Multi-way High-throughput Implementation of Kyber

Xuan Yu[1][0009−0009−1922−7719], Jipeng Zhang[1], Junhao Huang[2], Donglong Chen[2], and Lu Zhou[1,✉]

[1] Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China
yxcos2021@163.com, jp-zhang@outlook.com, lu.zhou@nuaa.edu.cn
[2] Guangdong Provincial Key Laboratory of IRADS, BNU-HKBU United International College, Zhuhai, China {huangjunhao,donglongchen}@uic.edu.cn

Abstract. This paper presents a novel approach to implement a multi-way Key Encapsulation Mechanism (KEM) that takes full advantage of the parallelism provided by SIMD instructions. Specifically, our multi-way `KeyGen()` function is capable of generating multiple unique key pairs simultaneously. To start, we introduce a multi-way data format to support the proposed multi-way KEM implementation. We then introduce a multi-way NTT implementation based on this novel data format. Compared to traditional one-way NTT implementation, our multi-way NTT significantly reduces the complicated permutation operations, leading to overall performance improvements. In terms of SHA3-related computations, while previous one-way Kyber implementations have used multi-way SHAKE to speed up the matrix and vector generation, the inherent execution flow of the one-way KEM cannot fully utilize the parallelism of the multi-way SHA3 implementation. On the contrary, our multi-way implementation effectively parallelizes these SHA3 computations, resulting in substantial speed enhancements. We have applied this methodology to Kyber on AVX2 and AVX-512, developing a 16-way Kyber implementation for AVX2 and a 32-way implementation for AVX-512. With faster multi-way NTT and fully parallelized SHA3 computations, the key generation, encapsulation, and decapsulation in Kyber on AVX2 and AVX-512 achieve impressive speed-ups of 36.0%/54.6%/25.9% and 80.6%/130.3%/51.3%, respectively, compared to traditional one-way AVX2 implementation. Lastly, we demonstrate the versatility of our multi-way approach in real-world applications. For example, the multi-way `KeyGen()` function can be seamlessly integrated into the TLS protocol using OpenSSL ENGINE APIs, extending its advantages to a wide range of TLS applications. Additionally, the multi-recipient KEM (mKEM) protocols used for secure group messaging can also benefit from our multi-way approach to enhance their performance.

Keywords: Post-quantum cryptography · Kyber · SIMD · Optimized implementation · Multi-way implementation.

1 Introduction

In 1999, Shor [28] proposed efficient polynomial-time algorithms on a hypothetical quantum computer which can break the conventional public-key cryptosystems once large-scale quantum computers become practical. To counter potential quantum attacks in the quantum era, the National Institute of Standards and Technology (NIST) [2] initiated the post-quantum cryptography (PQC) standardization project to solicit, evaluate, and standardize the PQC algorithms that are secure against potential quantum attacks. NIST has completed the third round process and selected four PQC candidates for final standardization. Till the end of 2023, NIST has published three PQC standardization drafts, namely FIPS203 [24], FIPS204 [23], and FIPS205 [25].

Among the three drafts, Kyber [5] is the only KEM scheme that has been standardized and officially named ML-KEM in NIST FIPS203 due to its high security and excellent efficiency. Kyber is an IND-CCA2-secure KEM scheme, in which the IND-CCA2-secure KEM is constructed over the IND-CPA-secure public-key encryption using the tweaked Fujisaki-Okamoto (FO) transform [13]. The security of Kyber is based on a strong lattice-based hard problem, specifically the module learning-with-errors (MLWE) problem [21]. As Kyber has been standardized by NIST and is the only KEM scheme in the final selection, the efficient implementation of Kyber on various platforms becomes an important research focus with significant implications for ensuring data security and privacy for different application scenarios in the quantum era.

Nowadays, with the rapid development of the Internet and big data, the existing servers face the significant task of handling a substantial volume of data processing. For instance, Big name sales, like Black Friday, can drive traffic levels 30 times higher than a normal day [22]. These transactions normally employ the Transport Layer Security (TLS) protocol, involving extensive public-key cryptographic computations. In the face of the quantum era, optimizing Kyber, the only KEM scheme standardized by NIST, to enable high throughput implementation on servers emerges as a crucial research focus. Such optimization not only alleviates computational pressure and reduces server response time but also provides a strong guarantee of data security in the quantum era.

1.1 Related Work

There is much significant work done by other researchers in the field of implementing cryptographic schemes on various platforms.

Seiler [27] proposed a signed Montgomery reduction that enables a fast approach with integer instructions, based on which he presented the state-of-the-art implementation of Kyber on AVX2. Cheng et al. [11] presented an optimized implementation of SIKE using Intel AVX-512IFMA instructions, parallelizing and speeding up the base/extension field arithmetic, point arithmetic, and isogeny computations performed by SIKE. The instantiation with the `SIKEp503` parameter set is approximately 1.5 times faster than the to-date best AVX-512IFMA-base SIKE software. Becker and Kannwischer [7] introduced two new techniques

for the fast implementation of the Keccak permutation on the A-profile of the Arm architecture, based on which they presented the implementation of SPHINCS$^+$, achieving up to 1.89× performance improvements compared to the state of the art. Sujoy Sinha Roy [26] proposed a high-throughput software implementation of Saber, called "SaberX4", which applies the batching technique and processes four Saber KEM operations in parallel using the AVX2 instructions. The implementation of SaberX4 achieves nearly 1.5 times higher throughput compared to the AVX2-optimized non-batched implementation of Saber. Zheng et al. [33] introduced an improved parallel small polynomial multiplication and a tailored reduction method to improve the efficiency. They also presented an optimized implementation of Dilithium by utilizing AVX2 and AVX-512 and achieved 36.6%-47.4% speed-ups for key generation, signing, and verification for Dilithium2/3/5, respectively. Huang et al. [18,19] introduced two improved Plantard's modular arithmetic (Plantard arithmetic), based on which they presented faster implementations of Kyber on Cortex-M3, Cortex-M4, and RISC-V platforms, achieving the speedup ranging from 11.28% to 56.95% for NTT and INTT respectively. Greconici [15] presented a speed optimization for Kyber on the open-source RISC-V architecture, including optimized implementations of NTT/INTT, Montgomery reduction, and Barrett reduction.

