Finite Field Arithmetic Using AVX-512 For Isogeny-Based Cryptography

Gabriell Orisaka[1], Diego F. Aranha[1,2], Julio López[1]

[1] Institute of Computing, University of Campinas, Brazil

[2]Department of Engineering, Aarhus University, Denmark

orisaka@lasca.ic.unicamp.br, dfaranha@eng.au.dk, jlopez@ic.unicamp.br

Abstract. *Isogeny-based cryptography introduces new candidates to quantum-resistant cryptographic protocols. The cost of finite field arithmetic dominates the cost of isogeny-based cryptosystems. In this work, we apply AVX-512 vector instructions to accelerate the finite field modular multiplication. We benchmark our implementation on a Skylake-X processor and discuss the applicability of our contribution and the directions for future work.*

1. Introduction

The future prospect of quantum computers poses a threat to most currently used public-key cryptosystems, such as the widely implemented schemes with security based on factoring integers or computing discrete logarithms over elliptic curves. Even though a general purpose large-scale quantum computer—a computing system capable of exploring properties of quantum-mechanics to perform operations on data—does not exist today, recent developments in this area of research have helped to motivate researchers, government, and corporate bodies to take action [Costello et al. 2016]. Recently, among the post-quantum cryptographic techniques, cryptosystems that base their security on the difficulty of finding isogenies between supersingular elliptic curves (isogeny-based cryptography) are looking promising [Jao and Feo 2011, Feo et al. 2014, Galbraith et al. 2017, Yoo et al. 2017]. They usually offer significantly smaller key sizes than other post-quantum alternatives. However, their performance can be a few orders of magnitude slower. Therefore, a number of researchers have been trying to optimize their performance [Costello et al. 2016, Azarderakhsh et al. 2016, Costello et al. 2017, Costello and Hisil 2017]. The use of AVX2 vector instructions to accelerate prime field and elliptic curve arithmetic have already been studied before [Faz-Hernández and López 2014, Faz-Hernández and López 2015]. In this work, we present implementation techniques using AVX-512 to perform efficient finite field arithmetic in order to improve isogeny-based protocols.

The paper is organized as follows. In Section 2 we briefly introduce the context for isogeny-based cryptography. Section 3 gives preliminaries on prime field arithmetic. In Section 4, we present implementation techniques using AVX-512 for finite field arithmetic. Section 5 summarizes the performance results of our implementation. Finally, Section 6 concludes the paper.

2. Isogeny-Based Cryptography

Let E_1 and E_2 be elliptic curves defined over a finite field $\mathbb{F}_q$ of characteristic p. An *isogeny* $\phi : E_1 \rightarrow E_2$ is a non-constant rational map that preserves the point at infinity $\mathcal{O}$

and is also a group homomorphism between $E_1(\overline{\mathbb{F}_q})$ and $E_2(\overline{\mathbb{F}_q})$. The *degree* of an isogeny is defined as its degree as a rational map. A curve E is considered *supersingular* if its endomorphism ring is isomorphic to an order in a quaternion algebra [Jao and Feo 2011]; moreover, all supersingular curves over finite fields of characteristic p are isomorphic to curves defined over $\mathbb{F}_{p^2}$ [Yoo et al. 2017]. Consequently, all arithmetic operations in the various cryptosystems based on isogenies between supersingular curves are usually performed over $\mathbb{F}_{p^2}$. The security of these cryptosystems is based around the hardness of the underlying assumptions, in particular, the *Computational Supersingular Isogeny* (CSSI) problem, as discussed in [Adj et al. 2018]. The authors argued that a 448-bit prime would be sufficient to offer 128 bits of security against all known classical and quantum attacks (in a realistic model of quantum computation). Motivated by this reduction in parameter size, we focus our efforts on providing an efficient implementation of finite field arithmetic for the prime $p_{434} = 2^{216} \cdot 3^{137} - 1$.

