# High-Throughput EdDSA Verification on Intel Processors with Advanced Vector Extensions

Bowen Zhang[1], Hao Cheng[2,3,4](✉), Johann Großschädl[1], and Peter Y. A. Ryan[1]

[1] DCS and SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
bowen.zhang.002@student.uni.lu,
{johann.groszschaedl,peter.ryan}@uni.lu

[2] School of Cyber Science and Technology, Shandong University, Qingdao, China

[3] State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao, China

[4] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao, China
hao.cheng@sdu.edu.cn

**Abstract.** The Edwards-curve Digital Signature Algorithm (EdDSA) is a deterministic digital signature scheme that has recently been adopted in a range of popular security protocols. Verifying an EdDSA signature involves the computation of a double-scalar multiplication of the form $SB - hA$, which is a costly operation. The vector extensions of modern Intel processors, such as AVX2 and AVX-512, offer a variety of options to speed up double-scalar multiplication thanks to their massive SIMD-parallel processing capabilities. However, in certain application domains like fintech or e-voting, several or many EdDSA verifications have to be performed, and what counts in the end is not the execution time of one single signature verification, but how long it takes to verify a certain number of signatures. For such applications, it makes more sense to use SIMD instructions to maximize the throughput of a batch of verification operations instead of minimizing the latency of one verification. In this paper, we introduce high-throughput AVX2/AVX-512 implementations of EdDSA verification executing four (resp., eight) instances of double-scalar multiplication in a SIMD-parallel fashion, whereby each instance uses a 64-bit element of the 256-bit (resp., 512-bit) vectors. We analyze three techniques for double-scalar multiplication, one that separates the computation of $SB$ and $hA$, while the other two integrate or interleave them based on a joint-sparse form or non-adjacent form representation of the scalars $S$ and $h$. Our experiments with 256-bit AVX2 vectorization on an Intel Cascade Lake CPU show that the separate method achieves the best results and reaches a single-core throughput of 48,182 double-scalar multiplications per second, which exceeds the throughput of the currently fastest latency-optimized implementation by a factor of 1.33.

## 1 Introduction

The rise of quantum computing has prompted a global shift towards post-quantum cryptography (PQC), with NIST standardizing quantum-resistant algorithms to mitigate future threats. However, the transition to PQC is not instantaneous, and classical cryptographic algorithms like elliptic curve cryptography (ECC) remain indispensable in the interim. Hybrid approaches, such as Apple pq3 [1] and PQXDH [21], which combine ECC with PQC, are emerging as essential solutions to ensure security and smoothness of this transitional period. Among ECC-based schemes, the Edwards-curve Digital Signature Algorithm (EdDSA) stands out as a popular component in these hybrid approaches for its security, efficiency, and simplicity. EdDSA [6] can be described as a variant of the classic Schnorr signature scheme [27] with some adaptations to improve both efficiency and security. Most importantly, EdDSA replaces the multiplicative group $\mathbb{Z}_p^*$ by the additive group of points on a twisted Edwards curve [5] that is birationally-equivalent to Curve25519 [4], thereby reducing the execution time and the size of keys and signatures, respectively. The designers of EdDSA also slightly modified the procedure for signature generation so as to support *batch verification*, which allows for verifying a set of $n > 1$ signatures faster than when the conventional verification function is executed $n$ times. As reported in [6], verifying a batch of 64 EdDSA signatures on an Intel Nehalem processor is about 52% faster than computing 64 single-signature verifications separately. On the security side, the improvements include the insertion of the signer's public key in the hash computation (to address concerns about multi-target attacks) and a deterministic generation of the per-message secret scalars (as a hedge against "bad randomness"). Thanks to all these tweaks, EdDSA is a robust yet highly-efficient signature scheme that has found broad adoption in the past 15 years. Besides major security protocols like SSL/TLS, IKEv2, and SSH, EdDSA (or a variant of it) is also an integral part of various applications ranging from block chains (e.g., Monero, Tezos, Stellar[5]) over anonymity tools (e.g., Tor) to electronic voting systems.

When the message to be signed is short, the most costly computation of the signature generation is a scalar multiplication $rB$, where $B$ is a generator of an additive subgroup of large prime order $\ell$ and $r$ is in the range of $[0, \ell - 1]$. The optimized EdDSA software described in [6] performs this scalar multiplication using a fixed-base comb method [8] with a radix-16 signed-digit representation of $r$ and employs a look-up table of 256 pre-computed points. In this way, the scalar multiplication needs only 64 point additions (and exactly as many table queries) and four point doublings. On the other hand, the verification involves a more costly *Double-Scalar Multiplication (DSM)* of the form $SB - hA$, where $B$ is fixed while $A$ becomes only known at run-time. Most optimized software libraries perform this computation in an interleaved or a simultaneous fashion with "joint doublings," which roughly halves the number of point doublings to be performed in total compared to a separate computation of $SB$ and $hA$. The simplest variant of the *simultaneous technique* (in [17] referred to as "Shamir's trick") pre-computes the point $B - A$ and performs the DSM via the double-and-add method, whereby the addend is either $B$, $-A$ or $B - A$, depending on value of the bits of $S$ and $h$ at a certain position. More advanced variants aim

[5] See `https://www.susanka.eu/coins-crypto` (accessed 2025-01-20).

to reduce the number of additions by lowering the joint Hamming weight of the two scalars (e.g., through a representation in Joint Sparse Form (JSF) or some similar low-weight form [29]) and using a table containing linear combinations of $B$ and $A$, i.e., the points $\pm uB \pm vA$ for small $u$ and $v$ [17].

The *interleaving technique* to perform a DSM differs from the simultaneous technique since it does not pre-compute $B - A$ (and also no other combination of $B$ and $A$) [17]. A basic interleaved computation of $SB - hA$ uses the normal binary representation of the scalars and adds $B$ and $A$ individually, though the point doublings are still performed "jointly." More sophisticated variants of the interleaving method take advantage of a low-weight expansion of $S$ and $h$, like the width-$w$ Non-Adjacent Form (NAF), in combination with two tables, one containing multiples of $B$ and the other multiples of $A$ [17]. For example, the EdDSA implementation introduced in [6] leverages a width-5 NAF expansion of the scalars, i.e., each of the tables contains $2^{w-2} = 8$ points. However, since one point, namely $B$, is fixed, it can be beneficial to use tables of different size along with different representations of the two scalars. Most of these algorithms for comb or window-based (double-)scalar multiplication are additive variants of older methods for (multi-)exponentiation, which were widely studied in the context of classic discrete-logarithm cryptosystems [3,22]. Some exponentiation techniques can even be traced back to papers from the 1960s, e.g., [2,26].

*Vector Extensions.* Basically any modern high-performance processor architecture comes with vector instructions to increase the performance of a multitude of tasks that can take advantage of SIMD-level parallelism, most notably video processing and computer gaming. Examples of common vector instruction sets include Intel's Advanced Vector eXtensions (AVX) and successors (e.g., AVX2 and AVX-512, which support 256-bit and 512-bit vectors, respectively) and the NEON extensions for the ARM architecture. AVX2 adds 16 vector registers to the x64 architecture (`ymm0`–`ymm15`) and provides instructions for SIMD-parallel loads/stores and arithmetic operations with a granularity of bytes, half-words (16 bits), words (32 bits), as well as double-words (64 bits). For example, the AVX2 instruction `vpmuludq` performs a 4-way-parallel multiplication on packed 64-bit unsigned integers and yields four 64-bit results. On the other hand, the ordinary x64 `mul` instruction executes only a single $(64 \times 64)$-bit multiplication that produces a 128-bit result. AVX-512 further enriches the x64 architecture with 32 vector registers of a length of 512 bits (`zmm0`–`zmm31`). Accordingly, the `vpmuludq` instruction is able to execute eight $(64 \times 64 \rightarrow 64)$-bit multiplications in parallel, i.e., AVX-512 has twice the multiplication-power of AVX2.

