralellism, such as vector instructions of Intel's Advanced Vector eXtensions (AVX). By these special instructions, multiple operations can be performed simultaneously on vectors. For the latest generation of AVX, called AVX-512, each vector (consisting of an array of data) is of length 512 bits and can be operated as eight 64-bit elements, meaning that eight 64-bit operations can be performed within a similar time as a single 64-bit operation. The advantages of AVX-512 have been exploited to speed-up the isogeny computation by a few works [KG19, CFGR22]. In [CFGR22], the authors proposed optimizations in several layers, including base-field arithmetic, extension-field arithmetic, elliptic curve arithmetic, and isogeny computation. Combining all those techniques, the execution time of their implementation is 2.40 times faster compared to that of [Mic17].

To further improve the speed of the isogeny computation, many researchers [KAK16, KAK$^+$20, COR22] turned to multi-core platforms, on which multiple operations can be performed simultaneously on different cores. We note that the use of vector instructions is somewhat similar to the multi-core setting, but in the former, the same instruction is performed on all vector elements. As the execution environment changes, strategies and the cost function have to be revised accordingly. The earliest work that analyzes strategies and the cost functions specifically for the multi-core setting is due to Hutchinson and Karabina [HK18]. Their main contributions are a formalization of a parallel isogeny computation on multi-core platforms called *per-curve parallel (PCP)* and a dynamic programming algorithm constructing optimal strategies under this PCP parallel computation. The experimental results show that, in the multi-core setting, optimal strategies under PCP lead to lower costs compared to original optimal strategies of [FJP14] designed for serial computation. This implies that serial optimal strategies of [FJP14] are not necessarily optimal in the multi-core environment. Looking at the experimental results, the theoretical costs of optimal strategies under PCP are up to 24%, 40%, and 51% cheaper than the costs of optimal strategies of [FJP14] when the number of cores is two, four, and eight, respectively. And when implementing the computation on a three-core platform using the techniques of [HK18] along with other optimizations, Cervantes-Vázquez et al. [COR22] could achieve more than 35% speed-up in the execution time compared to the serial implementation. These results clearly show an impact on the speeding-up of isogeny computation for multi-core platforms at the strategy-level.

Although speeding-up at the strategy-level is important, there is no implementation that attempts to do so beyond PCP. To demonstrate how the computation can be further sped-up at the strategy-level for vectorized and parallel software implementation, we consider the recent work of Phalakarn et al. [PSH22] which has proposed the *precedence-constrained scheduling (PCS)* technique to evaluate strategies. In that work, the isogeny computation is considered to have two orthogonal axes: strategy *construction* and strategy *evaluation*, i.e., the order in which operations are performed affects the strategy cost. The PCS technique is an evaluation scheme which is shown to lead to lower cost compared to PCP for the same strategy. The theoretical costs of strategies under PCS are up to 20% less than those under PCP. Although the theoretical strategy costs under PCS are promising, there is no work, to the best of our knowledge, that applies the PCS technique to practical software or hardware implementations of isogeny-based protocols.

Contributions. This work presents two software implementations of the degree-ℓ^e isogeny computation, which aim to further speed-up this operation at the strategy-level by exploiting parallelism and the PCS technique. Since [PSH22] did not consider any implementation aspects, we also provide analyses and modifications on how to effectively apply the PCS technique to the unique execution environment of each implementation. For comparison purposes, the implementations are based on the SIKEp751 parameter set of SIKE. Both of our implementations are available at `https://github.com/kittiphonp/PCS`.

The first implementation is designed for multi-core processors without AVX-512IFMA instructions. We consider processors with two to four cores, and tailor strategies and schedulings under PCS for each case. We use OpenMP for multi-threading with extensive techniques to handle synchronization among cores. Moreover, we include an optimization of [COR22] and modify the scheduling algorithm of [PSH22] to take advantage of this optimization. As a result, we are able to obtain speed-ups of up to 12.53%, 12.57%, and 14.36% when compared to [COR22] using two, three, and four cores, respectively.

The second implementation is designed for multi-core processors which support AVX-512IFMA. To our best knowledge, this work is the first to combine vectorization with multi-core processors for isogeny computation. We start by improving PCP to suit our specific execution environment. Then, PCS is modified and applied to the strategies crafted for AVX-512 implementations. We also use the synchronization technique as in the first implementation. Under this environment, PCS is very effective and can reduce a large amount of the execution time over PCP. A speed-up of up to 34.05% is achieved by our four-core implementation when compared to the single-core implementation of [CFGR22].

Generalization. The proposed implementations have a potential to be adapted for the computation of degree-$\ell_1\ell_2\cdots\ell_n$ isogenies, which are used in some isogeny-based protocols such as CSIDH. The only strategy-level difference is that in this case the isogeny computation is more complex. In particular, the cost of computing an ℓ_i-isogeny differs from that of an ℓ_j-isogeny. This is also true for point multiplication by $[\ell_i]$ and $[\ell_j]$.

To handle these differences, we require some changes to both strategy construction and evaluation. For strategy construction, there are already some works that have studied optimal strategies for the isogeny computation [HLKA20, CR22]. Although both approaches are designed for serial implementation, they can be extended to vectorized and parallel implementation in the same manner. For strategy evaluation, the scheduling algorithm used by the PCS technique needs to be applicable with tasks of varying length. Example of such algorithms are [Gra66, CS99]. Tables 1 and 2 summarize and compare strategy construction and evaluation techniques for both isogeny computations.

Table 1: Strategy construction and evaluation techniques for computing isogenies of degree ℓ^e in various settings.

Settings	**Strategy Construction**	**Strategy Evaluation**
Single-core	Optimal strategies [FJP14]	Sequential
Multi-core	PCP [HK18, COR22]	Modified PCS (Section 3.1)
AVX & Multi-core	Modified PCP (Section 4.1)	Modified PCS (Section 4.2)

Table 2: Strategy construction and evaluation techniques for computing isogenies of degree $\ell_1\ell_2\cdots\ell_n$ in various settings.

Settings	**Strategy Construction**	**Strategy Evaluation**
Single-core	Optimal strategies [HLKA20, CR22]	Sequential
Multi-core	PCP	Modified PCS using [Gra66, CS99]
AVX & Multi-core	Modified PCP	

2 Preliminaries

In this section, we give a brief overview of methods to compute large smooth-degree isogenies, strategies for isogeny computation, the per-curve parallel (PCP) technique, the precedence-constrained scheduling (PCS) technique, and the AVX-512 instructions.

2.1 Large Smooth-Degree Isogeny Computation and Strategies

Let E and E' be elliptic curves over a field F. An *isogeny* from E to E' is a surjective morphism with a finite kernel. When specifying an elliptic curve E over F and a point $R \in E(F)$, one can compute the unique isogeny $\phi : E \to E/\langle R\rangle$ satisfying $\ker\phi = \langle R\rangle$ using Vélu's [Vél71] or $\sqrt{\text{é}}$lu's [BDFLS20] formulas. The degree of ϕ is equal to the order of R.

Many isogeny-based cryptosystems require a computation of large smooth-degree isogeny ϕ and the image curve E' given the starting curve E and the kernel generator R. We describe in this section how this computation can be performed effectively.

Let the order of R be ℓ^e, where ℓ is a small prime. Instead of directly computing the degree-ℓ^e isogeny, it is suggested by [FJP14] to break down the computation into a chain of e degree-ℓ isogenies:

$$\phi : E = E_0 \xrightarrow{\phi_0} E_1 \xrightarrow{\phi_1} E_2 \xrightarrow{\phi_2} \cdots \xrightarrow{\phi_{e-2}} E_{e-1} \xrightarrow{\phi_{e-1}} E_e = E'.$$

Specifically, for $0 \le i < e$, ϕ_i is an ℓ-isogeny from E_i to E_{i+1} with a kernel generated by $[\ell^{e-i-1}]R_i$, where $R_0 = R$ and $R_{i+1} = \phi_i(R_i)$. The straightforward way of implementing this is to first perform $e-i-1$ point multiplications by $[\ell]$ to R_i in order to obtain the kernel generator $[\ell^{e-i-1}]R_i$, next generate (ϕ_i, E_{i+1}) using Vélu's or $\sqrt{\text{é}}$lu's formulas from $(E_i, [\ell^{e-i-1}]R_i)$, and then compute $R_{i+1} = \phi_i(R_i)$. This can be done in a different way by simply noting that $\phi([\ell]R) = [\ell]\phi(R)$. De Feo et al. [FJP14] captured how an ℓ^e-isogeny can be computed using *strategies* which we define below.

Definition 1 (Strategy)**.** The directed graph showing all possible operations for computing ℓ^e-isogeny is defined as $\mathcal{T}_e = (\mathcal{V}_e, \mathcal{E}_e)$ where $\mathcal{V}_e = \{(i,j) : 0 \le i,j < e \wedge i+j < e\}$ and $\mathcal{E}_e = \{\langle(i,j),(i+1,j)\rangle, \langle(i,j),(i,j+1)\rangle : (i,j) \in \mathcal{V}_e \wedge i+j \ne e-1\}$.

A *strategy* $\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S})$ for computing ℓ^e-isogeny is a subgraph of $\mathcal{T}_e$ such that there is a path from $(0,0)$ to $(i, e-i-1)$ for $0 \le i < e$ in $\mathcal{S}$.