3. Prime Field Arithmetic

As noted in [Faz-Hernández and López 2015], a common approach to represent a field element in several cryptographic libraries is to use a *multi-precision representation*. In this representation, a field element $a \in \mathbb{F}_p$, in an n-bit base scheme, can be written as:

$$A(n) = \sum_{i=0}^{s-1} a_i 2^{in}, \tag{1}$$

such that $a \equiv A(n) \bmod p$ and $0 \leq a_i < 2^n$ for $n \in \mathbb{Z}^+$ and $s = \lceil \frac{\lceil \log_2 p \rceil}{n} \rceil$. A disadvantage of using this representation in an n-bit architecture is that after some arithmetic operations, the carry bits must be sequentially propagated, limiting the available parallelism. To overcome this issue, a *redundant multi-precision representation* is used, choosing a base size n smaller than the machine register size and representing each coefficient a_i with enough bits in each machine word to store the carry bits produced by arithmetic operations. Thus, a prime field element $a \in \mathbb{F}_p$ is represented by a tuple of coefficients $\mathbf{A} = \{a_0, a_1, ..., a_{s-1}\}$, where each a_i requires n-bits.

Operations like addition, subtraction, multiplication and so forth, are performed modulo p. In this preliminary work, we focus on the prime field multiplication, since it is usually the most performance-critical operation. In the case of isogeny-based key exchange, for example, an efficient modular multiplication is crucial for achieving high performance [Costello et al. 2016]. In general, prime field multiplication is performed in two parts: the integer multiplication and then the modular reduction.

Integer multiplication. Using the representation described previously, an integer multiplication can be viewed similarly to a polynomial multiplication. That is, given two prime field elements $\mathbf{A}$ and $\mathbf{B}$, the tuple $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ is computed in the following manner:

$$C(n) = \sum_{i=0}^{2s-2} c_i 2^{in} = A(n) \times B(n) = \left(\sum_{i=0}^{s-1} a_i 2^{in} \right) \left(\sum_{i=0}^{s-1} b_i 2^{in} \right) = \sum_{i=0}^{2s-2} \sum_{j=0}^{i} a_j b_{i-j} 2^{in} \tag{2}$$

Namely, let $\mathbf{A} = \{a_0, a_1, ..., a_{s-1}\}$ and $\mathbf{B} = \{b_0, b_1, ..., b_{s-1}\}$ be two integers expressed in the *redundant multi-precision representation*, the product between them will produce $A \times B = \{c_0, c_1, ..., c_{2s-2}\}$, having at most $(2s - 1)$ coefficients.

Modular Reduction. Ideally, to perform a modular reduction one would compute the remainder of a division by p, using a multi-precision division algorithm. Unfortunately, the multi-precision division is a costly operation. A more efficient technique is to perform a *Montgomery Reduction*. To reduce a number T, the technique works by adding multiples of p to cancel out the least significant bits of T, until a multiple of a constant R, known as *Montgomery* constant, is reached. By applying this method, however, the obtained result would be $TR^{-1} \bmod p$, instead of the ideally desired result $T \bmod p$. Nonetheless, this undesired term R^{-1} is only a small nuisance, since it is possible to carry out many arithmetic operations in sequence and only remove it at the end of the computation. The *Montgomery Reduction* algorithm is shown in Algorithm 1.

Algorithm 1 Montgomery Reduction

Input: Prime $p = \{p_0, p_1, ..., p_{s-1}\}$ with $p > 2$, integers $R = 2^{n \cdot s}, p' = -p^{-1} \bmod 2^n$ and $\mathbf{T} = \{t_0, t_1, ..., t_{2s-2}\}$.
Output: $TR^{-1} \bmod p$.

1: $\mathbf{A} \leftarrow \mathbf{T}$
2: **for** $i \leftarrow 0$ to $(s-1)$ **do**
3: $\quad u_i \leftarrow a_i \cdot p' \mod 2^n$
4: $\quad \mathbf{A} \leftarrow \mathbf{A} + u_i \cdot p \cdot 2^{i \cdot n}$
5: **end for**
6: $\mathbf{A} \leftarrow \mathbf{A}/2^{s \cdot n}$
7: **return A**

4. Efficient Implementation Using AVX-512

AVX-512 is a set of 512-bit SIMD instructions initially proposed by Intel in 2013 and first implemented in the Knights Landing architecture; and, more recently in 2017, in the Skylake-X architecture. Such instructions are useful to exploit the data-level parallelism present in implementations of prime field arithmetic, elliptic curve arithmetic and, ultimately, isogeny-based cryptographic software.

4.1. The AVX-512 Instruction Set

The instruction set is composed mainly by integer, floating point and mask vector instructions. Integer vector instructions in AVX-512 can pack eight 64-bit integers, or sixteen 32-bit integers within a 512-bit register, enabling twice the number of data that AVX2 can process with a single instruction [Reinders 2013]. We detail in this section the most relevant instructions used in this work, referred by a mnemonic described in Table 1.