Implementation results presented in the literature indicate that the massive parallel processing capabilities of AVX2 and AVX-512 are relatively difficult to exploit for elliptic curve cryptosystems like X25519 and Ed25519. An example of such results are the performance figures of LIB25519 [7] for the computation of a shared secret key for X25519 (i.e., variable-base scalar multiplication). The fastest vectorized AVX2 implementation contained in LIB25519 uses 10 limbs per field-element and reaches an execution time of 87,870 cycles on a Skylake CPU. On the other hand, an ordinary (unvectorized) x64 implementation has an execution time of about 133,947 Skylake cycles, i.e., the AVX2 vector engine reduces the latency by a factor of 1.52. At a first glance, this performance gain is relatively disappointing when taking into account that `vpmuludq` can execute four times more $(64 \times 64)$-bit multiplications than `mul` or `mulx`. However, there are a number of reasons for this modest speed-up, which range from differences in the representation of the field elements (i.e., 10 versus four or five limbs) to micro-architectural properties (i.e., instruction latencies and throughputs). An additional aspect that deserves attention is the fact that the number of limbs used by the AVX2 implementation (i.e., 10) can not be evenly divided by the number of 64-bit elements of a vector (i.e., four), which means that numerous AVX2 instructions are executed on vectors that are half empty. The situation becomes even worse for AVX-512 because many vectors are 75% empty and, as a consequence, only two out of the eight 64-bit elements of a vector contribute to the computation of the result. This under-utilization of vector instructions causes sub-optimal performance and also wastes a lot of energy.

*X25519 Vectorization.* Elliptic curve cryptosystems offer many options to take advantage of parallel processing. Besides the field arithmetic itself, such options do also exist for the point arithmetic (e.g., certain formulae for point addition or doubling support the parallel execution of two or more field operations) and the scalar multiplication (e.g., some scalar multiplication algorithms allow two point operations to be carried out in parallel). The actual speed-up a software developer can achieve by taking advantage of SIMD-parallel processing at the layer of the field arithmetic, point arithmetic, and scalar multiplication is often restricted by dependencies between operations that enforce a sequential execution. This explains why the speed-up factor due to vectorization (compared to conventional execution) is always below the degree of parallelism offered by the vector instructions. However, in the case of X25519, it seems the performance one can gain by increasing the parallel-processing capabilities of an execution environment is significantly below the theoretical maximum. To give a concrete example, the benchmarking results released in [18] demonstrate that vectorized implementations of X25519 do not scale particularly well when migrating from one generation of AVX to the next. While AVX-512 (theoretically) doubles the degree of parallelism versus AVX2 (since it is possible to execute operations on eight 64-bit elements instead of four), the actual reduction in execution time is much smaller, namely only 25% (74,368 versus 99,400 Skylake cycles [18]).

The parallelization bottlenecks sketched above (i.e., partially empty vectors and sequential dependencies) are highly challenging to overcome, which implies that latency-optimized implementations are not capable to utilize the massive computing power of vector engines like AVX2 in an ideal way. This observation initiated research on how to use vector engines to optimize throughput instead of latency and paved the way to vectorized implementations of cryptosystems that have the goal of minimizing the execution time of several instances of an operation instead of a single instance. For example, such throughput-optimized implementations have been described for X25519 key exchange [11] and isogeny-based elliptic curve cryptosystems [10,9]. The variable-base scalar multiplication on Curve25519 introduced in [11] reached significantly higher throughput than any of the latency-optimized implementations from [12,14,13,19,24]. There are some important differences between optimization for latency and optimization for throughput. Of course, minimizing the latency of one scalar multiplication also reduces the total execution time of several scalar multiplications, but due to the mentioned parallelization bottlenecks, optimizing for low latency entails a sub-optimal utilization of vector engines. High-throughput implementations effectively overcome these bottlenecks by executing, for example, four instances of scalar multiplication in parallel with AVX2 instructions, where each instance uses a single 64-bit element of a 256-bit vector. This approach naturally solves the problem of partially empty (or even largely empty) vectors and also reduces the issues with sequential dependencies because the vast majority of operations can now be executed in a SIMD-parallel fashion (in essence, the only operation that is not parallelizable for high throughput is table look-up).

*Our Contributions.* Previous research on throughput-optimized implementation of elliptic curve cryptography solely considered key exchange schemes such as X25519. In this paper, we introduce the first high-throughput implementation of the Ed25519 signature system, especially the verification. More precisely, we present a vectorized implementation of Ed25519 verification that can execute four instances of a DSM of the form $SB - hA$ (with a fixed point $B$) in parallel using AVX2 instructions, whereby each instance occupies a 64-bit element of the 256-bit vectors. Both scalars, as well as point $A$, can be different in each of the four instances. Such a throughput-optimized implementation of the verification has many real-world use cases. For example, it allows one to quickly check the validity of a chain of (typically three) certificates used by TLS to authenticate web servers. This validation can take advantage of a parallel execution of the DSMs needed to verify the three signatures in the certificates. Blockchains, like those used by crypto currencies, require verifications of large numbers of signatures to confirm the validity of the transactions in a block before the block can be added to the chain. What counts in the end is the time needed for all these verification operations and not for a single one. Similarly, (very) large numbers of signatures have to be verified by e-voting systems, which is much faster when the DSM implementation targets high throughput instead of low latency.

Our main contribution is a detailed analysis and comparison of algorithms for DSM to identify the best one for maximizing throughput. These algorithms include simultaneous and interleaved techniques, which are widely used by low-latency implementations, but also an uncommon approach that fully separates the computation of $SB$ and $hA$. This separated method exploits the individual arithmetic strengths of the twisted Edwards model and the Montgomery model by performing the fixed-base scalar multiplication $SB$ with a comb method on the twisted Edwards curve, but the variable-base scalar multiplication $hA$ on the birationally-equivalent Montgomery curve using a Montgomery ladder, as proposed in [16]. Our results for AVX2 show that, somewhat surprisingly, the separated method achieves the best results and outperforms the simultaneous and interleaving techniques. Consequently, DSM algorithms that are efficient in terms of latency are not (necessarily) the best option in a throughput-oriented setting, and this seems to hold also for AVX-512. We mainly focus on AVX2 in this paper as AVX2 instructions are supported by basically any modern Intel CPU, while relatively few come with AVX-512. However, we also present some results for an AVX-512 implementation executing eight instances in parallel. These results show that AVX-512 instructions almost double the throughput of the AVX2 implementation, which confirms the scalability of the separated technique for DSM. In addition, the comparison between AVX2 and AVX-512 results indicates that the implementation makes near-ideal use of the enormous parallel processing power of modern Intel CPUs. Our software is available at:

`https://github.com/zh-bw/AVXEd25519`

## 2 Background

### 2.1 Notation

This section presents the symbols used throughout the paper. For prime-field operations, we use $q$ to denote the prime defining the underlying field, and the prime field with $q$ elements is written as $\mathbb{F}_q$. The SIMD-parallel field arithmetic operates on *limb vectors* and *limb vector sets*, with the same representation as defined in [11]. In the case of AVX2, a limb vector consists of four limbs (each of a length of 29 bits), but these four limbs belong to four different field elements and not to one and the same field-element. Since we use 29-bit limbs, four field elements can be stored in nine limb vectors, called a limb vector set. A limb vector is represented by a bold lowercase letter (e.g., $\boldsymbol{v}$), and a limb vector set by a bold uppercase letter (e.g., $\boldsymbol{V}$). Furthermore, we denote a twisted Edwards curve as $E_T$, and a curve over $\mathbb{F}_q$ as $E_T(\mathbb{F}_q)$. The base point (generator) of $E_T$ is represented by $B$, its order by $\ell$, and $d$ denotes the curve parameter. An uppercase letter in script style (e.g., $\mathcal{P}$) represents a point vector set. Specifically, $\mathcal{O}$ is used for a point vector set containing only neutral elements. We use $||$ to indicate the concatenation of strings.