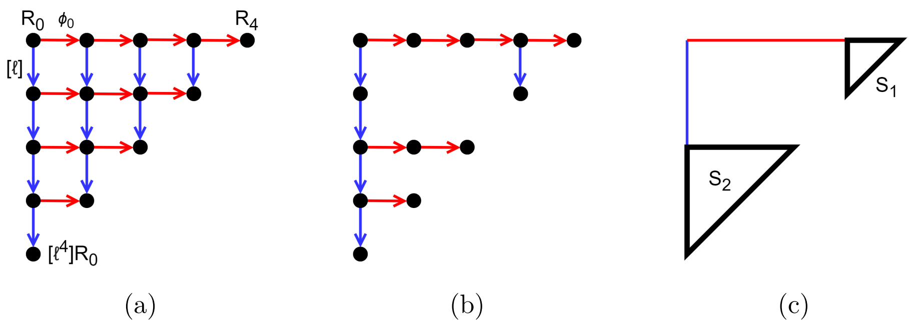

Figure 1: (a) The directed graph $\mathcal{T}_e$ for $e = 5$, (b) An example of a strategy $\mathcal{S}$ for $e = 5$, and (c) An abstraction of $\mathcal{S}$ using smaller strategies $\mathcal{S}_1$ and $\mathcal{S}_2$.

Examples of the graph $\mathcal{T}_e$ and a strategy $\mathcal{S}$ for $e = 5$ are depicted in Figure 1(a) and 1(b), respectively. In brief, a vertex (i, j) in the i-th column and j-th row represents $[\ell^j]R_i$, a left-to-right edge $\langle(i, j), (i+1, j)\rangle$ represents the computation of $[\ell^j]R_{i+1} = \phi_i([\ell^j]R_i)$, and a top-to-bottom edge $\langle(i, j), (i, j+1)\rangle$ represents the computation of $[\ell^{j+1}]R_i = [\ell]([\ell^j]R_i)$. Hence, all edges on a path from $(0, 0)$ to $(i, e-i-1)$ form a computation of the kernel generator for ϕ_i. Therefore, all strategies provide valid ways to compute ℓ^e-isogeny, but their usefulnesses are different. The paper [FJP14] associates a cost to each strategy, reflecting its execution time when implemented.

In the single-core setting, we utilize a strategy $\mathcal{S}$ as follows: for each $0 \leq i < e$, (i) perform point multiplications indicated by an edge $\langle(i, j), (i, j+1)\rangle$ in $\mathcal{S}$, (ii) generate (ϕ_i, E_{i+1}) from $(E_i, [\ell^{e-i-1}]R_i)$, and (iii) perform isogeny evaluations indicated by an edge $\langle(i, j), (i+1, j)\rangle$ in $\mathcal{S}$. Let $\mathcal{E}_{\mathcal{S},\text{mul}}$ be the set of all edges of the form $\langle(i, j), (i, j+1)\rangle$ in $\mathcal{S}$ and $\mathcal{E}_{\mathcal{S},\text{iso}}$ be the set of all edges of the form $\langle(i, j), (i+1, j)\rangle$ in $\mathcal{S}$. The cost function of a strategy proposed by [FJP14] is $\mathcal{C}_1(\mathcal{S}) = \#\mathcal{E}_{\mathcal{S},\text{mul}} \cdot c_{\text{mul}} + \#\mathcal{E}_{\mathcal{S},\text{iso}} \cdot c_{\text{iso}}$, where c_{mul} and c_{iso} denote the costs of performing one point multiplication by $[\ell]$ and one isogeny evaluation, respectively. In addition, De Feo et al. presented a dynamic programming equation which computes the lowest cost of all possible strategies for computing degree-ℓ^e isogeny, given $(e, c_{\text{mul}}, c_{\text{iso}})$: $\mathcal{C}_1^*(e) = \min_{0<i<e}\{\mathcal{C}_1^*(i) + \mathcal{C}_1^*(e-i) + (e-i) \cdot c_{\text{mul}} + i \cdot c_{\text{iso}}\}$. Strategies with the cost of $\mathcal{C}_1^*(e)$ are then called *optimal strategies*.

We explain the idea behind the equation of $\mathcal{C}_1^*(e)$ as follows. It is proved in [FJP14] that any optimal strategy must be constructed from two smaller optimal strategies $\mathcal{S}_1$ and $\mathcal{S}_2$, as shown in Figure 1(c). These two smaller optimal strategies are combined by a sequence of point multiplications and a sequence of isogeny evaluations. Suppose $\mathcal{S}_1$ covers the last $e - i$ columns and $\mathcal{S}_2$ covers the first i columns. Since $\mathcal{S}_1$ and $\mathcal{S}_2$ are both optimal, their costs are $\mathcal{C}_1^*(e-i)$ and $\mathcal{C}_1^*(i)$, respectively. Including the cost of $(e-i) \cdot c_{\text{mul}}$ for the sequence of point multiplications and the cost of $i \cdot c_{\text{iso}}$ for the sequence of isogeny evaluations that connect $\mathcal{S}_1$ and $\mathcal{S}_2$, we obtain the equation for $\mathcal{C}_1^*(e)$. Hence, this equation also indicates how one can construct an optimal strategy.

2.2 Per-Curve Parallel (PCP) Technique

When working on multi-core platforms, we are provided with a processor with K cores and, hence, up to K operations can be performed simultaneously. However, it is not simple to fully utilize all cores for degree-ℓ^e isogeny since there are dependencies among operations. Hutchinson and Karabina [HK18] were the first to analyze this setting and proposed an original computation model. Under their *per-curve parallel (PCP)* computation, a strategy $\mathcal{S}$ is utilized as follows: for each $0 \leq i < e$, (i) *serially* perform point multiplications indicated by $\langle(i, j), (i, j+1)\rangle$, (ii) generate (ϕ_i, E_{i+1}) from $(E_i, [\ell^{e-i-1}]R_i)$, and (iii) perform isogeny evaluations indicated by $\langle(i,j),(i+1,j)\rangle$, *up to K evaluations at a time.* The assumption of PCP is that K evaluations on K cores take time equal to one evaluation.

As the computation model changes, the cost function must be revised. Let $\mathcal{E}_{\mathcal{S},\mathrm{iso},i}$ be the set of all edges of the form $\langle(i,j),(i+1,j)\rangle$ in $\mathcal{S}$, i.e., the set of evaluations using ϕ_i. The cost function of a strategy under PCP is proposed as $\mathfrak{C}_K^{\mathrm{PCP}}(\mathcal{S}) = \#\mathcal{E}_{\mathcal{S},\mathrm{mul}} \cdot c_{\mathrm{mul}} + \sum_{0\le i<e}(\lceil \#\mathcal{E}_{\mathcal{S},\mathrm{iso},i}/K\rceil \cdot c_{\mathrm{iso}})$. Although the cost function looks more complicated than that for the single-core setting, Hutchinson and Karabina were able to construct a dynamic programming equation which yields the least cost of all strategies under PCP, when $(e, c_{\mathrm{mul}}, c_{\mathrm{iso}}, K)$ is fixed:

$$\mathfrak{C}_K^{\mathrm{PCP}^*}(e,k) = \begin{cases} 0 & \text{if } e=1, \\ \mathfrak{C}_K^{\mathrm{PCP}^*}(e,K) + (e-1)\cdot c_{\mathrm{iso}} & \text{if } e>1 \text{ and } k=0, \\ \min\limits_{0<i<e}\{\mathfrak{C}_K^{\mathrm{PCP}^*}(i,k-1) + \mathfrak{C}_K^{\mathrm{PCP}^*}(e-i,k) + (e-i)\cdot c_{\mathrm{mul}} + c_{\mathrm{iso}}\} & \text{otherwise.} \end{cases}$$

The least cost of all strategies for ℓ^e-isogeny under PCP is $\mathfrak{C}_K^{\mathrm{PCP}^*}(e,K)$ with $k=K$. A strategy having this cost is thus called *optimal* under PCP.

The notion $\mathfrak{C}_K^{\mathrm{PCP}^*}(e,k)$ denotes a subproblem of calculating the least cost of strategies for ℓ^e-isogeny under PCP when k out of K cores are available for the first isogeny evaluations in step (iii) above. In other words, for $\mathfrak{C}_K^{\mathrm{PCP}^*}(e,k)$, we can evaluate up to k points when we first perform step (iii) and then, if there are still points to be evaluated, up to K points in the following iterations. We give an example below to help understanding the notation.

Example 1. Suppose we are to evaluate the strategy in Figure 1(b) when $k=2$ out of $K=3$ cores are available for the first isogeny evaluations in step (iii). For ϕ_0, we need to perform three isogeny evaluations. Since we can perform only $k=2$ evaluations in the first iteration, we have to perform the remaining one evaluation in the second iteration, in which we can evaluate up to K points. Two iterations take a time of $2\cdot c_{\mathrm{iso}}$. On the other hand, for ϕ_1, we need to perform two isogeny evaluations and we can perform all of them in the first iteration. Thus, this takes a time of c_{iso}.

The idea behind this equation is that an optimal strategy under PCP can again be decomposed into two smaller optimal strategies. The third case of the equation selects an optimal decomposition among all possible choices. However, one core is spent on isogeny evaluations which connect both parts (shown in Figure 2 as a dotted line). Therefore, $\mathfrak{C}_K^{\mathrm{PCP}^*}(e,k)$ is decomposed to $\mathcal{S}_1$ and $\mathcal{S}_2$ with costs of $\mathfrak{C}_K^{\mathrm{PCP}^*}(e-i,k)$ and $\mathfrak{C}_K^{\mathrm{PCP}^*}(i,k-1)$, respectively. In the second case where $e>1$ and $k=0$, we can evaluate no points when we first perform step (iii). This means that all cores are occupied. We can convert those working cores into a cost of $(e-1)\cdot c_{\mathrm{iso}}$ and start step (iii) fresh with K cores. Hence, we have $\mathfrak{C}_K^{\mathrm{PCP}^*}(e,0) = \mathfrak{C}_K^{\mathrm{PCP}^*}(e,K) + (e-1)\cdot c_{\mathrm{iso}}$.