1.2 Motivations

Currently, researchers are primarily focused on speeding up one-way Kyber on different platforms with various instruction set architectures (ISA), as listed in Section 1.1. However, to the best of our knowledge, there is limited exploration specifically on multi-way implementations using the Single-Instruction-Multiple-Data (SIMD) instructions for Kyber. The main reasons for focusing on the implementation of multi-way Kyber are as follows.

First of all, from the application perspective, we identified some real-world application scenarios that favor the multi-way approach. This is inspired from the mKEM and amKEM primitives presented in [4], where the CPA-secure public-key encryption (PKE) primitive needs to be invoked N times to generate N distinct encapsulation keys in the secure group messaging and broadcast application scenarios. This observation motivated us to explore the multi-way implementation of PKE and KEM. As a result, when N is sufficiently large, our proposed multi-way implementation can be leveraged to efficiently generate batched encapsulation keys, thereby enhancing the performance of these primitives.

Secondly, upon analyzing the subroutines of Kyber, we find that multi-way implementation of Kyber is not only a seldom-explored subject but also an effective optimization method. Intel's Advanced Vector Extensions instruction sets, including AVX2, and AVX-512, are designed to provide high performance with expanded registers and advanced SIMD instructions. The existing Kyber implementations using SIMD mainly focused on accelerating one-way Kyber, where all registers and parallelism are utilized for one-way operations. The dataflow is rearranged to facilitate one-way acceleration, which requires time-consuming permutation instructions, leading to unavoidable performance penalties. In comparison

to the one-way implementations adopted by most work (e.g., [1,10,7,33,31]), the multi-way implementation has the potential to reduce redundant operations and mitigate the permutation costs associated with rearranging dataflow required by one-way implementations, thereby significantly improving the throughput of the cryptographic algorithm.

Finally, as demonstrated in [7], the multi-way vectorized SHA3 implementation leveraging SIMD exhibits more efficiency compared to its one-way scalar implementation. However, the inherent execution flow of the one-way KEM cannot fully utilize the parallelism of the multi-way SHA3 implementation. In contrast, the multi-way Kyber implementation would facilitate a more seamless and extensive utilization of multi-way SHA3, thereby yielding a significant enhancement in efficiency and throughput for Kyber implementation. More details are described in Section 3.3.

1.3 Contributions

Based on the above observations, we explore a multi-way Kyber implementation on both AVX2 and AVX-512, respectively. Our contributions are as follows:

- We develop a multi-way KEM implementation that fully leverages the parallelism of SIMD instructions. We introduce a multi-way data format to implement the multi-way Kyber, including `16-way format` and `32-way format` tailored for AVX2 and AVX-512, respectively. Based on the multi-way data format, we instantiate a 16-way implementation on AVX2 and a 32-way implementation on AVX-512 for Kyber, where each lane in the multi-way implementation performs a one-way Kyber instance.
- We design and implement a 16-way and a 32-way NTT-based polynomial multiplication for Kyber on AVX2 and AVX-512, respectively. According to the characteristics of the AVX2 instruction set, we adopt a 3+3+1 layer-merging strategy to implement the 16-way NTT similar to the one-way NTT implementation in [3]. Different from the 16-way implementation, we adopt a 4+3 layer-merging strategy for NTT implementation on AVX-512. Owing to the characteristic of our multi-way implementation methodology, both multi-way NTT implementations reduce the permutation operations that are indispensable in one-way implementation, thereby improving the overall efficiency.
- We further optimize the utilization of SHA3-related operations in our multi-way implementation, parallelizing all SHA3-related operations within the KEM primitive using 4-way and 8-way SHA3, thereby substantially accelerating the SHA3-related computations and significantly enhancing the throughput of Kyber.
- Our investigation reveals that the proposed multi-way KEM implementation exhibits broad applicability. Previous works [8,32] have successfully implemented batched key generation for NTRU Prime and X25519, respectively, and integrated them into TLS using the OpenSSL ENGINE APIs, thereby validating the efficacy of the multi-way batched implementation. Moreover,

our multi-way method for KEM is also well-suited for various cryptographic protocols and primitives, including KEMTLS, mKEM, amKEM, and others.

Code. The software of this work is available at https://github.com/ccccosine/Kyber_AVX2_16w.git and https://github.com/ccccosine/Kyber_AVX512_32w.git.

2 Preliminaries

In this section, we briefly revisit the specification of Kyber and NTT-based polynomial multiplication. Then, the AVX2 and AVX-512 instruction sets will also be introduced.

2.1 Kyber

The security of Kyber [9] is based on the hardness of solving the module learning-with-errors (MLWE) problem. The IND-CCA2-secure Kyber KEM scheme (Kyber.CCAKEM) is constructed over an IND-CPA-secure PKE called CPAPKE using a variant of the Fujisaki-Okamoto transform [13]. The Kyber.CPAPKE protocol includes key generation, encryption, and decryption, which are described in Algorithms 1, 2, and 3, respectively.