### 2.2 Intel AVX2 extensions

In this work, we take advantage of the 256 bit vector extensions, namely using AVX2 instructions, to enhance the throughput of our software. Intel's Advanced Vector eXtension 2 (AVX2) is an instruction set extension designed for the SIMD parallel computing paradigm. It was first introduced with the Haswell microarchitecture in 2013, and is now supported by most of the main stream processors. AVX2 is built upon its original AVX, by expanding the width of the vector to 256-bit, and increasing the operand number from two to three in instruction format. AVX2 has sixteen 256-bit vector registers (`ymm0`-`ymm15`) and hundreds of vector instructions. We test our software on an Intel Cascade Lake processor. As a super-scalar processor, Cascade Lake follows an out-of-order fashion when assigning the micro-operations decoded from the AVX instructions to different execution ports. The Cascade Lake has a very identical architecture to its predecessor Skylake [30], which means it has eight execution ports (port 0 to 7). Ports 0, 1, and 5 can handle vector instructions, while ports 2, 4, 5, and 7 are used for memory access instructions. Branching and other instructions are assigned to port 6. Taking some arithmetic and logical operation instructions as an example, we summarize their latency, CPI, and execution ports[6] in Table 1. To be specific, the registers of three operands in use for AVX2 instructions are `ymm, ymm, ymm`.

As mentioned in the last section, we will describe the implementation details of our throughput-optimized EdDSA software using AVX2 as a case study because AVX2 is more widely available than AVX-512. However, we also developed a prototype for AVX-512, which was relatively simple since AVX2 and AVX-512 are very similar when optimizing for high throughput. As will be shown in Section 4, the throughput almost doubled when we migrated our software from AVX2 to AVX-512.

**Table 1.** Execution details of example instructions.

| Operation | Instruction | Latency | CPI | Ports |
|---|---|---|---|---|
| ADD | `vpaddq` | 1 | 0.33 | 0, 1, and 5 |
| MUL | `vpmuludq` | 5 | 0.50 | 0 and 1 |
| AND | `vpand` | 1 | 0.33 | 0, 1, and 5 |
| OR | `vpor` | 1 | 0.33 | 0, 1, and 5 |
| XOR | `vpxor` | 1 | 0.33 | 0, 1, and 5 |

## 2.3 Ed25519

Ed25519 is an instance of EdDSA based on an elliptic curve known as Edwards25519. It is defined by the formulae:

$$x^2 + y^2 = 1 + dx^2y^2,$$

where $d$ is a specific constant with $d = -121665/121666$, and all operations are performed over the prime field $\mathbb{F}_q$ with $q = 2^{255} - 19$. Ed25519 plays a key role in transport layer protocols such as TLS and was included in the FIPS185-6 Digital Signature Standard by NIST in 2023. In the following, we provide a brief overview of the main algorithms of Ed25519. For more details, we refer the reader to [6].

**Point encoding and decoding.** All integers are coded in the little-endian format. A point $P = (x, y)$ on $E_T(\mathbb{F}_q)$ is encoded as follows. First, clear the

[6] See `https://uops.info/table.html`

most significant bit of the $y$-coordinate and encode it as a 256-bit string in little-endian format. Then, copy the least significant bit of the $x$-coordinate to the place of the most significant bit of the final bit-string. The point encoding can be defined formally as:

$$Encode(P) = (x \, mod \, 2) \, || \, y.$$

The point decoding takes a 256-bit string $s$ as input. First, it extracts the $y$-coordinate from $s$ with $y = (s_{255}, ..., s_0)_2$. Then, it recovers the $x$-coordinate by calculating $x = \sqrt{(y^2 - 1)/(dy^2 + 1)}$. If $s_{255} \neq x$ mod 2 then set $x = q - x$. The point decoding fails if either $y \notin \mathbb{F}_q$ or the square root does not exist. The point decoding can be defined formally as:

$$Decode(s) = (x, y).$$

**Key generation.** The private key and public key are generated using Algorithm 1. The execution time of key generation is dominated by the fixed-base scalar multiplication $sB$.

```
Algorithm 1 Ed25519 key generation
Output:$(sk, pk)$.
1: $sk \leftarrow \{0,1\}^{256}$                         ▷ Generate a random 256-bit integer
2: $\textsc{Hash}(sk) = (h_{511}, ..., h_0)_2$
3: $s = 2^{254} + \sum_{3 \leq i \leq 253} 2^i h_i$
4: $pk = Encode(sB)$
5: return $(sk, pk)$
```

**Signature generation.** The input to the signing procedure is the signer's key pair $(sk, pk)$, and a message $M$ of arbitrary size. The signature is calculated as shown in Algorithm 2. The execution time of signature generation is dominated by the fixed-base scalar multiplication $rB$.

```
Algorithm 2 Ed25519 signature generation
Input:$(sk, pk)$, message $M$.
Output:$(R \,||\, S)$.
1: $\textsc{Hash}(sk) = (h_{511}, ..., h_0)_2$
2: $r = \textsc{Hash}((h_{511}, ..., h_{256})_2 \,||\, pk \,||\, M) \bmod \ell$
3: $R = Encode(rB)$
4: $S = r + \textsc{Hash}(R \,||\, pk \,||\, M)s \bmod \ell$        ▷ $s$ is obtained at key generation
5: return $(R \,||\, S)$
```

**Signature verification.** To verify a signature $(R \,||\, S)$ on message $M$, given signer's public key $pk$, the signature verification proceeds as specified in Algorithm 3. The verification time primarily depends on the DSM, i.e., the computation of $SB - hA$, for which we discuss some efficient implementations in Section 3.

**Algorithm 3** Ed25519 signature verification

```
Input: $pk$, message $M$, $(R \,||\, S)$.
Output: $Accept$ or $Reject$.
 1: $R' = Decode(R)$
 2: $A = Decode(pk)$
 3: if any $Decode$ fails then
 4:     return $Reject$
 5: end if
 6: $h = \textsc{Hash}(R \,||\, pk \,||M) \, mod \, \ell$
 7: if $R' = SB - hA$ then
 8:     return $Accept$
 9: end if
10: return $Reject$
```

## 3 Implementation

When assuming that the message $M$ to be signed is not excessively long, the most time-consuming operation in EdDSA verification is the DSM of the form $R = SB - hA$, which contains a fixed-base scalar multiplication $SB$ and a variable-base scalar multiplication $hA$. From a high-level perspective, there exist two main techniques for computing a DSM, namely (i) the *combined* approach that computes $SB$ and $hA$ with joint doublings (two variants of this approach, namely the simultaneous and interleaving technique, were sketched in Sect. 1), and (ii) the *separate* approach that computes $SB$ and $hA$ sequentially and independently of each other. The former approach usually takes advantage of a special representations of the scalars to reduce the number of point operations, e.g., Non-Adjacent Form (NAF) or Joint-Sparse Form (JSF), whereas the latter utilizes the birational equivalence between the twisted Edwards model and the Montgomery model.