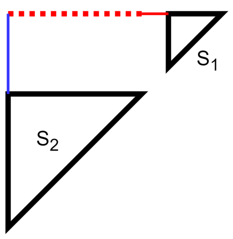

Figure 2: Computing the cost of strategies using PCP: Strategy $\mathcal{S}$ is decomposed into $\mathcal{S}_1$ with a cost of $\mathfrak{C}_K^{\mathrm{PCP}^*}(e-i,k)$ and $\mathcal{S}_2$ with a cost of $\mathfrak{C}_K^{\mathrm{PCP}^*}(i,k-1)$.

The work of Hutchinson and Karabina actually presented another computation model called *consecutive-curve parallel (CCP)*. Briefly, CCP allows point multiplications and isogeny evaluations to be performed simultaneously, but only for operations on the same curve E_i or the next curve E_{i+1}. This flexibility improves CCP from PCP to some extent. We refer the interested readers to [HK18] for details of CCP.

2.3 Precedence-Constrained Scheduling (PCS) Technique

Although PCP and CCP provide faster implementations in the multi-core setting when compared to the use of optimal strategies of [FJP14], the formers still do not fully utilize all cores as much as possible. This issue was analyzed by Phalakarn et al. [PSH22] who then proposed the *precedence-constrained scheduling (PCS)* technique as described below.

The PCS technique, shown in Algorithm 1, formalizes ℓ^e-isogeny computation as a scheduling problem with precedences. In detail, the computation $[\ell^{j+1}]R_i = [\ell]([\ell^j]R_i)$ requires the value of $[\ell^j]R_i$, and the computation $[\ell^j]R_{i+1} = \phi_i([\ell^j]R_i)$ requires the value of $[\ell^j]R_i$ and also the kernel generator $[\ell^{e-i-1}]R_i$ of ϕ_i. These precedences are described using a task dependency graph, which can be constructed from an input strategy (Lines 3–7). This task dependency graph is then used to schedule operations to processor's cores. The work of [PSH22] applied to the task dependency graph two scheduling algorithms, Hu's [Hu61] and Coffman-Graham's [CJG72] algorithms, to obtain a scheduling (Lines 8 and 14–16). In short, a scheduling is a sequence $\mathcal{S} = \langle S_1, \cdots, S_n \rangle$ where S_t is a set of points to be computed in iteration t and $\#S_t$ is no more than the number of processor's cores. Although both algorithms do not guarantee to provide scheduling with earliest finish time, it was experimentally shown that both provided good approximations. Finally, the cost of the scheduling for the given strategy is calculated (Lines 17–21). The cost is denoted by $\mathcal{C}_K^{\mathrm{Hu}}(\mathcal{S})$ or $\mathcal{C}_K^{\mathrm{CG}}(\mathcal{S})$ depending on the scheduling algorithm used.

Figure 3 below provides an example of a comparison between applying PCP and PCS to the same strategy. Suppose $K = 2$ and $c_{\mathrm{mul}} = c_{\mathrm{iso}}$. When using PCP, the steps that each operation performs are shown as in Figure 3(a) and the last operation is done in Step 10. The same number means that operations are performed simultaneously in the same step. When using PCS, we first obtain the task dependency graph of the strategy as in Figure 3(b). Applying a scheduling algorithm to it, we get the result as in Figure 3(c) where the last operation is done in Step 9.

Unlike the single-core setting and PCP, an optimal strategy under PCS has yet to be discovered due to the complexity of the computation model. In [PSH22], the authors used optimal strategies under PCP as inputs to the algorithm. Those optimal strategies are randomly constructed by the dynamic programming equation $\mathcal{C}_K^{\mathrm{PCP}^*}$. After constructing sufficient amount of strategies, the strategy with the lowest cost under PCS found until certain fixed number of iterations can be used for the implementation.

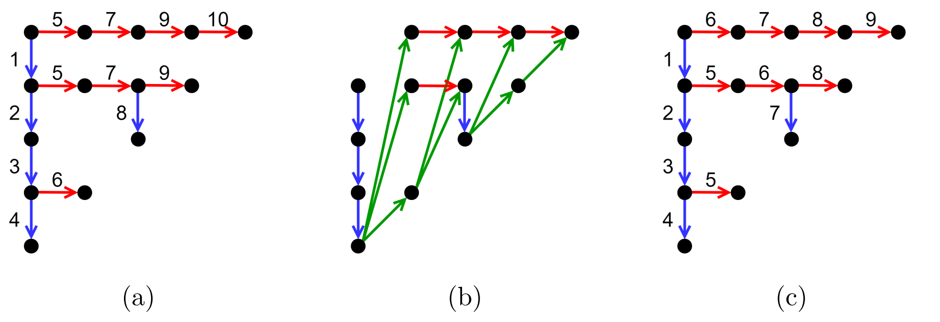

Figure 3: (a) A strategy with the steps shown when using PCP, (b) The task dependency graph of the strategy, and (c) The same strategy with the steps shown when using PCS.

Algorithm 1: Precedence-Constrained Scheduling (PCS) Technique

```
1  $\mathcal{C}_K^{\mathrm{Hu/CG}}(\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S}))$ :
2      $\mathcal{E}_\mathcal{S}^* \leftarrow \mathcal{E}_\mathcal{S}$
3      for $\langle(i,j),(i+1,j)\rangle \in \mathcal{S}$ do
4          $\mathcal{E}_\mathcal{S} \leftarrow \mathcal{E}_\mathcal{S} \cup \{\langle(i,e-i-1),(i+1,j)\rangle\}$
5      $\mathcal{E}_\mathcal{S} \leftarrow \mathcal{E}_\mathcal{S} \setminus \{\langle(0,0),(0,1)\rangle, \langle(0,0),(1,0)\rangle\}$
6      $\mathcal{V}_\mathcal{S} \leftarrow \mathcal{V}_\mathcal{S} \setminus \{(0,0)\}$
7      Remove transitive edges from $\mathcal{E}_\mathcal{S}$
8      Label all vertices $v \in \mathcal{S}$ with $\mathcal{L}(v)$ using Hu's or Coffman-Graham's algorithms
9      $\mathbb{S} \leftarrow \langle\rangle$
10     $t \leftarrow 0$
11     cost $\leftarrow 0$
12     while $\mathcal{V}_\mathcal{S} \neq \emptyset$ do
13         $t \leftarrow t+1$
14         $S_t \leftarrow \{K$ vertices in $\mathcal{S}$ with highest $\mathcal{L}(\cdot)$ and their in-degrees are $0\}$
15         Append $S_t$ to $\mathbb{S}$
16         Remove all vertices in $S_t$ and their out-going edges from $\mathcal{S}$
17         $\mathrm{cost}_t \leftarrow 0$
18         for $(i,j) \in S_t$ do
19             if $\langle(i,j-1),(i,j)\rangle \in \mathcal{E}_\mathcal{S}^*$ then $\mathrm{cost}_t \leftarrow \max\{\mathrm{cost}_t, c_{\mathrm{mul}}\}$
20             else $\mathrm{cost}_t \leftarrow \max\{\mathrm{cost}_t, c_{\mathrm{iso}}\}$
21         cost $\leftarrow$ cost $+ \mathrm{cost}_t$
22     return (cost, $\mathbb{S}$)
```

2.4 Intel's Advanced Vector eXtension AVX-512

The latest generation of Intel's Advanced Vector eXtensions (AVX), which is AVX-512, provides a way to vectorize and speed-up a software by using vectors of length 512 bits and vectorized instructions. One extension of AVX-512 used by [CFGR22] and this work is the Integer Fused Multiply-Add extension (IFMA or AVX-512IFMA) which is useful for software libraries requiring large integer arithmetic. As we are mainly interested in the strategy-level optimization, we briefly explain the high-level usage of AVX-512.

In [CFGR22] and our implementation, we consider 512-bit vectors as eight elements of 64 bits: $[a,b,c,d,e,f,g,h]$ where each variable is of size 64 bits. The AVX-512 instructions allow us to compute $[a,b,\dots,h] \oplus [a',b',\dots,h'] = [a \oplus a', b \oplus b', \dots, h \oplus h']$ within a similar time as $a \oplus a'$ for certain operations $\oplus$. Thus, we can consider AVX-512 as a form of parallel computation where all cores perform the same operation.

When the operand sizes are larger than 64 bits, they must be divided into 64-bit blocks in order to use AVX-512 instructions: $a = a_{n-1}a_{n-2}\dots a_1a_0$ where a_i is of size 64 bits. For a computation, we can use n vectors $V_i = [a_i, b_i, \dots, h_i]$ for $0 \le i < n$ to represent eight operands and perform $V_i \oplus V_i'$. One needs to take care of any carry and dependency between blocks to ensure correctness. Nonetheless, there are other usages when we have less than eight operands. Other possible options are (i) using $n/2$ vectors $W_i = [a_i, a_{i+n/2}, b_i, b_{i+n/2}, c_i, c_{i+n/2}, d_i, d_{i+n/2}]$ for $0 \le i < n/2$ to represent four operands, (ii) using $n/4$ vectors $X_i = [a_i, a_{i+n/4}, a_{i+2n/4}, a_{i+3n/4}, b_i, b_{i+n/4}, b_{i+2n/4}, b_{i+3n/4}]$ for $0 \le i < n/4$ to represent two operands, and (iii) using $n/8$ vectors $Y_i = [a_i, a_{i+n/8}, a_{i+2n/8}, a_{i+3n/8}, a_{i+4n/8}, a_{i+5n/8}, a_{i+6n/8}, a_{i+7n/8}]$ for $0 \le i < n/8$ to represent only one operand. Following [CFGR22], we call these representations and computations as 8-way, 4-way, 2-way, and 1-way, respectively.