Algorithm 1 Kyber.PKE.KeyGen [9]

Output: $sk \in \mathcal{B}^{12\cdot k\cdot n/8}$;$pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$

1: $seed \leftarrow \mathcal{B}^{32}$ ▷ Generate a 32-byte random seed
2: $(\rho, \sigma) := \mathsf{SHA3{-}512}(seed)$
3: $\hat{\mathbf{A}} := \mathsf{GenMatrixA}(\rho)$
4: $(\mathbf{s}, \mathbf{e}) := \mathsf{SampleVec}(\sigma)$
5: $\hat{\mathbf{t}} := \mathsf{Compress}_{12}(\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s})) + \mathrm{NTT}(\mathbf{e})$
6: $\hat{\mathbf{s}}' := \mathsf{Compress}_{12}(\hat{\mathbf{s}})$
7: **return** $(pk := (\hat{\mathbf{t}}, \rho), sk := \hat{\mathbf{s}}')$

Algorithm 2 Kyber.PKE.Enc [9]

Input: $pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$; $m, r \in \mathcal{B}^{32}$
Output: $c \in \mathcal{B}^{d_u\cdot k\cdot n/8+d_v\cdot n/8}$

1: $\hat{\mathbf{t}}' := \mathsf{Decompress}_{12}(\hat{\mathbf{t}})$
2: $\hat{\mathbf{A}} := \mathsf{GenMatrixA}(\rho)$
3: $(\mathbf{r}, \mathbf{e_1}, \mathbf{e_2}) := \mathsf{SampleVec}(r)$
4: $\hat{\mathbf{r}} := \mathsf{NTT}(\mathbf{r})$
5: $\mathbf{u} := \mathsf{INTT}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
6: $\mathbf{v} := \mathsf{INTT}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + \mathbf{e_2} + \mathrm{Compress}_1(m)$
7: **return** $c := (\mathbf{u}' := \mathrm{Compress}_{10}(\mathbf{u}), \mathbf{v}' := \mathrm{Compress}_4(\mathbf{v}))$

Algorithm 3 Kyber.PKE.Dec [9]

Input: $sk \in \mathcal{B}^{12\cdot k\cdot n/8}$; $c \in \mathcal{B}^{(d_u\cdot k+d_v)\cdot n/8}$
Output: $m \in \mathcal{B}^{32}$

1: $\mathbf{u} := \mathsf{Decompress}_{10}(\mathbf{u}')$
2: $\mathbf{v} := \mathsf{Decompress}_4(\mathbf{v}')$
3: $\hat{\mathbf{s}} := \mathsf{Decompress}_{12}(\hat{\mathbf{s}}')$
4: **return** $\mathsf{Compress}_1(\mathbf{v} - \mathsf{INTT}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})))$

Algorithm 4 Kyber.KEM.KeyGen [9]

Output: $sk \in \mathcal{B}^{24\cdot k\cdot n/8+96}$; $pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$

1: $z \leftarrow \{0,1\}^{256}$
2: $(pk, sk') := \text{Kyber.PKE.KeyGen}()$
3: $sk := (sk'||pk||\mathrm{SHA3{-}256}(pk)||z)$
4: **return** (pk, sk)

Kyber has three parameter sets: Kyber512, Kyber768, and Kyber1024. In this work, we primarily focus on Kyber768, and the relevant parameters used are as follows: polynomial dimension $n = 256$; matrix and vector dimension $k = 3$;

Algorithm 5 Kyber.KEM.Encaps [9]

Input: $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Output: $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Output: $K \in \mathcal{B}^*$
1: $m \leftarrow \{0,1\}^{256}$
2: $m' := \text{SHA3}{-}256(m)$
3: $(\bar{K}, r) := \text{SHA3}{-}512(m'||\text{SHA3}{-}256(pk))$
4: $c :=$Kyber.PKE.Enc(pk, m, r)
5: $K := \text{SHAKE256}(\bar{K}||\text{SHA3}{-}256(c))$
6: **return** (c, K)

modulus $q = 3329$; and $d_u = 10$, $d_v = 4$ where d_u and d_v are the compression parameter used for compressing polynomial vectors and polynomials respectively. The $\mathcal{B}$ in the algorithm description means that the variable is in byte format and the variable with $\hat{}$ represents the variable in the NTT domain. The $\circ$ symbol denotes the coefficient-wise multiplication, and the ^{T} symbol represents the transposition of a vector or matrix.

Kyber computes on the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ where dimension $n = 256$ and modulus $q = 3329$. Because $q < 2^{12}$, polynomial coefficients can be accommodated in a 16-bit integer. This selected polynomial ring enables a 7-layer incomplete NTT with a primitive 256-th root of unity.