In this section, we show how the Intel AVX2 instructions can be used to execute four DSM instances in parallel, each with different points and scalars, to improve the throughput of EdDSA verification, whereby the NAF-based, JSF-based, and separate approaches are considered. Our throughput-optimized software enables a "batch execution" of EdDSA verification since four EdDSA signatures are verified simultaneously (the number of parallel instances increases to eight in the case of AVX-512). Therefore, our throughput-optimized implementation is somewhat similar to the (algorithmic) batch verification described in [6]. However, this batch verification has two disadvantages compared to our approach. First, batch verification is only effective when the size of the batch is relatively large (in Sect. 1 we mentioned a speed-up of 52% for a batch size of 64), whereas our implementation is able to boost throughput with a batch of only four (AVX2) or eight (AVX-512) signatures. Second, when the verification of a batch fails, each signatures still needs to be verified separately. In addition to describing throughput-optimization with AVX2 (resp., AVX-512), we also study how one can further improve the throughput by applying multiprocessing.

### 3.1 Limb Vector Sets

Similar to [11], the data structure we operate on for the field arithmetic is a $(4 \times 1)$-way *limb vector set*. For each field element, we employ a 29-bit-per-limb (i.e., radix-$2^{29}$) representation, meaning nine limbs are needed in total. Given four field elements $a$, $b$, $c$, and $d$, the $(4 \times 1)$-way limb vector $\boldsymbol{V}$ is defined formally as

$$\begin{aligned}\boldsymbol{V} = [a, b, c, d] &= [\sum_{i=0}^{8} 2^{29i} a_i, \sum_{i=0}^{8} 2^{29i} b_i, \sum_{i=0}^{8} 2^{29i} c_i, \sum_{i=0}^{8} 2^{29i} d_i] \\ &= \sum_{i=0}^{8} 2^{29i} [a_i, b_i, c_i, d_i] = \sum_{i=0}^{8} 2^{29i} \boldsymbol{v}_i,\end{aligned}$$

where $\boldsymbol{v}_i = [a_i, b_i, c_i, d_i]$ is called a limb vector. As explained in the previous section, in the case of AVX2 a limb vector set consists of nine limb vectors, each containing four limbs from four different field elements. Analogous to [11], a *point vector set* is then used for the elliptic curve arithmetic. Given four points $A$, $B$, $C$, and $D$ in affine coordinates, the point vector set $\mathcal{P}$ is defined formally as

$$\begin{aligned}\mathcal{P} &= [A, B, C, D] \\ &= [(x_A, y_A), (x_B, y_B), (x_C, y_C), (x_D, y_D)] \\ &= ([x_A, x_B, x_C, x_D], [y_A, y_B, y_C, y_D]) \\ &= (x_{\mathcal{P}}, y_{\mathcal{P}}),\end{aligned}$$

where the limb vector set $x_{\mathcal{P}}$ (resp. $y_{\mathcal{P}}$) represents the $x$-coordinate (resp. $y$-coordinate) of four points. For points in projective coordinates, the corresponding point vector set is $\mathcal{P} = (X_{\mathcal{P}}, Y_{\mathcal{P}}, Z_{\mathcal{P}})$. We adopt the formulas for point addition/doubling from [20].

### 3.2 Implementation using NAF-based approach

The first implementation option uses the $\omega$-NAF representation [28] for the scalars, applied with a so-called interleaving technique [15]. This NAF-based approach relies on two look-up tables, one for the fixed point $B$ and the other for the variable point $A$, using a coefficient of the $\omega$-NAF expansion (of the scalar) as the input for a table. Taking scalar $h$ and point $A$ as an example, the $\omega_A$-NAF expansion of $h$ is $h = \sum_{i=0}^{l-1} h_i 2^i$, where each coefficient $|h_i| < 2^{\omega_A - 1}$,

```c
#include <immintrin.h>
/* 4-way arithmetic operations on packed 64-bit integers */
#define VSUB(X, Y)          _mm256_sub_epi64(X, Y)
#define VABS32(X)           _mm256_abs_epi32(X)
/* 4-way bitwise logical operations on packed 64-bit integers */
#define VXOR(X, Y)          _mm256_xor_si256(X, Y)
#define VAND(X, Y)          _mm256_and_si256(X, Y)
#define VOR(X, Y)           _mm256_or_si256(X, Y)
#define VSHR(X, Y)          _mm256_srli_epi64(X, Y)
/* Other 4-way operations on packed 64-bit integers */
#define VSET164(X)          _mm256_set1_epi64(X)
#define VSET64(W, X, Y, Z) _mm256_set_epi64x(W, X, Y, Z)
#define VZERO               _mm256_setzero_si256()
/* Structure for projective point representation */
typedef struct projective_point {
    __m256i x[NWORDS];
    __m256i y[NWORDS];
    __m256i z[NWORDS];
} ProPoint;
```

**Listing 1.** Defined macros for AVX intrinsics and structure for point representation.

```c
void table_query(ProPoint *r, ProPoint *table, __m256i b)
{
  const __m256i babs = VABS32(b), one = VSET164(1);
  __m256i xP[9], yP[9], zP[9], t[9], temp, bsign, bmask;
  uint32_t xcoor[4][9], ycoor[4][9], zcoor[4][9], index[4];
  int i, j;

  /* Extract the coefficients from a vector */
  for (i = 0; i < 4; i++) index[i] = (((uint32_t*) &babs)[i*2] + 1) / 2;
  /* Start table query*/
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 9; j++) {
      xcoor[i][j] = ((uint32_t*) &table[index[i]].x[j])[i*2];
      ycoor[i][j] = ((uint32_t*) &table[index[i]].y[j])[i*2];
      zcoor[i][j] = ((uint32_t*) &table[index[i]].z[j])[i*2];
    }
  }
  /* Form the point vector set with query results */
  for (i = 0; i < 9; i++) {
    xP[i] = VSET64(xcoor[3][i], xcoor[2][i], xcoor[1][i], xcoor[0][i]);
    yP[i] = VSET64(ycoor[3][i], ycoor[2][i], ycoor[1][i], ycoor[0][i]);
    zP[i] = VSET64(zcoor[3][i], zcoor[2][i], zcoor[1][i], zcoor[0][i]);
  }
  /* Conditional negation */
  bsign = VSHR(b, 31);
  bmask = VSUB(zero, bsign);
  for (i = 0; i < 9; i++) {
    temp  = VAND(VXOR(xP[i], yP[i]), bmask);
    xP[i] = VXOR(xP[i], temp);
    yP[i] = VXOR(yP[i], temp);
  }
  mpi29_copy_avx2(t, zP);
  mpi29_gfp_neg_avx2(t);
  mpi29_cswap_avx2(zP, t, bsign);
  /* Copy final result to the point vector set */
  mpi29_copy_avx2(r->x, xP);
  mpi29_copy_avx2(r->y, yP);
  mpi29_copy_avx2(r->z, zP);
}
```