3 Multi-core Implementation using PCS

Our first strategy-level-optimized software implementation of isogeny computation is designed for multi-core processors without AVX-512IFMA instructions. For this setting, we utilize the equation $\mathcal{C}_K^{\text{PCP}^*}$ to construct strategies. To achieve better speed-up, we consider the optimization of [COR22] and modify PCS to accommodate such optimization.

3.1 Modifying PCS for Speed-up

We first describe the speed-up technique of Cervantes-Vázquez et al. [COR22] for multi-core platforms. Some isogeny-based protocols require computing a point R from other points, e.g., $R \leftarrow P + [m]Q$ for $0 \leq m < \ell^e$, before passing it as an input for isogeny computation. For fixed points P and Q of degree ℓ^e, this computation, typically performed using a three-point ladder algorithm, takes time roughly $e \cdot c_{\text{mul}}$ when the size of m is of e bits. However, $[\ell^j]R = [\ell^j]P + [m][\ell^j]Q$ can be computed faster. This is because (i) $[\ell^j]P$ and $[\ell^j]Q$ can be precomputed when P and Q are public and fixed, and (ii) $[m][\ell^j]Q = [m \bmod \ell^{e-j}][\ell^j]Q$ as $\deg([\ell^j]Q) = \ell^{e-j}$. The computation of $[m \bmod \ell^{e-j}]([\ell^j]Q)$ would only take time of $(e-j) \cdot c_{\text{mul}}$. This observation suggests the following implementation.

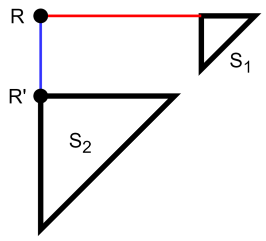

Figure 4: Optimization for multi-core platform proposed by [COR22]: we can compute R' and $\mathcal{S}_2$ without knowing R, thus we can compute R' and $\mathcal{S}_2$ in parallel with R.

Let R denote the result of $P + [m]Q$ and R' denote that of $[\ell^j]P + [m][\ell^j]Q$, where R' is a corner point of a strategy $\mathcal{S}_2$. Since we do not need to know R to compute R' and the computation time of R' is smaller than that of R, Cervantes-Vázquez et al. proposed, for the multi-core platforms, that we devote one core for computing R. While the computation takes place, we use one core to compute R' and then $K-1$ cores to compute the whole strategy $\mathcal{S}_2$. They would choose j such that the computation time of R is close to the computation time of R' and $\mathcal{S}_2$. After R is computed, several isogeny evaluations are serially performed on R before we start computing $\mathcal{S}_1$ using all K cores.

The optimization of [COR22] significantly reduces the isogeny computation time. However, it can be further reduced, as we can see that the isogeny evaluations performed on R are done serially and the computation does not utilize all available cores. Here, we apply the PCS technique to fully utilize those cores. The modified PCS that makes use of this optimization is presented in Algorithm 2.

The algorithm works in two phases. The first phase is for operations performed after R' is computed but before R. The cost is thus initialized as $(e-j) \cdot c_{\text{mul}}$. In this phase, we can utilize $K' = K - 1$ cores and we cannot compute $\phi(R)$, represented by $(1, 0)$, since the computation of R is not yet finished. The cost computation in Lines 21–25 is done as in the original PCS. In Line 26, we check whether the current cost is at least $e \cdot c_{\text{mul}}$, the cost for computing R. If so, this implies that the computation for R is completed and we can start the next phase. In the second phase, we can utilize all K cores. The algorithm continues until all computational tasks are scheduled. For our algorithm, we do not require that the computation time of R is close to the computation time of R' and $\mathcal{S}_2$.

Algorithm 2: Modified PCS for the optimization of [COR22]

```
      ... (Lines 1–10 are the same as in Algorithm 1)
11    cost ← $(e - j) \cdot c_{\mathrm{mul}}$
12    $K' \leftarrow K - 1$
13    Remove vertices $(0,1), \dots, (0,j)$ and their out-going edges from $\mathcal{S}$
14    while $\mathcal{V}_{\mathcal{S}} \neq \emptyset$ do
15        $t \leftarrow t + 1$
16        $\mathcal{V}' \leftarrow \{v \in \mathcal{V}_{\mathcal{S}} : \text{in-degree}(v) = 0\}$
17        if $K' \neq K$ then $\mathcal{V}' \leftarrow \mathcal{V}' \setminus \{(1,0)\}$
18        $S_t \leftarrow \{K'$ vertices in $\mathcal{V}'$ with highest $\mathcal{L}(\cdot)\}$
19        Append $S_t$ to $\mathbb{S}$
20        Remove all vertices in $S_t$ and their out-going edges from $\mathcal{S}$
21        $\text{cost}_t \leftarrow 0$
22        for $(i,j) \in S_t$ do
23            if $\langle (i,j-1),(i,j) \rangle \in \mathcal{E}^*_{\mathcal{S}}$ then $\text{cost}_t \leftarrow \max\{\text{cost}_t, c_{\mathrm{mul}}\}$
24            else $\text{cost}_t \leftarrow \max\{\text{cost}_t, c_{\mathrm{iso}}\}$
25        $\text{cost} \leftarrow \text{cost} + \text{cost}_t$
26        if $\text{cost} \geq e \cdot c_{\mathrm{mul}}$ then $K' \leftarrow K$
27    return $(\text{cost}, \mathbb{S})$
```

3.2 Handling Synchronization

We use the OpenMP API to accommodate multi-threading for our implementation. Although the OpenMP is usually used in the single instruction, multiple data (SIMD) paradigm, this tool provides a way to perform different operations on different cores.

The construct we are using is the OpenMP's `"sections"` and `"section"`. They let us explicitly describe what each core does. Below we show how to use them with three cores.

```
#pragma omp sections
{
  #pragma omp section
    core1_op();
  #pragma omp section
    core2_op();
  #pragma omp section
    core3_op();
}
```

From our scheduling, it is straightforward to convert $\mathbb{S} = \langle S_1, \dots, S_n \rangle$ into a code: we can have n `"sections"`, the i-th `"sections"` represents S_i, and K `"section"` in each `"sections"`, each represents one core. Nonetheless, the resulting implementation is not effective, as there is overhead when starting and ending `"sections"`. To overcome this issue, it is better to have only one `"sections"` and put all operations of each core across all iterations into each `"section"`, e.g., the first `"section"` includes operations for the first core from all $S_1, \dots, S_n$.

However, we also need a synchronization mechanism to ensure the order of operations. For instance, it is possible that the second core starts its second operation while the first core is still working on its first operation and the second operation of the second core requires a result of the first operation of the first core. Considering the algorithm that we used to construct $\mathbb{S}$, we should ensure that all operations in S_1 are finished before we start S_2 in our implementation. We solve this issue by implementing our own barriers.

```
 #pragma omp section        #pragma omp section        #pragma omp section
{                          {                          {
   core1_op1();               core2_op1();               core3_op1();
   L1_1 = 0;                  L2_1 = 0;                  L3_1 = 0;
   while(L2_1 || L3_1);       while(L1_1 || L3_1);       while(L1_1 || L2_1);

   core1_op2();               core2_op2();               core3_op2();
   L1_2 = 0;                  L2_2 = 0;                  L3_2 = 0;
   while(L2_2 || L3_2);       while(L1_2 || L3_2);       while(L1_2 || L2_2);

   ...                        ...                        ...
}                          }                          }
```

The codes above show an example when working with three cores. All three `"section"` are under one single `"sections"` similar to the earlier fragment. The initial values of all variables `Lj_i`, representing a status for j-th core and i-th operation, is 1. For the scheduling, let S_i contain three operations: `core1_opi`, `core2_opi`, and `core3_opi`. We give one operation to each core. After a core performs its task, that core changes the variable to 0, signaling other cores that its operation is done. Then, that core keeps checking whether other cores finish their operations. As soon as all variables are set to 0, all cores continue to the next operation. This mechanism ensures correct order and is not costly when implemented.

3.3 Implementation Results

We implement our first strategy-level-optimized isogeny computation, combining all speed-up techniques in previous subsections. The implementations are based on the SIKEp751 parameter set, where the underlying field is $\mathbb{F}_{p^2}$ with $p = 2^{372}3^{239} - 1$. We computed two isogenies of degree 4^{186} and 3^{239}, respectively. After that, we execute them on an Intel(R) Core(TM) i7-8700 processor, benchmarking them with those of [COR22]. For the reproducibility of the results, the Intel Hyper-Threading and Intel Turbo Boost technologies were disabled. For benchmarking, we employ the same set of parameters, including the prime and the extension field, used in the works that we compare our results with.

Table 3 compares execution times of several implementations with one to four cores. The execution times for the single-core setting are shown for reference. For the multi-core setting, we compare four implementations: the implementation of [COR22] and our implementation which utilizes PCS, each with and without the optimization of [COR22] described in Section 3.1. We note that the optimization is applied only on the implementations of 4^{186}-isogeny computation. For the isogeny computation, there are two rounds for each ℓ^e-isogeny computed: the first round includes the computation of (ϕ, E') from (E, R) and the computation of three image points $\phi(P_1), \phi(P_2), \phi(P_3)$ for some given points P_1, P_2, P_3, while the second round includes only the computation of (ϕ, E') from (E, R).