Algorithm 6 Kyber.KEM.Decaps [9]

Input: $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Input: $sk \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$
Output: $K \in \mathcal{B}^*$
1: $pk := sk + 12 \cdot k \cdot n/8$
2: $h := sk + 24 \cdot k \cdot n/8 + 32 \in \{0,1\}^{256}$
3: $z := sk + 24 \cdot k \cdot n/8 + 64$
4: $m' :=$Kyber.PKE.Dec(sk, c)
5: $(\bar{K}', r') := \text{SHA3}{-}512(m'||h)$
6: $c' :=$Kyber.PKE.Enc(pk, m', r')
7: **if** $c = c'$ **then**
8: **return** $K := \text{SHAKE256}(\bar{K}'||\text{SHA3}{-}256(c))$
9: **else**
10: **return** $K := \text{SHAKE256}(z||\text{SHA3}{-}256(c))$
11: **end if**
12: **return** K

Algorithms 4, 5, and 6 denote the key generation, encapsulation, and decapsulation of the Kyber.CCAKEM, respectively. We can see that Kyber.CCAKEM protocols are constructed based on Kyber.CPAPKE protocol and some symmetric primitives. The `Kyber.CCA.KeyGen` utilizes the `Kyber.CPA.KeyGen` to generate key pairs. The `Kyber.CCA.Encaps` encrypts the random message using `Kyber.CPA.Enc` to obtain the ciphertext and employ SHA3-related primitives to

generate a shared key. The communication peer receives the ciphertext and utilizes the `Kyber.CPA.Dec` to recover the message generated by `Kyber.CCA.Encaps`, and then compute the shared key with SHA3-related primitives.

2.2 Number Theoretic Transform (NTT)

Number Theoretic Transform is an efficient way to compute polynomial multiplication on $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with the time complexity of $O(n\log n)$ [12]. Since $\mathbb{Z}_q$ in Kyber only contains 256-th primitive roots of unity but not 512-th primitive roots of unity [5], the polynomial $f(X) = X^{256} + 1$ can only be factorized into 128 degree-1 polynomials modulo q as follows:

$$X^{256} + 1 = \prod_{i=0}^{127}(X^2 - \zeta^{2i+1}) = \prod_{i=0}^{127}(X^2 - \zeta^{\mathbf{2br}_7(i)+1}),$$

where $\mathbf{br}_7(i)$ for $i = 0,\cdots,127$ denotes the bit reversal of the unsigned 7-bit integer i and $\zeta = 17$ is the first 256-th primitive root of unity modulo q. After NTT, the polynomial $f(X)$ is given by

$$(f \bmod X^2 - \zeta^{\mathbf{2br}_7(0)+1}, \cdots, f \bmod X^2 - \zeta^{\mathbf{2br}_7(127)+1}).$$

The computation of NTT can also be represented as follows:

$$\mathrm{NTT}(f) = \hat{f} = \hat{f}_0 + \hat{f}_1 X + \cdots + \hat{f}_{255} X^{255}$$

with

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j}\zeta^{(\mathbf{2br}_7(i)+1)j},$$

$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1}\zeta^{(\mathbf{2br}_7(i)+1)j}.$$

Besides, in the light of the Chinese remainder theorem, the natural ring homomorphism is in fact an isomorphism. Hence, we can compute the polynomial multiplication f and g as $f \cdot g = \mathrm{INTT}(\mathrm{NTT}(f) \circ \mathrm{NTT}(g))$, where $\circ$ represents the coefficient-wise multiplication of two degree-1 polynomials in the NTT domain.

2.3 AVX2 and AVX-512

AVX2 and AVX-512 are two advanced SIMD instruction sets developed by Intel to enhance the parallel computing capabilities and vectorization supports of processors.

AVX2 is an extension of the AVX instruction set, introducing additional instructions and features to strengthen the processor's support for vectorized operations. AVX2 is equipped with sixteen 256-bit YMM registers (`ymm0-ymm15`) to perform the SIMD instructions. It also introduces additional vectorized load

and store instructions, as well as bitwise instructions. These enhancements enable improved performance and increased efficiency for a wide range of applications, including multi-media processing, scientific simulations, and cryptography.

AVX-512 is the latest extension of the AVX instruction set, introducing 32 usable 512-bit ZMM registers (`zmm0-zmm31`) to further expand the vector width and parallel computing capabilities. AVX-512 offers a broader set of floating-point and integer operation instructions, along with new memory operation instructions, making it suitable for high-performance computing and handling large-scale data. AVX-512 supports greater data parallelism, enabling the processing of more data elements in one instruction.

SIMD instructions support operations on data at 64-bit, 32-bit, 16-bit, and 8-bit granularity within registers. Simply put, take the instruction `VPADDW` on AVX2 as an example. There are 16 16-bit integers $a_0, \cdots, a_{15}$ stored in a 256-bit register. Similarly, the other 16 16-bit integers $b_0, \cdots, b_{15}$ are stored in the other 256-bit register. `VPADDW` is able to compute 16 16-bit additions in parallel, i.e., $a_0 + b_0, \cdots, a_{15} + b_{15}$, and the results are stored in the destination register. The instruction `VPADDW` is also available on the AVX-512 platform. However, unlike AVX2, AVX-512 has 32 16-bit data elements stored in a register when using `VPADDW`. Consequently, the destination register also contains 32 16-bit results.

3 The multi-way Kyber Implementation

The term "multi-way implementation" indicates the capability to execute multiple independent operations in parallel, where each lane in the multi-way implementation performs a one-way Kyber instance. For example, in the case of 16-way NTT-based multiplication, it allows for the parallel execution of 16 independent polynomial multiplications. Similarly, 16-way Kyber.CCAKEM.KeyGen enables the generation of 16 independent key pairs at once. Our multi-way Kyber implementation is built upon the official AVX2 implementation of Kyber [20] provided by Kyber's team, and can be seamlessly adapted to the ML-KEM primitive proposed in the FIPS203, which builds upon Kyber with modifications.

Before we move forward to the implementation details, we need to specify the polynomial representation formats in one-way and multi-way implementations. In one-way AVX2 Kyber implementation, each polynomial consists of 256 coefficients $a_0, a_1, \cdots, a_{254}, a_{255}$, which are stored in sequence. We define this format as `one-way format`.