**Listing 2.** Simplified C source code of batch table query.

**Algorithm 4** Batch DSM implementation using NAF-based approach.

```
Input: Fixed base point vector set $\mathcal{B}$, variable base point vector set $\mathcal{A}$, scalar sets $\boldsymbol{S}$
    and $\boldsymbol{H}$, a precomputed table $\mathcal{T}_\mathcal{B}$ for $\mathcal{B}$, and window sizes $\omega_\mathcal{A}$ and $\omega_\mathcal{B}$
Output: $\mathcal{R} = \boldsymbol{S}\mathcal{B} - \boldsymbol{H}\mathcal{A}$
 1: $\mathcal{T}_\mathcal{A} \leftarrow$ TableComputation($\mathcal{A}$)        ▷ Compute the look-up table for $\mathcal{A}$
 2: $\boldsymbol{S'} \leftarrow \omega_\mathcal{B}$-NAF($\boldsymbol{S}$), $\boldsymbol{H'} \leftarrow \omega_\mathcal{A}$-NAF($\boldsymbol{H}$)
 3: $t \leftarrow$ MaxLength($\boldsymbol{S'}, \boldsymbol{H'}$)
 4: $\mathcal{R} \leftarrow \mathcal{O}$, $\mathcal{P} \leftarrow \mathcal{O}$
 5: for $i$ from $t-1$ to 0 by 1 do
 6:     $\mathcal{R} \leftarrow$ PointDoubling($\mathcal{R}$)
 7:     if $\boldsymbol{S'_i} \neq \mathbf{0}$ then
 8:         $\mathcal{P} \leftarrow$ TableQuery($\mathcal{T}_\mathcal{B}, \lfloor \frac{|\boldsymbol{S'_i}|+1}{2} \rfloor$)        ▷ Refer to Listing 2
 9:         $\mathcal{R} \leftarrow$ PointAddition($\mathcal{R}, \mathcal{P}$)
10:     end if
11:     if $\boldsymbol{H'_i} \neq \mathbf{0}$ then
12:         $\mathcal{P} \leftarrow$ TableQuery($\mathcal{T}_\mathcal{A}, \lfloor \frac{|\boldsymbol{H'_i}|+1}{2} \rfloor$)        ▷ Refer to Listing 2
13:         $\mathcal{R} \leftarrow$ PointAddition($\mathcal{R}, \mathcal{P}$)
14:     end if
15: end for
16: return $\mathcal{R}$
```

i.e., the range of the coefficients and the size of the look-up table is determined by $\omega_A$. In addition, in this expansion, the most significant coefficient $h_{l-1}$ is non-zero, and all non-zero coefficients are odd. The associated look-up table is composed of the points

$$\left\{ A_i = (2i+1)A \;\middle|\; 0 \leq i < 2^{\omega_A - 2} \right\}.$$

Note the specific parameters $\omega_B = 7$ and $\omega_A = 5$ are used in state-of-the-art NAF-based implementations, e.g., [13] and [7], and therefore we adopt the same setting for our implementation.

Algorithm 4 describes our batch DSM implementation using the NAF-based approach. After getting the NAF representations of the scalars, we first identify the longest NAF expansion among all instances and pad others with zeros on the MSB side. Then, the main loop (line 5 to 15) starts, in which each iteration processes a single coefficient of each expansion. Note that, if a coefficient is zero in an instance, the related table query and point addition could be skipped for this instance. However, due to the SIMD pattern, we will skip the computation only if *all* coefficients in the four (resp., eight) instances are zero. A slightly simplified version of the batch table-query is shown in Listing 2, which is implemented using the AVX2 intrinsics and projective point representation detailed in Listing 1. The table query loads points in extended affine coordinates[7], and to handle the case that some coefficients in a vector are zero, the point at infinity (again in extended affine coordinates) is included in the look-up table.

[7] An extended affine point has the form $(u, v, w)$, where $u = (x+y)/2,\ v = (y-x)/2,\ w = dxy$, and $d$ is the parameter of the twisted Edwards curve [6].

**Algorithm 5** Batch DSM implementation using JSF-based approach.

```
Input: Fixed base point vector set $\mathcal{B}$, variable base point vector set $\mathcal{A}$, and two scalar
    sets $\boldsymbol{S}$ and $\boldsymbol{H}$
Output: $\mathcal{R} = \boldsymbol{S}\mathcal{B} - \boldsymbol{H}\mathcal{A}$
 1: $(\boldsymbol{S'}, \boldsymbol{H'}) \leftarrow$ JointSparseForm$(\boldsymbol{S}, \boldsymbol{H})$
 2: $t \leftarrow \max(\boldsymbol{S'}, \boldsymbol{H'})$
 3: $\mathcal{T} \leftarrow [-\mathcal{A}, \mathcal{B}+\mathcal{A}, \mathcal{B}, \mathcal{B}-\mathcal{A}]$          ▷ Compute the look-up table
 4: $\mathcal{R} \leftarrow \mathcal{O}$, $\mathcal{P} \leftarrow \mathcal{O}$
 5: for $i$ from $t-1$ to 0 by 1 do
 6:     $\mathcal{R} \leftarrow$ PointDoubling$(\mathcal{R})$
 7:     $\boldsymbol{D_i} \leftarrow 3\boldsymbol{S'_i} + \boldsymbol{H'_i}$
 8:     if $\boldsymbol{D_i} \neq \boldsymbol{0}$ then
 9:         $\mathcal{P} \leftarrow$ TableQuery$(\mathcal{T}, \boldsymbol{D_i})$          ▷ Refer to Listing 2
10:         $\mathcal{R} \leftarrow$ PointAddition$(\mathcal{R}, \mathcal{P})$
11:     end if
12: end for
13: return $\mathcal{R}$
```

### 3.3 Implementation using JSF-based approach

The second implementation option uses the JSF [29] representation for scalars, and needs only one joint look-up table containing besides the two points $B$ and $-A$ also their sum and difference. The JSF expansion uses a signed coefficient set $\{-1, 0, 1\}$ to represent two scalars $s$ and $h$ in a matrix of the form:

$$\begin{pmatrix} s_{l-1} & s_{l-2} & \cdots & s_0 \\ h_{l-1} & h_{l-1} & \cdots & h_0 \end{pmatrix},$$

where $s = \sum_{i=0}^{l-1} s_i 2^i$ and $h = \sum_{i=0}^{l-1} h_i 2^i$. Thus, the corresponding look-up table contains four entries, namely $[-A, B+A, B, B-A]$. The JSF of two scalars has, in the average case, a joint Hamming weight of 0.5 and, therefore, reduces the number of point additions by 25% compared t o a conventional implementation. The idea of reducing the joint Hamming weight of two scalars can be extended in two directions. First, one can extend the digit set from $\{0, \pm1\}$ to $\{0, \pm1, \pm3\}$ or $\{0, \pm1, \pm3, \pm5\}$ to further reduce the Joint Hamming weight (e.g., from 0.5 to roughly 0.38 when $\pm3$ is included in the digit set) at the expense of a larger table. A second direction is to generalize the JSF to support more than two scalars, e.g., to minimize the joint Hamming weight of eight scalars, which is relevant in the case when four DSMs are computed in parallel.

Algorithm 5 presents our batch DSM implementation using JSF-based approach. Compared to the NAF-based implementation (Algorithm 4), the main difference is that the processed coefficient is now a linear combination $d_i$ of a column $(s_i, h_i)$ from the matrix, instead of the JSF expansion itself. On the other hand, a similarity is that we will skip the table query and point addition in an iteration only if all coefficients being processed are zero.

### 3.4 Implementation using the separate approach

The third implementation option is computing two scalar multiplications $SB$ and $hA$ separately. Cheng et al. [11] present a batch implementation of fixed-base scalar multiplication used in X25519 key generation[8], whereby the table query operation loads all possible points (related to a scalar nibble) from a pre-computed table and employs a mask to extract the right one to ensure constant-time execution. Their implementation prevents cache-timing attacks at the cost of additional operations, which is not necessary for our implementation since the two scalars used in EdDSA verification are not secret. Our implementation therefore selects the corresponding points straightforwardly, and thus reduces the execution time[9].