It is clear from Table 3 a reduction in the execution times when there are more cores available. Overall, our implementations have better speed compared to [COR22], and the reduction percentage increases when there are more cores. For implementations including the optimization of [COR22], the maximum reduction is up to 14.36% for the second round of 4^{186}-isogeny computation with four cores. For implementations without the optimization, the maximum reduction is up to 16.79% also for the second round of 4^{186}-isogeny computation with four cores. These results show the significance of strategy-level optimization for low-latency parallel isogeny computation.

When the aforementioned implementations of isogeny computation is employed to build the complete isogeny-based protocol SIKEp751, we observe superior results with PCS. We refer the interested readers to the appendix for details.

Table 3: Execution times of various isogeny computation implementations (in million CPU cycles). The first round includes the computation of (ϕ, E') from (E, R) and three image points $\phi(P_1), \phi(P_2), \phi(P_3)$ for some points P_1, P_2, P_3. The second round includes only the computation of (ϕ, E') from (E, R). $(*)$ denotes implementations utilizing the optimization by [COR22] described in Section 3.1. The % reduction shows how much the execution time of our implementation (the row above) is reduced from that of [COR22] (the row before).

# Cores	Implementation	4^{186}-isogeny		3^{239}-isogeny	
		Round 1	Round 2	Round 1	Round 2
1	[COR22]	22.96	18.85	25.98	22.16
2	[COR22]	20.60	16.47	23.23	19.39
	This work	18.78	14.68	21.24	17.42
	% reduction	8.83	10.87	8.57	10.16
	[COR22] $(*)$	16.30	14.69		
	This work $(*)$	14.76	12.85		
	% reduction	9.45	12.53		
3	[COR22]	19.53	15.42	21.87	18.06
	This work	17.65	13.53	19.60	15.79
	% reduction	9.63	12.26	10.38	12.57
	[COR22] $(*)$	14.95	13.32		
	This work $(*)$	13.34	11.71		
	% reduction	10.77	12.09		
4	[COR22]	19.09	14.89	21.03	17.15
	This work	16.62	12.39	18.68	14.80
	% reduction	12.94	16.79	11.17	13.70
	[COR22] $(*)$	14.08	12.67		
	This work $(*)$	12.81	10.85		
	% reduction	9.02	14.36		

4 AVX-512 and Multi-Core Implementation using PCS

We present in this section our second strategy-level-optimized software implementation of isogeny computation designed for processors supporting AVX-512. We first consider the execution environment for the implementation and propose a modified version of $\mathcal{C}_K^{\mathrm{PCP}^*}$ that suits better in this setting. We then apply the PCS technique to an implementation of isogeny computation that uses AVX-512 instructions and multi-threading. Lastly, timing results comparing previous implementations with ours are presented.

4.1 Constructing Better Strategies using Modified PCP

We consider the implementation of [CFGR22] as a starting point. As mentioned in Section 2.4, their implementation sees 512-bit vectors as eight elements. Although in Section 2.4 we described the usage at the low-level operations $a \oplus a'$, the idea can be applied to a higher level of isogeny evaluations. Here, we define n-way isogeny evaluation for $n \in \{8, 4, 2, 1\}$ as representing n elliptic curve points in one vector and evaluating n points simultaneously. They are designed so that we use as many as possible vector elements, although there are less than eight points to be evaluated and, as a consequence, less points evaluated implies less execution time. As an example, we performed an experiment to obtain execution times for each number of points evaluated concurrently by 4-isogeny. The experiments were performed on an Intel(R) Core(TM) i5-11400 processor. Table 4 below shows the results.

Table 4: Execution times of [CFGR22] for different number of points evaluated concurrently by 4-isogeny. $(*)$ denotes cases where some vector elements are unused.

Points evaluated	1	2	3	4	5	6	7	8
Computation	1-way	2-way	4-way $(*)$	4-way	4-way then 1-way	4-way then 2-way	8-way $(*)$	8-way
Execution time (CPU cycles)	4900	5800	9300	9400	13600	14500	16400	16500

Based on these results, it is obvious that the execution times differ for different number of points evaluated in the AVX-512 implementation. Therefore, using the equation $\mathcal{C}_K^{\mathrm{PCP}^*}$ with $K = 8$ to construct strategies in this execution environment is not accurate and might not give us the best speed-up, as PCP assumes that K evaluations take time equal to one evaluation. Therefore, before we apply the PCS technique, we should modify PCP first in order to obtain better strategies for the current setting.

For $\mathcal{C}_K^{\mathrm{PCP}^*}(e, k)$, the equation focuses on the number of cores k available to perform operations. We instead focus on the number of operations to be performed. We elaborate our intuition with the following figure.

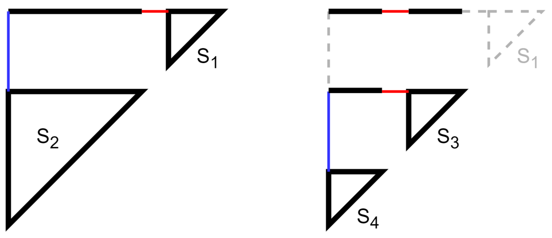

Figure 5: Computing the costs of strategies using modified PCP: We need one isogeny evaluation when moving from $\mathcal{S}_2$ to $\mathcal{S}_1$ and two evaluations from $\mathcal{S}_4$ to $\mathcal{S}_3$.

Suppose we decompose strategy $\mathcal{S}$ to $\mathcal{S}_1$ and $\mathcal{S}_2$. The costs of point multiplications (shown as a blue vertical thin line) can be determined. One isogeny evaluation connecting $\mathcal{S}_2$ and $\mathcal{S}_1$ (shown as a red horizontal thin line) can also be calculated. For $\mathcal{S}_2$, we see that there is a line representing isogeny evaluations above the triangle of $\mathcal{S}_2$. This line needs to be taken into account when we perform operations of $\mathcal{S}_2$. While PCP says that this line will occupy one core of the processor and we are left with $K - 1$ cores, we otherwise take a note that there is one line included with $\mathcal{S}_2$. Recursively, $\mathcal{S}_2$ is then decomposed to $\mathcal{S}_3$ and $\mathcal{S}_4$. Since we noted that there is one line above $\mathcal{S}_2$ triangle, we can infer that there will be one line above $\mathcal{S}_3$ triangle and two lines above $\mathcal{S}_4$ triangle. Also, we know that the number of isogeny evaluations between $\mathcal{S}_3$ and $\mathcal{S}_4$ (shown as red horizontal thin lines) is two, which is equal to the number of lines above the $\mathcal{S}_4$ triangle.

To formalize this intuition, let $\mathcal{C}^{\mathrm{MP}^*}(e, r)$ denote the lowest cost of strategies for ℓ^e-isogeny when there are r lines above the strategy triangle. What we would like to find is thus $\mathcal{C}^{\mathrm{MP}^*}(e, 0)$. The dynamic programming equation for the modified PCP is as follows:

$$\mathcal{C}^{\mathrm{MP}^*}(e, r) = \begin{cases} 0 & \text{if } e = 1, \\ \min\limits_{0<i<e} \{\mathcal{C}^{\mathrm{MP}^*}(i, r+1) + \mathcal{C}^{\mathrm{MP}^*}(e-i, r) + f_{\mathrm{mul}}(e-i) + f_{\mathrm{iso}}(r+1)\} & \text{otherwise.} \end{cases}$$

The function $f_{\mathrm{mul}}(n)$ denotes the cost of serially performing n point multiplications by $[\ell]$, and the function $f_{\mathrm{iso}}(n)$ denotes the cost of performing n isogeny evaluations. If we let

$f_{\text{mul}}(n) = n \cdot c_{\text{mul}}$ and $f_{\text{iso}}(n) = \lceil n/K \rceil \cdot c_{\text{iso}}$, our $\mathcal{C}^{\text{MP}^*}$ will be $\mathcal{C}^{\text{PCP}^*}$. In other words, our equation is a generalization of PCP.

In the setting of AVX-512, the values of $f_{\text{iso}}(n)$ for $1 \le n \le 8$ are taken according to Table 4. When $n > 8$, it is not straighforward to say that $f_{\text{iso}}(n) = f_{\text{iso}}(n-8) + f_{\text{iso}}(8)$. For an example of $n = 10$, we can split the computation to $8+2$, $6+4$, or even $4+4+2$. To obtain the optimal computation, we need another dynamic programming equation:

$$f_{\text{iso}}(n) = \begin{cases} \text{See Table 4} & \text{if } 0 \le n \le 8, \\ \min\limits_{1 \le i \le 8} \{f_{\text{iso}}(n-i) + f_{\text{iso}}(i)\} & \text{otherwise.} \end{cases}$$

Lastly, for the value of $f_{\text{mul}}(n)$, it is almost linear to n. Based on our experiment, [CFGR22] gave $f_{\text{mul}}(n) = 5200n + 800$ (in CPU cycles) for point multiplications by [4].

4.2 Applying PCS to AVX-512 Implementation

After obtaining better strategies, we are ready to apply the PCS technique to them in order to get a more efficient implementation. Since the main advantage of PCS is to perform point multiplications and isogeny evaluations simultaneously but vectorization works well with a single type of operation at a time, we decided to consider multi-threading for our AVX-512 implementation. In particular, we utilize AVX-512 instructions and also two to four cores of the multi-core processor. To the best of our knowledge, this work is the first to combine both vectorization and multi-core processor for isogeny computation.