- **Integer arithmetic.** Integer addition and subtraction ($\mathbf{ADD}$, $\mathbf{SUB}$) can now operate on eight packed 64-bit integers. The $\mathbf{MUL}$ instruction is able to compute eight products between 32-bit integers and store the eight 64-bit results in a 512-bit register.
- **Permutation.** The $\mathbf{PERM}$ instruction moves 64-bit words across the entire 512-bit register using a specified pattern. The $\mathbf{SHUF}$ instruction operates on 32-bit words, but the movement is restricted within 128-bit lanes.
- **Logical.** The $\mathbf{AND}$ instruction computes the logical AND operation on a pair of 512 bits of data, and stores the result in a destination register.

- **Shift.** Logical shift instructions are available to perform fixed and variable length displacements on packed eight 64-bit integers. The **SRAI** instruction shifts packed 64-bit integers right while shifting in sign bits.
- **Broadcast.** Using the **BCAST** instruction it is possible to replicate a 64-bit integer to all 64-bit words of the 512-bit register.

Table 1. Latency and reciprocal throughput of relevant AVX-512 instructions measured in the Skylake-X processor [Fog 2018].

Type	Mnemonic	Instruction	Latency (cycles)	Reciprocal Throughput (cycles/op)
Integer arithmetic	**ADD/SUB**	`vpaddq`/`vpsubq`	1	0.5
	MUL	`vpmuldq`	10	2
Permutation	**PERM**	`vpermq`	3	1
	SHUF	`vpshufd`	1	1
Logical	**AND**	`vpandd`	1	0.5
Shift	**SRAI**	`vpsraq`	1	0.5
Broadcast	**BCAST**	`vpbroadcastq`	3	1

4.2. AVX-512 Implementation

For our implementation of the prime field $\mathbb{F}_p$, where $p = 2^{216} \cdot 3^{137} - 1$, and its quadratic extension, we chose the *redundant multi-precision representation* with $n = 28$, since the wider multiplier available in AVX-512 is an eight packed 32-bit multiplier. That choice also leaves us with 8 bits to store the carries (after a multiplication instruction), which gives us some flexibility by minimizing the required number of carry bit propagations. However, after a few arithmetic operations a carry propagation must be performed in order to clean up the carry bits and avoid an overflow in the succeeding operation that might generate a carry bit. The performance of this operation, as well as that of the implementation of the prime field multiplication and modular reduction, are highly dependent on the manner the 64-bit words are organized in the 512-bit register. It can influence the number of AVX-512 instructions necessary to perform some prime field operation, including the type of the instructions used, which can have different execution latencies. In [Faz-Hernández and López 2015], the authors noted that AVX2 permutation instructions in the Haswell micro-architecture that move bytes across 128-bit lanes have higher latencies than permutations instructions that do not. This seems to be the result of architectural design choices in the Haswell micro-architecture that appear to persist in the Skylake-X micro-architecture and AVX-512 (see Table 1 for instruction latencies).

4-Way Interleaved Tuples. In order to restrict the movement of 64-bit words across 128-bit lanes in the 512-bit register, and to minimize the number of 64-bit word shuffling during carry bit propagation, we chose to organize the prime field elements using 4-way interleaved tuples, with the 64-bit words from each element stored vertically in the register array (a strategy similar to the one used in [Faz-Hernández and López 2015]). A tuple denoted by $\langle \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ represents the interleaving of the prime field elements $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ using eight 512-bit registers. That is, $R = \langle \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle = \{R_0, R_1, ..., R_7\}$, where each 512-bit register $R_i = [a_i, a_{8+i}, b_i, b_{8+i}, c_i, c_{8+i}, d_i, d_{8+i}]$. This representation leads to a 4-way mode of operation: every execution of our arithmetic operations will

operate on 4 different set of operands. Parallel and batch executions of isogeny-based cryptographic protocols could take advantage of this 4-way mode of operation.