To facilitate the multi-way Kyber implementation using SIMD instructions, we design a new format, namely the `multi-way format`. Notably, this format is not the straightforward concatenation of multiple polynomials in the `one-way format`. To illustrate, consider the `16-way format` as an instance. As depicted in Figure 1, every letter denotes an independent polynomial, with the subscript of the letter indicating the indices of polynomial coefficients. In the `16-way format`, we schedule 16 i-th coefficients ($i \in [0, 256)$) from 16 distinct polynomials in sequence so that we can load these 16 i-th coefficients into one 256-bit AVX2 register. The `32-way format` follows a similar design tailored for

AVX-512. In the rest of the paper, the 16-way Kyber implementation leverages the `16-way format` while the 32-way Kyber implementation employs the `32-way format` to represent polynomials.

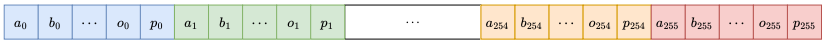


Fig. 1. The 16-way format to represent 16 polynomials

3.1 NTT/INTT implementation

NTT/INTT are one of the most time-consuming subroutines in Kyber. The one-way implementation of Kyber using AVX2 [20] adopts the 1+6 layer-merging strategy to implement NTT. However, in the 6-layer implementation of NTT, the coefficient order produced in each NTT layer can not be directly used in the next NTT layer. Therefore, additional permutation instructions are required to rearrange coefficients' order after each NTT layer, leading to a significant performance overhead.

In our 16-way implementation, the 16-way NTT/INTT can perform 16 independent polynomial multiplications in parallel. To achieve this, we follow the `16-way format` and load 16 16-bit coefficients in each 256-bit AVX2 register, with each coefficient originating from a different polynomial. Since there are only sixteen 256-bit registers available, it is insufficient to accommodate 4-layer merging NTT implementation, as this would exhaust all registers for accommodating coefficients, precluding other essential computations. Therefore, we utilize the 3+3+1 layer-merging strategy to implement the 16-way NTT/INTT. In this way, we need 8 256-bit AVX2 registers to load 8×16 coefficients from 16 different polynomials. As for the remaining 8 registers, we reserve one register for storing 16 packed q constants, and the rest for temporary usage during the computation of butterfly units. AVX-512 offers up to 32 512-bit registers, allowing us to load the `32-way format`'s coefficients into 16 512-bit registers for the 32-way implementation. Consequently, the 4+3 layer-merging strategy can be employed on AVX-512. The remaining 16 512-bit registers are adequate for storing the packed constants q and other intermediate values. Moreover, compared to the one-way implementation that requires additional permutation instructions, the order of output in each NTT layer of our multi-way implementation of Kyber meets the requirement of dataflow order, thereby eliminating all the permutation operations.

The core operations of NTT and INTT are the butterfly units, which are commonly implemented with the Cooley-Tukey (CT) butterfly [12] and the Gentleman-Sande (GS) butterfly [14], respectively. The CT butterfly inputs data in normal order and outputs data in bit-reversal order. In contrast, the GS butterfly takes inputs in bit-reversal order and produces outputs in normal order.

During the NTT operations, the powers of ζ, referred to as twiddle factors, are often stored in a precomputed format for saving expenses when computing Montgomery reduction. The precomputed twiddle factor is in the form of $\zeta^k \times 2^{16} \bmod q$, where the ζ^k is a power of ζ. We rearrange the order of the twiddle factors to align with the order they are used in our implementation.

In the one-way INTT implementation on AXV2, the coefficients within a register have different ranges, causing all coefficients to be reduced simultaneously when one requires reduction. In contrast, our multi-way implementation enables the coefficients in a register to share a consistent range, leading to more efficient coefficient reduction. This ultimately saves a total of 20 and 62 modular reductions for the 16-way and 32-way implementation, respectively.

3.2 The multi-way Kyber.CPAPKE implementation

Apart from the multi-way NTT/INTT, there are other subroutines in the Kyber.CPAPKE that are required to be implemented in multi-way. In this section, we will introduce the subroutines in the Kyber.CPAPKE that need further adjustment during the multi-way Kyber.CPAPKE implementation.

Algorithm 7 Kyber.PKE.KeyGen$-N$way [9]

Output: $sk \in \mathcal{B}^{(12\cdot k\cdot n/8)\times N}$; $pk \in \mathcal{B}^{(12\cdot k\cdot n/8+32\cdot 2)\times N}$