Revisiting Modified PCP. For $K \in \{2, 3, 4\}$ cores, we can perform up to $8K$ isogeny evaluations at a time. Hence, we revisit our $\mathcal{C}^{\text{MP}^*}$ equation for necessary modifications.

The only thing we need to modify is the function $f_{\text{iso}}(n)$. Now Table 4 can be used for n up to $8K$. The same issue happens for $n > 8K$. Nonetheless, the fix is straightforward.

$$f_{\text{iso}}(n) = \begin{cases} \text{See Table 4 for } \lceil n/K \rceil & \text{if } 0 \le n \le 8K, \\ \min\limits_{1 \le i \le 8K} \{f_{\text{iso}}(n-i) + f_{\text{iso}}(i)\} & \text{otherwise.} \end{cases}$$

Balancing Times of Two Operations. Under PCS, point multiplications by $[\ell]$ and isogeny evaluations are allowed to be performed in parallel on different cores. To effectively perform both, their execution times should not differ much. For example, we see that performing one point multiplication by [4] takes time $f_{\text{mul}}(1) = 6000$ and performing 2-way 4-isogeny evaluations takes time $f_{\text{iso}}(2) = 5800$. Therefore, it is effective to perform one point multiplication by [4] on one core and 2-way 4-isogeny evaluations on another.

Modifying PCS. The PCS technique can be applied to our setting with some changes. To obtain low-latency implementation, we consider two issues. The first one is the balance of operations previously mentioned. By the proof of [HK18], any two point multiplications by $[\ell]$ in a strategy constructed by $\mathcal{C}^{\text{PCP}^*}$ or $\mathcal{C}^{\text{MP}^*}$ cannot be computed at the same time. This implies that, if one core is used to perform point multiplications, then other remaining cores will be used for isogeny evaluations or left idle. In the former case, the execution times of other cores should be close to the core performing point multiplications. In Algorithm 3 which is a modified version of Algorithm 1, these are shown in Lines 15–18.

The second issue we consider is the number of isogeny evaluations performed at one time when no point multiplication is available. Under PCP, since K evaluations take time equal to one evaluation, it is best to greedily perform as many evaluations as possible. However, for AVX-512, that is not the case. As an example, suppose $K = 4$ and there are currently 25 isogeny evaluations to be performed. We could perform seven of them on one core and six on three other cores. This will take time $f_{\text{iso}}(7)$. However, it is better to perform only 24 of them and leave one for later, taking time $f_{\text{iso}}(6)$. This has not been considered before because of the design principle of PCP used in all implementations. Thus, our first step is to perform isogeny evaluations as a multiple of K.

We can optimize this further. By looking at Table 4 and Section 2.4, it is more effective to perform one, two, four, eight isogeny evaluations at a time on each core. For the example of 25 evaluations, we can perform only 16 of them, rather than 24, with the cost of $f_{\text{iso}}(4)$. By this scheduling, we could effectively utilize AVX-512 instructions. This optimization appears in Lines 19–26 of Algorithm 3.

Algorithm 3: Modified PCS for K-Core AVX-512 Implementation

```
    ... (Lines 1–11 are the same as in Algorithm 1)
12  while $\mathcal{V}_{\mathcal{S}} \neq \emptyset$ do
13      $t \leftarrow t + 1$
14      $\mathcal{V}' \leftarrow \{v \in \mathcal{V}_{\mathcal{S}} : \text{in-degree}(v) = 0\}$
15      if there exists $(i, j) \in \mathcal{V}'$ such that $\langle (i, j-1), (i, j) \rangle \in \mathcal{E}^*_{\mathcal{S}}$ then
16          $\mathcal{V}' \leftarrow \{2(K-1)$ vertices in $\mathcal{V}' \setminus \{(i, j)\}$ with highest $\mathcal{L}(\cdot)\}$
17          $S_t \leftarrow \{(i, j)\} \cup \mathcal{V}'$
18          $\text{cost}_t \leftarrow \max\{f_{\text{mul}}(1), f_{\text{iso}}(\lceil \#\mathcal{V}'/(K-1) \rceil)\}$
19      else
20          $n \leftarrow \#\mathcal{V}'$
21          if $8K \leq n$ then $n \leftarrow 8K$
22          if $4K \leq n < 8K$ then $n \leftarrow 4K$
23          if $2K \leq n < 4K$ then $n \leftarrow 2K$
24          if $K \leq n < 2K$ then $n \leftarrow K$
25          $S_t \leftarrow \{n$ vertices in $\mathcal{V}'$ with highest $\mathcal{L}(\cdot)\}$
26          $\text{cost}_t \leftarrow f_{\text{iso}}(\lceil n/K \rceil)$
27      Append $S_t$ to $\mathbb{S}$
28      Remove all vertices in $S_t$ and their out-going edges from $\mathcal{S}$
29      $\text{cost} \leftarrow \text{cost} + \text{cost}_t$
30  return $(\text{cost}, \mathbb{S})$
```

4.3 Implementation Results

We implement our second strategy-level-optimized isogeny computation which uses AVX-512 and two-to-four cores, including optimizations in Sections 4.2 and 3.2. The implementations are based on the SIKEp751 parameter set, where the underlying field is $\mathbb{F}_{p^2}$ with $p = 2^{372}3^{239} - 1$. We computed two isogenies of degree 4^{186} and 3^{239}, respectively. Then, we execute them on an Intel(R) Core(TM) i5-11400 processor, together with other existing works. As usual, we disable the Intel Hyper-Threading and Intel Turbo Boost technologies for reproducibility. For benchmarking, we employ the same set of parameters, including the prime and the extension field, used in the works that we compare our results with.

The results are shown in Table 5. In the table, two single-core implementations are those proposed by [Mic17] with no use of AVX technologies and [CFGR22] with the use of AVX-512. For two to four cores, we present two implementations for each one of them: one is obtained by applying PCP (Section 2.2) to strategies constructed from our modified PCP (Section 4.1), and the other is obtained by applying our modified PCS (Section 4.2) to strategies constructed from our modified PCP (Section 4.1). For the isogeny computation, there are two rounds for each ℓ^e-isogeny computed: the first round includes the computation of (ϕ, E') from (E, R) and the computation of three image points $\phi(P_1), \phi(P_2), \phi(P_3)$ for some given points P_1, P_2, P_3, while the second round includes only the computation of (ϕ, E') from (E, R).

Table 5: Execution times of various isogeny computation implementations (in million CPU cycles). The first round includes the computation of (ϕ, E') from (E, R) and three image points $\phi(P_1), \phi(P_2), \phi(P_3)$ for some points P_1, P_2, P_3. The second round includes only the computation of (ϕ, E') from (E, R). The % reduction shows how much the execution time of our best implementation in the eighth row is reduced from that of [CFGR22].

# Cores	Implementation	4^{186}-isogeny		3^{239}-isogeny	
		Round 1	Round 2	Round 1	Round 2
1	[Mic17]	20.11	16.49	22.73	19.40
	[CFGR22], AVX	8.39	7.71	10.25	9.54
2	This work, AVX, PCP	7.26	6.95	8.63	8.31
	This work, AVX, PCS	6.44	6.34	7.80	7.62
3	This work, AVX, PCP	6.64	6.50	7.94	7.75
	This work, AVX, PCS	5.89	5.96	7.16	7.13
4	This work, AVX, PCP	6.28	6.26	7.51	7.51
	This work, AVX, PCS	5.61	5.75	6.76	6.96
	% reduction	33.13	25.42	34.05	27.04

The results clearly show the advantage of using vectorization and multi-core processors for isogeny computation, as all of our multi-core implementations are faster than [Mic17] and [CFGR22]. We note that the underlying arithmetic computation of [CFGR22] and ours are the same. As indicated in the bottom row of Table 5, the reduction is up to 34.05% for Round 1 of 3^{239}-isogeny when utilizing four cores. Once again, the implementation results support the importance of optimizing isogeny computation at the strategy-level. Even though using AVX-512 on a multi-core platform leads to a faster implementation, we may not obtain the best results if we do not consider optimizations for strategy construction and evaluation. By changing the parallelization technique from PCP to PCS, the execution time can be reduced by up to $\frac{6.64-5.89}{6.64} = 11.30\%$ (4^{186}-isogeny, Round 1, three cores).

When the aforementioned implementations of isogeny computation is employed to build the complete isogeny-based protocol SIKEp751, we observe superior results with PCS. We refer the interested readers to the appendix for details. The appendix also includes wall times of all implementations to show the correspondence to cycle counts.

4.4 Efficiency Analysis

The previous subsection shows 25–34% reduction in the execution time when employing four cores, compared to the single-core implementation of [CFGR22]. At first glance, four-time speed-up (or equivalently 75% reduction) might be expected. In this subsection we analyze the theoretical speed-up together with efficiencies of our strategies and implementations. For the analysis, we refer to data under the column of 4^{186}-isogeny, Round 2, in Table 5.

In the following, we define the *overall computational cost* for a strategy designed for the multi-core setting as the cost of that strategy when it is evaluated using a single core. The *overall computational cost* reflects the number of operations in a strategy, while the multi-core strategy cost depends both on the number of operations and how much we can parallelize them.