Multiplication. The multiplication of two interleaved tuples $X = \langle \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ and $Y = \langle \mathbf{E}, \mathbf{F}, \mathbf{G}, \mathbf{H} \rangle$ is shown in Algorithm 2. The output of the algorithm is an interleaved tuple Z given in an intermediate state comprised of twenty-four 512-bit register, which will be reduced to its standard state of eight 512-bit register after the modular reduction. The multiplication is performed using the operand-scanning method (often referred as *schoolbook* method). In the first loop (lines 2-8), a temporary register M contains a copy of Y_i with its first, third, fifth and seventh 64-bit words copied to its second, fourth, sixth and eighth position, respectively. This task is performed efficiently using the $\mathbf{SHUF}$ instruction. Afterwards, in the inner loop, M will be multiplied by X_j and the result will be accumulated into Z_{i+j}. The second loop (lines 9-15) is similar, except that the temporary register N now holds a copy of Y_i with its second, fourth, sixth and eighth 64-bit words copied to its first, third, fifth and seventh position, respectively.

Algorithm 2 Multiplication algorithm using AVX-512

Input: Two interleaved tuples $X = \langle \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ and $Y = \langle \mathbf{E}, \mathbf{F}, \mathbf{G}, \mathbf{H} \rangle$.
Output: An interleaved tuple $Z = \langle \mathbf{I}, \mathbf{J}, \mathbf{K}, \mathbf{L} \rangle$ where $\mathbf{I} = \mathbf{A} \times \mathbf{E}$, $\mathbf{J} = \mathbf{B} \times \mathbf{F}$, $\mathbf{K} = \mathbf{C} \times \mathbf{G}$ and $\mathbf{L} = \mathbf{D} \times \mathbf{H}$.

1: $Z_i \leftarrow 0$ for $i \in \{0, 1, ..., 23\}$
2: **for** $i \leftarrow 0$ to 7 **do**
3: $\quad M \leftarrow \mathbf{SHUF}(Y_i, \texttt{0x44})$
4: $\quad$ **for** $j \leftarrow 0$ to 6 **do**
5: $\quad\quad Z_{i+j} \leftarrow \mathbf{ADD}(Z_{i+j}, \mathbf{MUL}(M, X_j))$
6: $\quad$ **end for**
7: $\quad Z_{i+7} \leftarrow \mathbf{MUL}(M, X_7)$
8: **end for**
9: **for** $i \leftarrow 0$ to 7 **do**
10: $\quad N \leftarrow \mathbf{SHUF}(Y_i, \texttt{0xEE})$
11: $\quad$ **for** $j \leftarrow 0$ to 6 **do**
12: $\quad\quad Z_{i+j+8} \leftarrow \mathbf{ADD}(Z_{i+j+8}, \mathbf{MUL}(N, X_j))$
13: $\quad$ **end for**
14: $\quad Z_{i+15} \leftarrow \mathbf{MUL}(N, X_7)$
15: **end for**
16: **return** Z

Modular Reduction. Algorithm 3 shows the *Montgomery Reduction* of an interleaved tuple X using AVX-512 instructions. The first loop (lines 2-11) corresponds to the same operations in the main loop of Algorithm 1 (lines 2-5). The second loop (lines 12-14) corresponds to the operation of the line 6 in Algorithm 1. Since that in the interleaved representation the 64-bit words are stored vertically in the registers, a division by a multiple of 2^n can be performed efficiently as a re-referencing of the register array.

Carry propagation. Carry bits might be generated after some arithmetic operation and stored in a reserved portion of the 64-bit word in the interleaved representation of the prime field element, as described in Section 3. However, after a sequence of operations it might be necessary to propagate these carry bits and clean up the space for future operations in order to avoid an overflow. This is specially true after a prime field multiplication.

Algorithm 3 Montgomery Reduction algorithm using AVX-512

Input: An interleaved tuple $X = \langle \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ in the intermediate state comprised of twenty-four 512-bit registers, and an interleaved tuple P of the prime p, i.e., $P = \langle p, p, p, p \rangle$.
Output: An interleaved tuple $Z = \langle \mathbf{I}, \mathbf{J}, \mathbf{K}, \mathbf{L} \rangle$ containing the Montgomery Reduction of the input value stored in X.