1: $seed_0 \sim seed_{N-1} \leftarrow \mathcal{B}^{32\times N}$ ▷ $N := 16$ or $N := 32$
2: **for** $i = 0$ to 3 **do**
3: $(\rho_{N/4\times i}, \sigma_{N/4\times i}) \sim (\rho_{N/4\times(i+1)-1}, \sigma_{N/4\times(i+1)-1}) := \mathsf{SHA3{-}512{-}}(N/4)\mathsf{way}$ $(seed_{(N/4\times i)\sim(N/4\times(i+1)-1)})$ ▷ 4-way/8-way SHA3 on AVX2/AVX-512
4: $\hat{\mathbf{A}} := \mathsf{GenMatrixA}(\rho_{0\sim N-1})$
5: $\hat{\mathbf{A}}\mathbf{seq} := \mathsf{Matrix\_seqto}N(\hat{\mathbf{A}})$ ▷ $\hat{\mathbf{A}}\mathbf{seq}$ represents $\hat{\mathbf{A}}$ is in $\mathsf{multi-way\ format}$
6: $(\mathbf{s_{0\sim N-1}}, \mathbf{e_{0\sim N-1}}) := \mathsf{SampleVec}(\sigma_{0\sim N-1})$
7: $(\mathbf{sseq_{0\sim N-1}}, \mathbf{eseq_{0\sim N-1}}) := \mathsf{Polyvec\_seqto}N(\mathbf{s_{0\sim N-1}}, \mathbf{e_{0\sim N-1}})$
8: $\hat{\mathbf{t}}\mathbf{seq_0}, \cdots, \hat{\mathbf{t}}\mathbf{seq}_{N-1} := \mathsf{Compress}_{12}(\hat{\mathbf{A}}\mathbf{seq} \circ \mathrm{NTT}(\mathbf{sseq_{0\sim N-1}}) + \mathrm{NTT}(\mathbf{eseq_{0\sim N-1}}))$
9: $\hat{\mathbf{s}}\mathbf{seq}'_0, \cdots, \hat{\mathbf{s}}\mathbf{seq}'_{N-1} := \mathsf{Compress}_{12}(\hat{\mathbf{s}}\mathbf{seq_0}, \cdots, \hat{\mathbf{s}}\mathbf{seq}_{N-1})$
10: $(\hat{\mathbf{t}}_{\mathbf{0\sim N-1}}, \hat{\mathbf{s}}_{\mathbf{0\sim N-1}}) := \mathsf{Keypair\_seqfrom}N(\hat{\mathbf{t}}\mathbf{seq_{0\sim N-1}}, \hat{\mathbf{s}}\mathbf{seq_{0\sim N-1}})$
11: **return** $(pk_{0\sim N-1} := (\hat{\mathbf{t}}_{\mathbf{0\sim N-1}}, \rho_{0\sim N-1}), sk_{0\sim N-1} := \hat{\mathbf{s}}'_{\mathbf{0\sim N-1}})$

Format sequence conversion. Our multi-way implementation leverages our specialized format, which also produces output in the `multi-way format`. This `multi-way format` is incompatible with the final result. So it is necessary to convert data between `one-way format` and `multi-way format`. As shown in the Algorithms 7, 8, and 9, the subroutines with `_seqtoN` or `_seqfromN` suffix are all conversion subroutines, responsible for transforming data between `one-way format` and `N-way format`. Specifically, N equals 16 for the 16-way AVX2 implementation and 32 for the 32-way AVX-512 implementation, respectively. The `seq` suffix of variables denotes that they are in `multi-way format`.

Inspired by the `rearrangement` subroutines of one-way Kyber implementation, we implement different format conversion subroutines for different formats of variables, including matrix, keypair, message, and so on. Compared to the

Algorithm 8 Kyber.PKE.Enc$-N$way [9]

Input: $pk \in \mathcal{B}^{(12\cdot k\cdot n/8+32\cdot 2)\times N}$; $m \in \mathcal{B}^{32\times N}$; $r \in \mathcal{B}^{32\times N}$ ▷ $N := 16$ or $N := 32$
Output: $c \in \mathcal{B}^{(d_u\cdot k\cdot n/8+d_v\cdot n/8)\times N}$

1: $\hat{\mathbf{t}}\mathbf{seq}_{0\sim N-1} := \mathsf{Keypair\_seqto}N(\hat{\mathbf{t}}_{0\sim N-1})$ ▷ $pk_{0\sim N-1} := (\hat{\mathbf{t}}_{0\sim N-1}, \rho_{0\sim N-1})$
2: $\hat{\mathbf{t}}\mathbf{seq}'_{0\sim N-1} := \mathsf{Decompress}_{12}(\hat{\mathbf{t}}\mathbf{seq}_{0\sim N-1})$
3: $\mathbf{mseq}_{0\sim N-1} := \mathsf{Msg\_seqto}N(m_{0\sim N-1})$
4: $\hat{\mathbf{A}} := \mathsf{GenMatrixA}(\rho_{0\sim N-1})$
5: $\hat{\mathbf{A}}\mathbf{seq} := \mathsf{Matrix\_seqto}N(\hat{\mathbf{A}})$
6: $(\mathbf{r}_{0\sim N-1}, \mathbf{e1}_{0\sim N-1}, \mathbf{e2}_{0\sim N-1}) := \mathsf{SampleVec}(r_{0\sim N-1})$
7: $(\mathbf{rseq}_{0\sim N-1}, \mathbf{e1seq}_{0\sim N-1}) := \mathsf{Polyvec\_seqto}N(\mathbf{r}_{0\sim N-1}, \mathbf{e1}_{0\sim N-1})$, $\mathbf{e2seq}_{0\sim N-1} := \mathsf{Poly\_seqto}N(\mathbf{e2}_{0\sim N-1})$
8: $\hat{\mathbf{r}}\mathbf{seq}_{0\sim N-1} := \mathsf{NTT}(\mathbf{rseq}_{0\sim N-1})$
9: $\mathbf{useq}_{0\sim N-1} := \mathsf{INTT}(\hat{\mathbf{A}}\mathbf{seq}^T \circ \hat{\mathbf{r}}\mathbf{seq}_{0\sim N-1}) + \mathbf{e1seq}_{0\sim N-1}$
10: $\mathbf{vseq}_{0\sim N-1} := \mathsf{INTT}(\hat{\mathbf{t}}\mathbf{seq}^T_{0\sim N-1} \circ \hat{\mathbf{r}}\mathbf{seq}_{0\sim N-1}) + \mathbf{e2seq}_{0\sim N-1} + \mathrm{Compress}_1(\mathbf{mseq}_{0\sim N-1})$
11: $\mathbf{useq}'_{0\sim N-1} := \mathrm{Compress}_{10}(\mathbf{useq}_{0\sim N-1}), \mathbf{vseq}'_{0\sim N-1} := \mathrm{Compress}_4(\mathbf{vseq}_{0\sim N-1})$
12: **return** $c_{0\sim N-1} := (\mathbf{u}'_{0\sim N-1}, \mathbf{v}'_{0\sim N-1})$
$:= \mathrm{Cipher\_seqfrom}N(\mathbf{useq}'_{0\sim N-1}, \mathbf{vseq}'_{0\sim N-1})$

one-way AVX2 implementation of Kyber, these format conversion subroutines were added as an extra component in our multi-way implementation of Kyber. The detail of format conversion is described in appendix A.