Theoretical Speed-Up. The theoretical single-core strategy cost from [CFGR22] (computed using the modified PCP in Section 4.1) is 5.53 million CPU cycles, and the four-core strategy cost of that achieved by our four-core implementation (computed by Algorithm 3) is 3.39 million CPU cycles. By looking only at the theoretical strategy costs, the expected speed-up is 38.70%. This is 1.5 times higher than what we achieved (25.42%, bottom row of Table 5) presumably due to the communication overhead among cores. Next, we analyze the maximum theoretical speed-up. For this, it is common to assume that we have an infinite number of cores. However, [HK18] and [COR22] described that the highest level of parallelization is obtained for $e-1$ cores as it is not useful to have more than $e-1$ cores. For 4^{186}-isogeny, e is 186. In this case, the strategy used for the computation is the *isogeny-based* strategy [FJP14] (i.e., all points except for those in the leftmost column are computed from isogeny evaluations) and its strategy cost is $f_{\mathrm{mul}}(e-1)+(e-1)\cdot f_{\mathrm{iso}}(1)$. For our current setting, this would result in 1.87 million CPU cycles. Hence, even in the case of having plentiful cores, the maximum theoretical speed-up is $\frac{5.53}{1.87}=2.96$ times (or equivalently 66.20% reduction). We note that the actual speed-up from the implementation is expected to be less than this due to the synchronization costs.

Strategy Efficiency. When there are more cores available, the strategies used in our implementations require more computations compared to the one used by [CFGR22]. For the strategy used by our four-core implementation, its *overall computational cost* (computed using the modified PCP in Section 4.1) is 6.68 million CPU cycles, which is higher than that of [CFGR22]. Nonetheless, a higher number of operations allows multiple cores to concurrently perform the computation, resulting in a lower latency. If we apply four cores to the strategy of [CFGR22], its strategy cost (computed by Algorithm 3) is 4.76 million CPU cycles, which is higher than ours. Therefore, we could say that we increase the *overall computational cost* by $\frac{6.68-5.53}{5.53}=20.80\%$, but reduce the four-core strategy cost by $\frac{4.76-3.39}{4.76}=28.78\%$. This is preferable in order to have low-latency implementations.

Implementation Efficiency. From the aforementioned results, the efficiency of our approach decreases when the number of cores increases. One way of looking at the efficiency of our K-core, $K\in\{2,3,4\}$, implementation is to express it as $\frac{T_1}{K\cdot T_K}$, where T_x is the execution time of the x-core implementation. For our implementations, the efficiencies are $\frac{7.71}{2\times 6.34}=60.80\%$ for two cores, $\frac{7.71}{3\times 5.96}=43.12\%$ for three cores, and $\frac{7.71}{4\times 5.75}=33.52\%$ for four cores. The efficiency is expected to decrease when there are more cores as we trade *overall computational cost* for latency.

In addition, we run our proposed multi-core implementations utilizing PCS on a single core to have a better understanding of the extent of any overheads in the execution time (e.g., due to communication between cores). The corresponding execution times are shown in Table 6, and we compare theoretical costs and actual running times of our multi-core implementations and their serializations corresponding to the computation of 4^{186}-isogeny in Table 7. Assuming that the execution times of the serialized versions correspond to their theoretical costs, the overhead percentage increases from $\frac{0.78-0.64}{0.64}=21.88\%$ for two cores to $\frac{0.57-0.39}{0.39}=46.15\%$ for four cores. This is another indication that having more cores may not always be beneficial due to an overhead increase.

5 Discussion

The implementation results in Sections 3 and 4 show notable speed-ups when applying PCS with the SIKEp751 parameter set of SIKE as software implementations. In this section, we discuss the applicability of the proposed techniques to a variety of other settings and provide some remarks on the cost functions and the optimality of our strategies.

Applicability to Other Settings. The settings considered below include different vectorization technologies, alternative implementations of arithmetic in the underlying finite field, hardware implementation, and other isogeny-based schemes.

Different vectorization technologies. Although AVX-512 is currently the most powerful extension available, other technologies such as AVX2 may be arguably far more widely used. When using AVX2, vectors are only of size 256 bits and one will need to adjust the implementations of point multiplications, isogeny evaluations, and other primitive

Table 6: Execution times of the serialized version of our proposed isogeny computation implementations (in million CPU cycles).

Implementation	4^{186}-isogeny		3^{239}-isogeny	
	Round 1	Round 2	Round 1	Round 2
Serialization of 2-core PCS	8.86	8.14	10.09	9.57
Serialization of 3-core PCS	9.63	8.87	11.44	10.24
Serialization of 4-core PCS	10.47	10.04	11.62	11.13

Table 7: Theoretical strategy costs and actual execution times of multi-core and serialized versions of our 4^{186}-isogeny computation implementations, Round 2 (in million CPU cycles). The Ratio columns show the ratio between Multi-Core and Serialized columns.

Implementation	Theoretical Cost			Execution Time		
	Multi-Core	Serialized	Ratio	Multi-Core	Serialized	Ratio
2-core PCS	4.06	6.39	0.64	6.34	8.14	0.78
3-core PCS	3.66	7.28	0.50	5.96	8.87	0.67
4-core PCS	3.39	8.65	0.39	5.75	10.04	0.57

arithmetic accordingly. After the implementations of fundamental operations are ready, we require the execution times of those operations (similar to what is discussed in Section 4.1) in order to adjust the functions f_{mul}, f_{iso} and apply our proposed techniques. We expect a similar extent of effectiveness when applying our work with AVX2-supported processors. It is also of interest to apply our techniques to ARM's Scalable Vector Extension (SVE).

Alternative arithmetic package. Our techniques are applicable regardless of the implementation of the arithmetic in the underlying field $\mathbb{F}_{p^2}$. The speed of the arithmetic operations has a profound influence on the speed of the implementation. This work relies on the arithmetic implementations of [Mic17, CFGR22] which may no longer represent the state-of-the-art due to the recent result of [Lon22]. A faster arithmetic level implementation will likely result in a higher speed-up. For that, one requires the execution times of those arithmetic operations to adjust with our work.

Hardware implementations. The number of operations that can be performed in parallel can vary based on implementations and the number of logic gates or FPGA slices available. The maximum number of operations performed concurrently and their execution times can be used to customize our work accordingly, similar to earlier discussion. Unlike other works [KAK16, KAK+20], the operations done in each iteration need to be specified explicitly. Thus, the control circuit may become complicated and optimizing it can be challenging.

Other isogeny-based schemes. For CSIDH, we have suggested some tweaks at the beginning of this paper, and several works have studied strategies for the protocol [HLKA20, CR22]. Nonetheless, while extending our work to CSIDH one also needs to think about the implementation of $\mathbb{F}_p$ arithmetic and the way to handle a case when a sampled point cannot generate an isogeny. For SQISign, its source code suggests the use of strategies as an improvement. This would be a natural application of our techniques to SQISign. As well, one requires the implementation of $\mathbb{F}_{p^2}$ arithmetic designed specifically for its prime.

The Cost Functions. In this work, the cost of a strategy is based solely on two parameters: the costs of computing point multiplications and isogeny evaluations. This is the approach used in all existing works, to our best knowledge, for both single-core [FJP14] and multi-core platforms [HK18, COR22]. Consequently, the theoretical cost may not be a close approximation of the actual execution time. For better cost computation, one may need to take into account the costs of synchronization, memory access, etc. However, our proposal considers strategy construction and evaluation as separate processes. Thus, we are currently not able to determine the architectural costs during the construction. To handle this, a new computation model needs to be devised, leading to an open problem.

Optimality of Strategies. Following [PSH22], we randomly construct optimal strategies under modified PCP and then parse them as inputs to PCS. We search up to 100,000 of such strategies, taking us a few hours. Nevertheless, the search is done only once in order to find one strategy used for the implementation, and the search time does not affect the execution time of the cryptographic protocol. Also, while our strategies are optimal under PCP, we do not claim that they are globally optimal in general due to several layers of approximation as stated in [PSH22] and since we only search for strategies with recursive structure as in Figure 1(c). Additional improvements when employing unstructured strategies may be possible, and the optimality of strategies remains to be an open problem.

6 Conclusion

This work has illustrated how software implementation of large smooth-degree isogeny computation, specifically with vectorization and parallelism, can be further sped-up at the strategy-level. For the first implementation, which considers only the multi-core parallelism, we are able to gain a speed-up when utilizing the modified PCS adapted for the existing optimization. The execution time is reduced by up to 14.36% in our implementation compared to that of [COR22]. The second implementation, equipped with AVX-512 technology and multi-core processors, combines the use of the modified PCP and PCS in order to provide effective strategies and evaluations crafted for the execution environment. Our benchmarking shows a reduction in execution time of up to 34.05% compared to the single-core implementation of [CFGR22] when utilizing up to four cores. Apart from these, our synchronization handling mechanism is also important in achieving a low-latency vectorized and parallel software implementation for the isogeny computation.

In this paper, we have also provided commentaries on the applicability of our work to a variety of other settings including AVX2, hardware implementations, and other isogeny-based schemes. Moreover, there is a possibility to further speed-up our implementations by adopting more cores and faster implementation of $\mathbb{F}_{p^2}$ arithmetic operations. Nonetheless, at some point, adding more cores will not be advantageous due to the increasing synchronization overhead. Thus, it is of interest to investigate the effect of applying our proposed approaches to the above-mentioned settings and improvements.

Acknowledgement. The authors would like to thank Ruben Niederhagen and the reviewers for their constructive comments on improving the manuscript. We would also like to acknowledge the authors of the implementations of [Mic17, CFGR22, COR22] which we use as basis for our implementations. The first author is supported by the Ripple Impact Fund through a Ripple Graduate Fellowship. The second author is supported by JSPS Grant-in-Aid for Transformative Research Areas A grant number JP21H05845.