1: $L \leftarrow \mathbf{BCAST}(\texttt{0x00000000FFFFFFFF})$
2: **for** $i \leftarrow 0$ to 15 **do**
3: $\quad C \leftarrow \mathbf{SRAI}(X_i, n)$
4: $\quad X_i \leftarrow \mathbf{AND}(X_i, L)$
5: $\quad X_{i+1} \leftarrow \mathbf{ADD}(X_{i+1}, C)$
6: $\quad U \leftarrow \mathbf{SHUF}(X_i, \texttt{0x44})$
7: $\quad$ **for** $j \leftarrow 0$ to 7 **do**
8: $\quad\quad X_{i+j} \leftarrow \mathbf{ADD}(X_{i+j}, \mathbf{MUL}(U, P_j))$
9: $\quad$ **end for**
10: $\quad X_{i+8} \leftarrow \mathbf{ADD}(X_{i+8}, \texttt{0x55}, X_{i+8}, \mathbf{SHUF}(X_i, \texttt{0x4E}))$
11: **end for**
12: **for** $i \leftarrow 0$ to 7 **do**
13: $\quad Z_i \leftarrow X_{i+16}$
14: **end for**
15: **return** Z

Algorithm 4 shows the propagation of these carry bits. The main loop (lines 3-11) is executed two times. Each of these times is a pass of the carry propagation in the 512-bit register array. After this operation, every coefficient stored in the register array is guaranteed to have at most $n + 1$ bits, leaving a safe margin to process another prime field operation.

Algorithm 4 Carry bits propagation algorithm using AVX-512

Input: An interleaved tuple $X = \langle \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$.
Output: The same interleaved tuple X with the carry bits propagated in-place.

1: $L \leftarrow \mathbf{BCAST}(\texttt{0x00000000FFFFFFFF})$
2: **for** $i \leftarrow 0$ to 1 **do**
3: $\quad$ **for** $j \leftarrow 0$ to 6 **do**
4: $\quad\quad C \leftarrow \mathbf{SRAI}(X_j, n)$
5: $\quad\quad X_j \leftarrow \mathbf{AND}(X_j, L)$
6: $\quad\quad X_{j+1} \leftarrow \mathbf{ADD}(X_{j+1}, C)$
7: $\quad$ **end for**
8: $\quad C \leftarrow \mathbf{SRAI}(X_7, n)$
9: $\quad X_7 \leftarrow \mathbf{AND}(X_7, L)$
10: $\quad X_0 \leftarrow \mathbf{ADD}(X_0, \texttt{0xAA}, X_0, \mathbf{SHUF}(C, \texttt{0x4E}))$
11: **end for**

5. Preliminary Results

We benchmarked our implementation on a Skylake-X processor (Core i7-7820X) at 3.60 GHz. Intel Turbo Boost and Hyper Threading technologies were disabled. Our source code was compiled using the GNU C Compiler v7.3.1. Table 2 shows the performance of our implementation of the 4-way finite field operations using AVX-512

in comparison to the 1-way x64 implementation from [Adj et al. 2018]. In Table 3, we show the performance of the 1-way operations of the SIDH v3.0 x64 library from [Azarderakhsh et al. 2018] executed in the same conditions. Their library implements finite field arithmetic for the primes $p_{503} = 2^{159} \cdot 3^{137} - 1$ and $p_{751} = 2^{372} \cdot 3^{239} - 1$. In an efficient x64 implementation these primes can be stored into twelve and eight 64-bit words, respectively. Assuming a modular multiplication can be computed in $T_{p_{503}}$ and $T_{p_{751}}$ for the respective primes, then a rough quadratic estimation for a x64 modular multiplication in $\mathbb{F}_{p_{434}}$ would be $0.34T_{p_{751}}$ and $0.76T_{p_{503}}$, since the prime $p_{434} = 2^{216} \cdot 3^{137} - 1$ can be stored into seven 64-bit words. Our 4-way vectorized implementation using AVX-512 is currently able to provide performance improvements over both estimated values.

Table 2. Timings (in cycles) for arithmetic operations in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ for the p_{434}.

	$\mathbb{F}_p$ reduction	$\mathbb{F}_p$ multiplication	$\mathbb{F}_{p^2}$ multiplication	$\mathbb{F}_{p^2}$ squaring
[Adj et al. 2018] (1-way)	165	321	501	378
Our work (4-way)	258	456	1,269	873
Speedup factor	2.56 ×	2.81 ×	1.58 ×	1.73 ×

Table 3. Timings (in cycles) for arithmetic operations in the SIDH 3.0 library [Azarderakhsh et al. 2018].