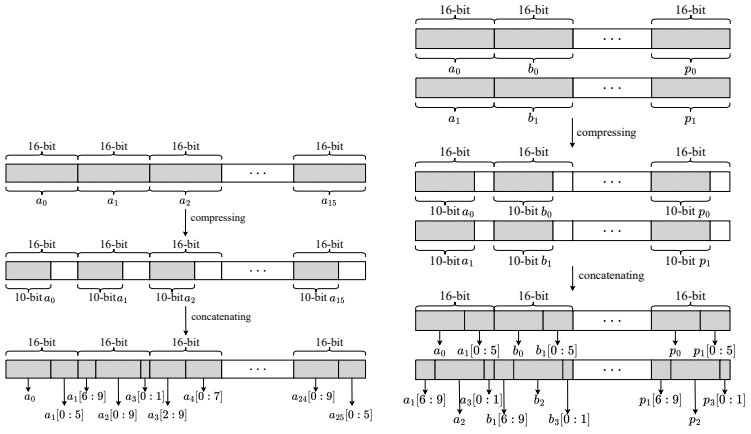
Reducing the cost of coefficient rearrangement. During the matrix generation (e.g., line 3 of Algorithm 1), the polynomials generated in this process are considered to be in the NTT domain and arranged in bit-reversal order. However, the output order of NTT in the one-way AVX2 Kyber implementation is not in bit-reversal order, indicating that there is a gap between the coefficient order of matrix $\mathbf{A}$ and the coefficient order of NTT's output. As a result, the one-way AVX2 Kyber implementation developed a coefficient `rearrangement` subroutine to bridge this gap, consuming 2064 instructions. For our multi-way implementation, the output order of our multi-way NTT is in bit-reversal order, eliminating the need for coefficient rearrangement, thereby improving efficiency.

Algorithm 9 Kyber.PKE.Dec$-N$way [9]

Input: $sk \in \mathcal{B}^{(12\cdot k\cdot n/8)\times N}$; $c \in \mathcal{B}^{(d_u\cdot k\cdot n/8+d_v\cdot n/8)\times N}$
Output: $m \in \mathcal{B}^{32\times N}$ ▷ $N := 16$ or $N := 32$

1: $(\mathbf{useq}'_{0\sim N-1}, \mathbf{vseq}'_{0\sim N-1}) := \mathrm{Cipher\_seqto}N(\mathbf{u}'_{0\sim N-1}, \mathbf{v}'_{0\sim N-1})$
2: $\hat{\mathbf{s}}\mathbf{seq}'_{0\sim N-1} := \mathrm{Keypair\_seqto}N(\hat{\mathbf{s}}'_{0\sim N-1})$ ▷ $sk_{0\sim N-1} := \hat{\mathbf{s}}'_{\mathbf{0}\sim N-\mathbf{1}}$
3: $\mathbf{useq}_{0\sim N-1} := \mathsf{Decompress}_{10}(\mathbf{useq}'_{0\sim N-1})$
4: $\mathbf{vseq}_{0\sim N-1} := \mathsf{Decompress}_4(\mathbf{vseq}'_{0\sim N-1})$
5: $\hat{\mathbf{s}}\mathbf{seq}_{0\sim N-1} := \mathsf{Decompress}_{12}(\hat{\mathbf{s}}\mathbf{seq}'_{0\sim N-1})$
6: $\mathbf{mseq}_{0\sim N-1} := \mathsf{Compress}_1(\mathbf{vseq}_{0\sim N-1} - (\mathsf{INTT}(\hat{\mathbf{s}}\mathbf{seq}^T_{0\sim N-1} \circ \mathsf{NTT}(\mathbf{useq}_{0\sim N-1}))))$
7: **return** $m_{0\sim N-1} := \mathrm{Msg\_seqfrom}N(\mathbf{mseq}_{0\sim N-1})$

In the one-way AVX2 implementation, the `Compress` subroutine in Algorithm 1 also utilizes the coefficient `rearrangement` subroutine. Recall that the polynomial coefficients in Kyber are 12-bit integers, as $q < 2^{12}$. For ease of programming, we store each coefficient in a 16-bit data type. The `Compress` subroutine in Algorithm 1 is used to compress every coefficient of the public and secret keys into 12-bit, which saves storage space and communication costs. As shown in lines 5 and 6 of Algorithm 1, the `Compress` subroutine takes inputs $\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s}) + \mathrm{NTT}(\mathbf{e})$ and $\hat{\mathbf{s}}$, respectively, which are not in bit-reversal order. Therefore, after compressing the polynomial coefficients, they need to be converted to bit-reversal order using the `rearrangement` subroutine to maintain compatibility with the reference implementation of Kyber. In contrast, the input of the `Compress` subroutine in our multi-way implementation is already in bit-reversal order, so there is no need to rearrange the coefficients, which further enhances the throughput of our multi-way implementation.