**Algorithm 6** Scalar multiplication on TE curve using Montgomery ladder [16]

```
Input: Twisted Edwards curve $E_T$ over $\mathbb{F}_q$ of cardinality $h\ell$ where $\ell$ is prime, rational
    point $P = (x, y) \in E_T(\mathbb{F}_q)$ with $\mathrm{ord}(P) \geq \ell$, scalar $k \in [0, \ell - 1]$.
Output: Point $Q = kP$ in projective coordinates.
1: if $k = 0$ then return $(0 : 1 : 1)$
2: if $k = \ell - 1$ then return $(-x : y : 1)$
3: $P_m \leftarrow \textsc{TedToMon}(P)$
4: $(Q_1, Q_2) \leftarrow \textsc{MonLadder}(P_m)$
5: $Q_r \leftarrow \textsc{RecoveryY}(Q_1, Q_2, P_m)$
6: $Q \leftarrow \textsc{MonToTed}(Q_r)$
7: return $Q$
```

The basic idea is to use the twisted-Edwards model of the curve for the fixed-base scalar multiplication $SB$ and the birationally-equivalent Montgomery model for the variable-base scalar multiplication $hA$. Algorithm 6 describes the the computation of $hA$ more formally. We compute $hA$ using the Montgomery ladder on a Montgomery curve that is birationally-equivalent to the twisted Edwards curve, which requires a mapping of points from one curve model to the other. These mappings utilize the point conversion formula in [5]. The mapping from twisted Edwards to Montgomery involves a conversion from projective to affine coordinates (so that we can use the conventional Montgomery ladder), for which our implementation adopts a simultaneous inversion technique [23]. Given a point $P' = (X_{P'}, Y_{P'}, Z_{P'})$ in projective coordinates on the twisted Edwards curve, the corresponding point $P = (X_P, Y_P, Z_P)$ in projective coordinates on the birationally-equivalent Montgomery curve (i.e., constant $a = 486662$) can be

[8] From a high-level perspective, this implementation is a SIMD-parallel variant of the fixed-base scalar multiplication in [6], which, as mentioned in Sect. 1, uses a look-up table with 256 points and executes 64 point additions and four doublings.

[9] One should switch back to constant-time table query when processing sensitive values.

```
void ted_mul_varbase(ProPoint *r, ProPoint *p, const __m512i *k)
{
  ProPoint h, q1, q2, q3;
  AffPoint s;
  __m256i t[9];
  /* Convert point on Ed25519 to Curve25519 */
  conv_ted_to_mon(&h, p);
  /* Obtain the affine coordinate of the input point on Curve25519 */
  mpi29_gfp_inv_4x1w(t, h.z);
  mpi29_gfp_mul_4x1w(s.x, t, h.x);
  mpi29_gfp_mul_4x1w(s.y, t, h.y);
  /* Perform variable base multiplication on Curve25519 */
  mon_mul_varbase(&q1, &q2, k, s.x);
  /* Perform point recovery on Curve25519 */
  point_recovery(&q3, &s, &q1, &q2);
  /* Convert point on Curve25519 to Ed25519 */
  conv_mon_to_ted(r, &q3);
}
```

**Listing 3.** Simplified code of $(4 \times 1)$-way variable-base multiplication

computed as shown on the left side of the formulae below

$$\begin{aligned}
X_P &= (Z_{P'} + Y_{P'})X_{P'} & X_{P'} &= \textsc{ModSqrt}(-a)X_P Y_P (X_P + Z_P)\\
Y_P &= \textsc{ModSqrt}(-a)(Z_{P'} + Y_{P'})Z_{P'} & Y_{P'} &= (X_P - Z_P)Y_P Z_P\\
Z_P &= (Z_{P'} - Y_{P'})X_{P'} & Z_{P'} &= Y_P Z_P (X_P + Z_P).
\end{aligned}$$

The mapping in the other direction can use the formulae on the right side. After the mapping of point $A$, we take advantage of the Montgomery ladder [23] to compute the scalar multiplication $kP$, which operates on only the affine $x$-coordinate of the point $P$, i.e., $X_P$ and $Z_P$. Then, to recover the affine $y$-coordinate of the point $P$ needed by EdDSA, we use the *y recovery* formula from [25]. Given the two points $Q_1 = kP = (X_1, Z_1)$ and $Q_2 = (k+1)P = (X_2, Z_2)$ output from Montgomery ladder and $P = (x_P, y_P)$, we can recover the full projective coordinate $(X_{rec}, Y_{rec}, Z_{rec})$ on Curve25519 by:

$$\begin{aligned}
X_{\text{rec}} &= 2By_P Z_1 Z_2 X_1\\
Y_{\text{rec}} &= Z_2[(X_1 + x_P Z_1 + 2aZ_1)(X_1 + x_P Z_1)] - 2aZ_1^2 - (X_1 - x_P Z_1)^2 X_2\\
Z_{\text{rec}} &= 2By_P Z_1 Z_2 Z_1.
\end{aligned}$$

Finally, the recovered point is converted back to its corresponding point on the twisted Edwards curve. The associated code is shown in Listing 3.

While the basic principle is relatively simple, one has to pay attention to certain corner cases in the point mappings and the recovery of the $y$-coordinate, which can fail the implementation. All these corner cases were analyzed in [16]. In short, encountering these corner cases can be avoided when we insist that the scalar is fully reduced (i.e., in in the range $[0, \ell - 1]$) and the point $A$ does not have low order. These are exactly t he requirements imposed by widely-used software libraries like LibSodium (see [16] for details).

```
void multiprocessing_example()
{
  /* Set thread number to 3 */
  omp_set_num_threads(3);
  /* Start the multiprocessing implementation */
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
        double_scalar_mul_4x1w();
      #pragma omp section
        double_scalar_mul_4x1w();
      #pragma omp section
        double_scalar_mul_4x1w();
    }
  }
}
```

**Listing 4.** Example code of a multiprocessing implementation of batch DSM using three threads.

### 3.5 Applying multiprocessing

To further improve software throughput, we take advantage of thread-level parallelism in addition to the parallelism offered by the vector instructions, for which we use OpenMP[10]. The provided directive `#pragma omp parallel` allows the user to distribute a computation across multiple threads. Since in the batch implementation each instance of DSA is independent, which implies that no synchronization between threads is actually needed, applying multiprocessing to our software is straightforward. To be specific, we simply use the OpenMP API directives `sections` and `section` to allocate a single $(4 \times 1)$-way DSM to each thread. Taking a multiprocessing implementation using three threads as an example, the associated code snippet is shown in Listing 4.

## 4 Evaluation

To measure the performance of our software, we use an Intel Xeon W-2245 Cascade Lake CPU, which is clocked at 3.9 GHz. Both Turbo Boost and Hyper-Threading are disabled during the measurements. Our source code is compiled using Clang version 14.0.0 with the optimization flag[11] `-O2`, to ensure alignment with other benchmark implementations [13,7]. The source code of the field-arithmetic operations was taken from the implementation described in [11], for which the source code is linked in the paper.

We benchmarked our three different DSM implementations described in Section 3 on said CPU, and present the measured results in Table 2. The separate approach clearly turns out to be the most efficient implementation option for

[10] `https://www.openmp.org`

[11] The optimization level does not have a significant impact on the results in our experiment.