References

[ALJK18] Reza Azarderakhsh, Elena Bakos Lang, David Jao, and Brian Koziel. EdSIDH: supersingular isogeny Diffie-Hellman key exchange on Edwards curves. In Anupam Chattopadhyay, Chester Rebeiro, and Yuval Yarom, editors, *Security, Privacy, and Applied Cryptography Engineering - 8th International Conference, SPACE 2018, Kanpur, India, December 15-19, 2018, Proceedings*, volume 11348 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2018.

[BDFLS20] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. Faster computation of isogenies of large prime degree. *Open Book Series*, 4(1):39–55, 2020.

[CD22] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Report 2022/975, 2022. https://eprint.iacr.org/2022/975.

[CFGR22] Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Peter Y. A. Ryan. Highly vectorized SIKE for AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(2):41–68, Feb. 2022.

[CJG72] Edward G. Coffman Jr. and Ronald L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[CLG09] Denis Xavier Charles, Kristin E. Lauter, and Eyal Z. Goren. Cryptographic hash functions from expander graphs. *Journal of Cryptology*, 22(1):93–113, January 2009.

[CLM$^+$18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 395–427. Springer, Heidelberg, December 2018.

[CLN16] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 572–601. Springer, Heidelberg, August 2016.

[COR22] Daniel Cervantes-Vázquez, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. Parallel strategies for SIDH: toward computing SIDH twice as fast. *IEEE Trans. Computers*, 71(6):1249–1260, 2022.

[CR22] Jesús-Javier Chi-Domínguez and Francisco Rodríguez-Henríquez. Optimal strategies for CSIDH. *Adv. Math. Commun.*, 16(2):383–411, 2022.

[CS99] Fabián A. Chudak and David B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *J. Algorithms*, 30(2):323–343, 1999.

[CSRT22] Jorge Chávez-Saab, Francisco Rodríguez-Henríquez, and Mehdi Tibouchi. Verifiable isogeny walks: Towards an isogeny-based postquantum VDF. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 441–460. Springer, Heidelberg, September / October 2022.

[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, Heidelberg, December 2020.

[DMPS19] Luca De Feo, Simon Masson, Christophe Petit, and Antonio Sanso. Verifiable delay functions from supersingular isogenies and pairings. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 248–277. Springer, Heidelberg, December 2019.

[DPB17] Javad Doliskani, Geovandro C. C. F. Pereira, and Paulo S. L. M. Barreto. Faster cryptographic hash function from supersingular isogeny graphs. Cryptology ePrint Archive, Report 2017/1202, 2017. https://eprint.iacr.org/2017/1202.

[FJP14] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Math. Cryptol.*, 8(3):209–247, 2014.

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

[Gra66] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell system technical journal*, 45(9):1563–1581, 1966.

[HK18] Aaron Hutchinson and Koray Karabina. Constructing canonical strategies for parallel implementation of isogeny based cryptography. In Debrup Chakraborty and Tetsu Iwata, editors, *INDOCRYPT 2018*, volume 11356 of *LNCS*, pages 169–189. Springer, Heidelberg, December 2018.

[HLKA20] Aaron Hutchinson, Jason T. LeGrow, Brian Koziel, and Reza Azarderakhsh. Further optimizations of CSIDH: A systematic approach to efficient strategies, permutations, and bound vectors. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 20, Part I*, volume 12146 of *LNCS*, pages 481–501. Springer, Heidelberg, October 2020.

[Hu61] T. C. Hu. Parallel sequencing and assembly line problems. *Operations research*, 9(6):841–848, 1961.

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

[JD11] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 19–34. Springer, Heidelberg, November / December 2011.

[KAK16] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Fast hardware architectures for supersingular isogeny Diffie-Hellman key exchange on FPGA. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *INDOCRYPT 2016*, volume 10095 of *LNCS*, pages 191–206. Springer, Heidelberg, December 2016.

[KAK$^+$20] Brian Koziel, A.-Bon E. Ackie, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari Kermani. SIKE'd Up: fast hardware architectures for Supersingular Isogeny Key Encapsulation. *IEEE Trans. Circuits Syst.*, 67-I(12):4842–4854, 2020.

[KG19] Dusan Kostic and Shay Gueron. Using the new VPMADD instructions for the new post quantum key encapsulation mechanism SIKE. In Naofumi Takagi, Sylvie Boldo, and Martin Langhammer, editors, *26th IEEE Symposium on Computer Arithmetic, ARITH 2019, Kyoto, Japan, June 10-12, 2019*, pages 215–218. IEEE, 2019.

[KJA+16] Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari Kermani. NEON-SIDH: Efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM. In Sara Foresti and Giuseppe Persiano, editors, *CANS 16*, volume 10052 of *LNCS*, pages 88–103. Springer, Heidelberg, November 2016.

[Lon22] Patrick Longa. Efficient algorithms for large prime characteristic fields and their application to bilinear pairings and supersingular isogeny-based protocols. Cryptology ePrint Archive, Report 2022/367, 2022. https://eprint.iacr.org/2022/367.

[Mic17] Microsoft Research. SIDH library, 2017. https://github.com/microsoft/PQCrypto-SIDH.

[MM22] Luciano Maino and Chloe Martindale. An attack on SIDH with arbitrary starting curve. Cryptology ePrint Archive, Report 2022/1026, 2022. https://eprint.iacr.org/2022/1026.

[PSH22] Kittiphon Phalakarn, Vorapong Suppakitpaisarn, and M. Anwar Hasan. Speeding-up parallel computation of large smooth-degree isogeny using precedence-constrained scheduling. In Khoa Nguyen, Guomin Yang, Fuchun Guo, and Willy Susilo, editors, *ACISP 22*, volume 13494 of *LNCS*, pages 309–331. Springer, Heidelberg, November 2022.

[Rob22] Damien Robert. Breaking SIDH in polynomial time. Cryptology ePrint Archive, Report 2022/1038, 2022. https://eprint.iacr.org/2022/1038.

[SLLH18] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. SIDH on ARM: Faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR TCHES*, 2018(3):1–20, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7266.

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

Appendix A: Benchmarks for SIKEp751

As a supplement, we had integrated both of our optimized isogeny computation implementations into SIKE [JAC+20], focusing on the SIKEp751 parameter set. Table 8 shows the implementation results when using our multi-core implementation with the optimization of [COR22], and Table 9 shows the results when using our AVX-512 and multi-core implementation. Specifically, the first round of isogeny computation in Table 3 and 5 refers to the key generation phase of SIDH [JD11] and the second round refers to the secret agreement phase. The reduction percentages, comparing our work with the best existing result, have the same trend as in our previous benchmarkings.

Appendix B: Wall Time for AVX-512 Implementations

In addition to benchmarking AVX-512 implementations for cycle counts, we also benchmarked them for wall times (with the Intel Hyper-Threading and Intel Turbo Boost technologies enabled to replicate actual runnings) as in Table 10. Based on the results, the cycle counts of all implementations (ref. Table 5) correspond to wall times.

Table 8: Execution times of various SIKE implementations using the SIKEp751 parameter set (in million CPU cycles). (∗) denotes implementations utilizing the optimization by [COR22] described in Section 3.1. The % reduction shows how much the execution time of our implementation (the row above) is reduced from that of [COR22] (the row before).

# Cores	Implementation	KeyGen	Encaps	Decaps
1	[COR22]	25.98	42.08	45.18
2	[COR22] (∗)	23.28	30.76	35.73
	This work (∗)	21.20	27.88	32.18
	% reduction	8.93	9.36	9.94
3	[COR22] (∗)	21.95	28.32	33.12
	This work (∗)	19.78	25.40	29.59
	% reduction	9.89	10.31	10.66
4	[COR22] (∗)	21.01	26.72	31.14
	This work (∗)	18.72	23.65	27.74
	% reduction	10.90	11.49	10.92

Table 9: Execution times of various SIKE implementations using the SIKEp751 parameter set (in million CPU cycles). (∗) denotes an implementation where two isogeny computations in Encaps are combined. The % reduction shows how much the execution time of our best implementation in the ninth row is reduced from that of [CFGR22] (∗) in the third row.

# Cores	Implementation	KeyGen	Encaps	Decaps
1	[Mic17]	22.88	36.87	44.21
	[CFGR22], AVX	10.26	16.12	17.93
	[CFGR22], AVX (∗)	10.26	12.80	17.93
2	This work, AVX, PCP	8.61	14.19	15.64
	This work, AVX, PCS	7.77	12.91	14.18
3	This work, AVX, PCP	7.94	13.14	14.43
	This work, AVX, PCS	7.24	11.93	13.11
4	This work, AVX, PCP	7.51	12.56	13.79
	This work, AVX, PCS	6.76	11.41	12.67
	% reduction	34.11	10.86	29.34

Table 10: Execution times of various isogeny computation implementations (in milliseconds). The Intel Hyper-Threading and Intel Turbo Boost technologies are enabled.

# Cores	Implementation	4^{186}-isogeny		3^{239}-isogeny	
		Round 1	Round 2	Round 1	Round 2
1	[Mic17]	4.693	3.787	5.225	4.445
	[CFGR22], AVX	1.958	1.801	2.396	2.233
2	This work, AVX, PCP	1.691	1.630	2.017	1.954
	This work, AVX, PCS	1.513	1.506	1.835	1.799
3	This work, AVX, PCP	1.548	1.526	1.872	1.821
	This work, AVX, PCS	1.371	1.397	1.670	1.672
4	This work, AVX, PCP	1.471	1.485	1.762	1.768
	This work, AVX, PCS	1.310	1.363	1.609	1.637