Domain	1-way operation	p_{503} $2^{159} \cdot 3^{137} - 1$	p_{751} $2^{372} \cdot 3^{239} - 1$
$\mathbb{F}_p$	modular reduction	107	205
	multiplication	237	445
$\mathbb{F}_{p^2}$	multiplication	610	1,185
	squaring	505	959

6. Conclusion and Future Work

In this work, we applied AVX-512 vector instructions to an implementation of finite field arithmetic. Our main contribution is a vectorized implementation of the integer multiplication and modular reduction to compose the arithmetic in $\mathbb{F}_{2^{216}.3^{137}-1}$ and its quadratic extension. In this implementation, we use an interleaved representation of the finite field elements which led to a 4-way mode of operation. This mode of operation can be used to speed up parallel and batch executions of isogeny-based protocols. We compare our performance results with other state-of-the-art x64 implementations of finite field arithmetic. In a future work, we intend to provide a fast squaring algorithm using AVX-512 instructions; also, a fast $\mathbb{F}_p$ inversion algorithm and implementation for the prime $2^{216} \cdot 3^{137} - 1$; and, finally, a complete implementation of an isogeny-based cryptographic protocol using the vectorized primitives studied.

Acknowledgements. The authors would like to thank Armando Faz-Hernández for discussions related to this paper, Francisco Rodríguez Henríquez for sending us their source code for comparison and the reviewers for their comments. This work is supported in part by the Intel/FAPESP grant 14/50704-7 under project "Secure Execution of Cryptographic Algorithms".

References

Adj, G., Cervantes-Vázquez, D., Chi-Domínguez, J.-J., Menezes, A., and Rodríguez-Henríquez, F. (2018). On the cost of computing isogenies between supersingular elliptic curves. Cryptology ePrint Archive, Report 2018/313. https://eprint.iacr.org/2018/313.

Azarderakhsh, R., Campagna, M., Costello, C., Feo, L. D., Hess, B., Jalali, A., Jao, D., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Renes, J., Soukharev, V., and Urbanik, D. (2018). SIDH v3.0. https://github.com/Microsoft/PQCrypto-SIDH. Accessed: 2018-08-10.

Azarderakhsh, R., Jao, D., Kalach, K., Koziel, B., and Leonardi, C. (2016). Key compression for isogeny-based cryptosystems. In *AsiaPKC@AsiaCCS*, pages 1–10. ACM.

Costello, C. and Hisil, H. (2017). A simple and compact algorithm for SIDH with arbitrary degree isogenies. In *ASIACRYPT (2)*, volume 10625 of *LNCS*, pages 303–329. Springer.

Costello, C., Jao, D., Longa, P., Naehrig, M., Renes, J., and Urbanik, D. (2017). Efficient compression of SIDH public keys. In *EUROCRYPT (1)*, volume 10210 of *LNCS*, pages 679–706.

Costello, C., Longa, P., and Naehrig, M. (2016). Efficient algorithms for supersingular isogeny Diffie-Hellman. In *CRYPTO (1)*, volume 9814 of *LNCS*, pages 572–601. Springer.

Faz-Hernández, A. and López, J. (2015). Fast implementation of Curve25519 using AVX2. In *LATINCRYPT*, volume 9230 of *LNCS*, pages 329–345. Springer.

Faz-Hernández, A. and López, J. (2014). On software implementation of arithmetic operations on prime fields using AVX2.

Feo, L. D., Jao, D., and Plût, J. (2014). Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Mathematical Cryptology*, 8(3):209–247.

Fog, A. (2018). Software optimization resources. https://www.agner.org/optimize/. Accessed: 2018-08-10.

Galbraith, S. D., Petit, C., and Silva, J. (2017). Identification protocols and signature schemes based on supersingular isogeny problems. In *ASIACRYPT (1)*, volume 10624 of *LNCS*, pages 3–33. Springer.

Jao, D. and Feo, L. D. (2011). Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *PQCrypto*, volume 7071 of *LNCS*, pages 19–34. Springer.

Reinders, J. (2013). Intel AVX-512 Instructions. https://software.intel.com/en-us/blogs/2013/avx-512-instructions. Accessed: 2018-08-10.

Yoo, Y., Azarderakhsh, R., Jalali, A., Jao, D., and Soukharev, V. (2017). A post-quantum digital signature scheme based on supersingular isogenies. In *Financial Cryptography*, volume 10322 of *LNCS*, pages 163–181. Springer.