**Table 2.** Latency and throughput of our AVX2 software on Intel Xeon W-2245 Cascade Lake CPU (latency is the execution time of four parallel instances).

| DSM implementation | Latency | Throughput |
|---|---|---|
| NAF-based approach | 494,376 cycles | 31,654 ops/sec |
| JSF-based approach | 502,868 cycles | 31,117 ops/sec |
| separate approach | 324,772 cycles | 48,182 ops/sec |

**Table 3.** Probability of skipping the table query and point addition in a single iteration.

| Representation | #Instances | Probability |
|---|---|---|
| 5-NAF | 1 | 83.6% |
|  | 4 | 48.7% |
| 7-NAF | 1 | 87.7% |
|  | 4 | 59.0% |
| JSF | 1 | 49.5% |
|  | 4 | 6.2% |

batch DSM, showing an improvement of the throughput by more than 52% compared to the other two options. A major reason is that the execution time for the separate approach does not depend on the representation of the scalars. In the NAF- and JSF-based implementations, the execution time is affected by the "density" of zero coefficients in their scalar representations since they allow one to skip operations (i.e., table query and point addition). Most latency-optimized implementations (e.g., [13,7,16]) adopt either a NAF- or JSF-based approach because it can yield fewer table lookups and point additions. However, the probability of skipping operations in NAF- and JSF-based implementations is much lower in a batch implementation, as shown more concretely in Table 3. The probability data are obtained from the experiments, where for each case, we examined one million random 255-bit integers (i.e., scalars). Per Table 3, the table query and point addition are skipped with a high probability in the single-instance case, but becomes much lower when four-instance are computed in parallel, as these operations will be skipped only if *all* the processed coefficients of the batch are zero. Furthermore, this issue will be even amplified if a more powerful vector extension is used, such as AVX-512, which allows one to compute eight DSMs in parallel instead of four. As a result, the separate method is a better implementation option for batch DSM software since it is completely unaffected by this issue.

We also carried out some experiments with variants of the JSF that try to minimize the joint Hamming weight of eight scalars, but the probability for skipping a table-query and point addition was always below 10%. Since this is only a marginal improvement over the conventional JSF representation, we did not consider it in the implementation described below. However, we remark that

**Table 4.** Comparison between our implementation (based on separate approach) and other AVX2 implementations of DSM on an Intel Xeon W-2245 Cascade Lake CPU, and the most efficient latency-optimized implementation is used as the baseline for throughput comparison.

| Reference | Latency | #Instances | Throughput | Ratio |
|---|---|---|---|---|
| [13] | 142,119 cycles | 1 | 27,526 ops/sec | 0.76 |
| [7] | 108,253 cycles | 1 | 36,138 ops/sec | 1.00 |
| This work | 324,772 cycles | 4 | 48,182 ops/sec | 1.33 |

**Table 5.** Latency and throughput of our separate-approach-based DSM implementation using AVX2 on an Intel Xeon W-2245 Cascade Lake CPU, where the multiprocessing is applied.

| #Thread | Latency | #Instances | Throughput | Ratio |
|---|---|---|---|---|
| 1 | 324,772 cycles | 4 | 48,182 ops/sec | 1.00× |
| 2 | 328,932 cycles | 8 | 95,173 ops/sec | 1.98× |
| 4 | 328,790 cycles | 16 | 190,400 ops/sec | 3.95× |
| 8 | 330,135 cycles | 32 | 379,251 ops/sec | 7.87× |
| 16 | 678,912 cycles | 64 | 368,843 ops/sec | 7.66× |

the throughput for the JSF-based implementation could be further improved by a few percent when an advanced JSF variant is used.

Table 4 shows the comparison between our most efficient implementation, i.e., the implementation based on the separate approach, and the currently best latency-optimized AVX2 implementations of DSM on Ed25519. We compiled the implementation of [13] with the same Clang version and `-O2` flag, whereas the implementation of [7] was compiled with GCC 11.4.0 and `-O2` flag since the GCC executable achieved better performance for this implementation than Clang. Notably, both [13] and [7] implementations use the NAF-based approach where $\omega_B = 7$ and $\omega_A = 5$ (as mentioned in Section 3.2). In Table 4, the ratio is calculated by comparing the throughput between the baseline and the targeted implementation, which indicates that our software achieves a 33% higher throughput than the best latency-optimized implementation.

Table 5 and Figure 1 present the performance data of our AVX2 implementation when multiple threads are used to further improve the software throughput, where the throughput improvement ratio increases linearly until eight threads. Beyond eight threads, the improvement diminishes and plateaus at a certain level. We also use the Intel Vtune Profiler[14] to investigate the logical core usage at different thread numbers, which shows the usage of logical cores matches the

[13] This is obtained by dividing the overall latency by the number of instances, entry for latency is $10^3$ cycles/# inst., entry for throughput is $10^3$ ops/sec.

[14] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html`

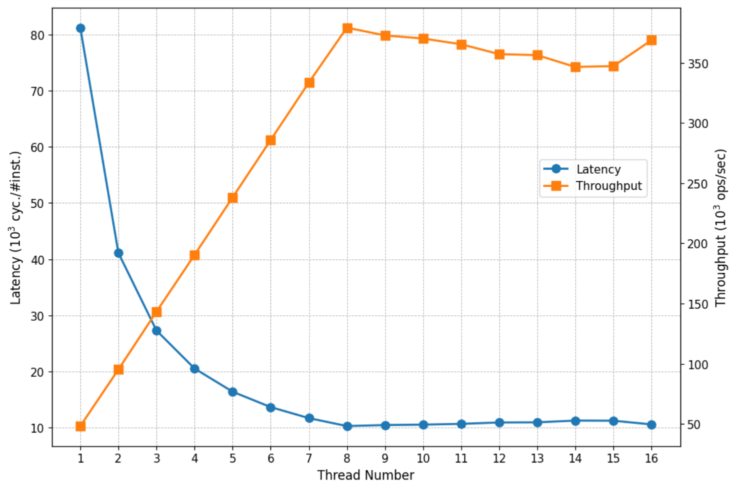

**Fig. 1.** Latency[13]and throughput of our separate-approach-based DSM implementation using AVX2 on an Intel Xeon W-2245 Cascade Lake CPU, where the different numbers of threads are enabled.

thread number when the number is below eight. Once the thread number exceeds eight, the logical core usage stabilizes at approximately 7.8 out of 16 cores. The reason is that there are eight physical cores and 16 logical cores on an Intel Xeon W-2245 CPU, with one set of AVX units per physical core. As a result, the overhead caused by the thread context switching will be introduced when the enabled thread number exceeds the actual physical core number. Therefore, the optimal thread number for our parallel implementation equals the number of physical cores of the CPU, which is eight on Intel Xeon W-2245 CPU. In this setting, the throughput is $379,251$ DSMs per second, as shown in Table 5.

Finally, we carried out some experiments with an AVX-512 implementation that computes eight DSMs in a SIMD-parallel fashion; the results can be found in Table 6. These results show that the throughput almost doubles when migrating from AVX2 to AVX-512. This is an important result since it confirms that optimizing for throughput allows one to overcome the parallelization bottlenecks of latency-oriented implementations (i.e., half-empty vectors and sequential dependencies) discussed in Section. 1. However, we note that the throughput gain is a bit below the (theoretical) factor of two in practice when shifting to AVX-512. We believe that one of the reasons is a higher Cycles-Per-Instruction (CPI) count for some instructions in AVX-512 than in AVX2. For example, for the addition (`vpaddq`) and `XOR` (`vpxor`), the CPI count is 0.33 in the AVX2 instruction set, while it is 0.5 in AVX-512.

**Table 6.** Latency and throughput of our separate-approach-based DSM implementation using AVX-512 on an Intel Xeon W-2245 Cascade Lake CPU, where the multiprocessing is applied.

| #Thread | Latency | #Instances | Throughput | Ratio |
|---|---|---|---|---|
| 1 | 415,177 cycles | 8 | 75,385 ops/sec | 1.00× |
| 2 | 420,727 cycles | 16 | 148,867 ops/sec | 1.97× |
| 4 | 424,115 cycles | 32 | 295,184 ops/sec | 3.92× |
| 8 | 423,650 cycles | 64 | 591,326 ops/sec | 7.84× |
| 16 | 807,653 cycles | 128 | 623,011 ops/sec | 8.26× |

## 5 Conclusion

In recent years, EdDSA has been adopted in a multitude of applications, ranging from security protocols over blockchains to electronic voting systems. In many application scenarios, users care about how many operations, especially signature verifications, can be performed within a certain time, instead of how quickly a single operation can be processed. However, virtually all software optimizations for EdDSA described in the literature focused on reducing the latency of one execution of the verification operation. In this paper, we proposed to use the vector engines of modern Intel processors to optimize throughput, where each signature can be verified independently but concurrently. We explored some state-of-the-art options for implementing DSM, namely the JSF-based approach, the NAF-based approach, and the separate approach. Our experiments confirm that for throughput-optimized software, the separate approach is the most efficient implementation option for batch DSM and improves throughput by 52% compared to the other two approaches. We found that for the two combined approaches, which are widely used in most of the latency-optimized implementations, lose the advantages of avoiding point operations at the evaluation stage for batch implementation, and this impact is amplified when a more powerful vector engine, e.g., AVX-512 is used. Our most efficient implementation (i.e., the implementation based on the separate approach) achieves a 33% higher throughput compared to the best latency-optimized implementation. In addition, we can further enhance our software throughput by utilizing multiprocessing and found that the optimal thread number for parallel implementation matches the number of physical CPU cores. The Xeon W-2245 CPU we used for benchmarking has eight cores and is capable of executing 623,011 DSMs per second. However, there exist Cascade Lake CPUs with up to 28 cores, which means that a single CPU could reach a throughput of roughly two million EdDSA verifications per second.

We envision that this paper will serve as inspiration for future research in throughput-optimized cryptography, motivated by our observation that the best implementation option for latency-optimized software is not necessarily ideal when high throughput is the main goal. As part of our future work, we plan to develop an EdDSA verification implementation using the latest AVX-512IFM instructions, which will further improve the throughput per core.

**Acknowledgements**

This work has been supported by the Luxembourg National Research Fund (FNR) via the CORE project ImPAKT (C21/IS/16221219/ImPAKT).

## References

1. AppleSecurity. imessage with pq3: The new state of the art in quantum-secure messaging at scale, 2024. `https://security.apple.com/blog/imessage-pq3/`.
2. R. Bellman and E. Straus. Addition chains of vectors. *The American Mathematical Monthly*, 71(7):806–808, 1964.
3. D. J. Bernstein. Pippenger's exponentiation algorithm. Preprint, available online at `https://cr.yp.to/papers.html#pippenger`, 2002.
4. D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography — PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Verlag, 2006.
5. D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted edwards curves. In *Progress in Cryptology–AFRICACRYPT 2008: First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings 1*, Lecture Notes in Computer Science, pages 389–405. Springer, 2008. `https://doi.org/10.1007/978-3-540-68164-9_26`.
6. D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012. `https://doi.org/10.1007/s13389-012-0027-1`.
7. D. J. Bernstein and K. Nath. Artifact: Lib25519, a microlibrary for the x25519 encryption system and the ed25519 signature system, 2024. `https://lib25519.cr.yp.to`.
8. E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 200–207. Springer, 1992.
9. H. Cheng, G. Fotiadis, J. Großschädl, and P. Y. A. Ryan. Highly vectorized SIKE for AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(2):41–68, 2022. `https://doi.org/10.46586/tches.v2022.i2.41-68`.
10. H. Cheng, G. Fotiadis, J. Großschädl, P. Y. A. Ryan, and P. B. Rønne. Batching CSIDH group actions using AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):618–649, 2021. `https://doi.org/10.46586/tches.v2021.i4.618-649`.
11. H. Cheng, J. Großschädl, J. Tian, P. B. Rønne, and P. Y. A. Ryan. High-throughput elliptic curve cryptography using AVX2 vector instructions. In *Selected Areas in Cryptography (SAC)*, LNCS 12804, pages 698–719. Springer-Verlag, 2020. `https://doi.org/10.1007/978-3-030-81652-0_27`.
12. T. Chou. Sandy2x: New Curve25519 speed records. In *Selected Areas in Cryptography (SAC)*, LNCS 9566, pages 145–160. Springer-Verlag, 2015. `https://doi.org/10.1007/978-3-319-31301-6_8`.

13. A. Faz-Hernández, J. López, and R. Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software (TOMS)*, 45(3):1–35, 2019. `https://doi.org/10.1145/3309759`.
14. A. Faz-Hernández and J. C. López-Hernández. Fast implementation of Curve25519 using AVX2. In *Progress in Cryptology (LATINCRYPT)*, LNCS 9230, pages 329–345. Springer-Verlag, 2015. `https://doi.org/10.1007/978-3-319-22174-8_18`.
15. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology (CRYPTO)*, LNCS 2139, pages 190–200. Springer-Verlag, 2001. `https://doi.org/10.1007/3-540-44647-8_11`.
16. J. Großschädl, C. Franck, and Z. Liu. Lightweight EdDSA signature verification for the ultra-low-power internet of things. In *Information Security Practice and Experience (ISPEC)*, LNCS 13010, pages 263–282. Springer-Verlag, 2021. `https://doi.org/10.1007/978-3-030-93206-0_16`.
17. D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004. `https://doi.org/10.1007/b97644`.
18. H. Hişil, B. Eğrice, and M. Yassi. Fast 4 way vectorized ladder for the complete set of Montgomery curves. Cryptology ePrint Archive, Report 2020/388, 2020. Available for download at `https://eprint.iacr.org`.
19. H. Hisil, B. Egrice, and M. Yassi. Fast 4 way vectorized ladder for the complete set of Montgomery curves. Cryptology ePrint Archive, Paper 2020/388, 2020. `https://eprint.iacr.org/2020/388`.
20. H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted edwards curves revisited. Cryptology ePrint Archive, Paper 2008/522, 2008. `https://eprint.iacr.org/2008/522`.
21. E. Kret and R. Schmidt. The pqxdh key agreement protocol, 2024. `https://signal.org/docs/specifications/pqxdh/`.
22. B. Möller. *Public-Key CryptographyTheory and Practice*. Ph.D. thesis, Technische Universität Darmstadt, 2003.
23. P. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation (MCOM)*, 48(177):243–264, 1987. `https://doi.org/10.1090/S0025-5718-1987-0866113-7`.
24. K. Nath and P. Sarkar. Efficient 4-way vectorizations of the Montgomery ladder. *IEEE Transactions on Computers (TOC)*, 71(3):712–723, 2021. `https://doi.org/10.1109/TC.2021.3060505`.
25. K. Okeya and K. Sakurai. Efficient elliptic curve cryptosystems from a scalar multiplication algorithm with recovery of the y-coordinate on a montgomery-form elliptic curve. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2162, pages 126–141. Springer-Verlag, 2001. `https://doi.org/10.1007/3-540-44709-1_12`.
26. N. Pippenger. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 258–263. IEEE Computer Society, 1976.
27. C.-P. Schnorr. Efficient identification and signatures for smart cards. In G. Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer Verlag, 1990.
28. J. A. Solinas. Efficient arithmetic on koblitz curves. *Designs, Codes and Cryptography (DCC)*, 19(2/3):195–249, 2000. `https://doi.org/10.1023/A:1008306223194`.
29. J. A. Solinas. Low-weight binary representation for pairs of integers. *Combinatorics and Optimization Research Report CORR 2001-41*, 2001.

30. WikiChip. Intel microarchitectures: Cascade lake, Accessed: 2024-10-06. `https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake`.