(a) Example in one-way AVX2 implementation

(b) Example in 16-way AVX2 implementation

Fig. 2. Compression subroutines of ciphertext ($a_1[0:5]$ denotes the least significant 6 bits of a_1's 10-bit value)

Optimization for the (de)compression format of the ciphertext. Similar to the compression of the public and secret keys, the ciphertext is also compressed in `Kyber.CPAPKE.Enc` (Algorithm 2) for saving memory space and communication size. Different from the compression of the key pairs, the ciphertext is compressed to 10 bits and then 256 10-bit coefficients are concatenated and stored in the memory. Due to the fact that AVX2/AVX-512 instructions process data in multiples of 8 bits (e.g., 16 bits, 32 bits, or 64 bits), the concatenation of 10-bit coefficients becomes more complex. The specific process of the

compression of ciphertext in the one-way implementation is described in Figure 2 (a). We take the first 256 bits of the ciphertext as an example and each line of the figure represents a 256-bit memory space.

We design a compressing method for our multi-way implementation. Specifically, the ciphertext in our multi-way implementation is in `multi-way format`, so we concatenate the coefficients of the same polynomial together. Taking the 16-way implementation as an example, as shown in Figure 2 (b), we consider the first 512 bits of the ciphertext. Each line in the figure represents a 256-bit memory space. Compared to the one-way `compression` subroutine, we consume fewer instructions by employing the new compressing method, improving the efficiency of the ciphertext compression. Furthermore, converting ciphertext from `multi-way format` to `one-way format` to ensure compatibility with one-way implementation is relatively easy to achieve.

The `decompression` subroutine in Kyber.CPAPKE is the inverse operation of the `compression` subroutine. Therefore, we leverage the inverse compressing method mentioned above for the `decompression` subroutine. Similarly, the new decompressing method reduces the number of instructions required, enhancing the throughput of our multi-way implementation.

3.3 The multi-way Kyber.CCAKEM implementation

Our multi-way implementation of Kyber.CCAKEM, i.e., `KEM.KeyGen-Nway()`, `KEM.Encaps-Nway()` and `KEM.Decaps-Nway()`, shares the same procedure with the one-way AVX2 implementation of Kyber, except for minor modifications. So we do not give the multi-way algorithm descriptions of Kyber.CCAKEM.

Generally speaking, all subroutines are executed N times in three algorithms of the Kyber.CCAKEM, generating N pairs of public and secret keys, N ciphertexts, and N shared secrets in the end. Specifically, N equals 16 for the 16-way implementation and 32 for the 32-way implementation. We made the following optimizations to accelerate this computation.

Parallel utilization of SHA3. As demonstrated in [17], the impact of SHA3 to the overall running time in Kyber can be predominant. Our multi-way Kyber.CCAKEM leverages the XKCP library's multi-way parallel SHA3 implementation to effectively accelerate SHA3-related computations [30]. In the Algorithms 1 and 2, the GenMatrixA subroutine involves $k \times k$ polynomials. The SHAKE output stream employed for every polynomial is independent, allowing the one-way AVX2 implementation to utilize 4-way SHAKE for generating polynomials of the matrices. For Kyber768 parameter set, where one matrix consists of $3 \times 3 = 9$ polynomials, the `GenMatrixA` subroutine employs ($2\times$4-way+$1\times$one-way) SHAKE to generate these 9 polynomials. Our multi-way implementation of Kyber fully exploits the multi-way SHAKE to generate matrices in the GenMatrixA subroutine of Algorithms 7 and 8. For instance, in the 16-way implementation of Kyber768, each matrix comprises $3 \times 3 \times 16 = 144$ polynomials, enabling us to utilize ($9 \times 4 \times$4-way) SHAKE to generate all the polynomials, thereby significantly enhancing the efficiency of SHA3-related operations.

The SHA3-related operations in the KEM, as exemplified in lines 2, 3, and 5 in Algorithm 5 and lines 5, 8, and 10 in Algorithm 6, are serially used once in the one-way Kyber primitive, thereby precluding the exploitation of parallel multi-way SHA3. In contrast, the SHA3-related operations in our multi-way implementation of Kyber ought to be executed N times. Consequently, we can naturally make use of the power of parallelism by employing (4×4-way) or (4×8-way) SHA3-related operations in parallel for 16-way and 32-way implementation, respectively, which improves the efficiency of our implementation. We will later show that the full utilization of multi-way SHA3 yields significant speed improvements, as evidenced in Table 1.

4 Performance Evaluation and Comparison

In this section, we compare the throughput of the multi-way Kyber implementation with the cumulative throughput of multiple one-way official AVX2 Kyber implementations provided by the Kyber team [20], which is the state-of-the-art implementation.

The benchmark experiments are conducted on a desktop machine with Ubuntu 22.04 operating system, 11th Gen Intel(R) Core(TM) i7-11700K CPU(Rocket Lake) running at 3.60 GHz, 8 CPUs, and 16 GiB memory. We use GCC 11.3.0 to compile all programs. We disable the TurboBoost and Hyper-Threading techniques to ensure the reproduction of the experiments. Each experiment is repeated 200,000 times and the average results are measured.

As mentioned in Section 3.3, our multi-way implementation of Kyber allows for more efficient utilization of the multi-way SHA3 implementation compared to the one-way implementation, which can efficiently accelerate the SHA3-related computations. The performance comparison of one-way and multi-way SHA3-related operations is exhibited in Table 1. From Table 1 we can see that the multi-way SHA3 implementation has a significant throughput advantage compared to the one-way implementation.

The multi-way implementation of Kyber presented in this paper is built upon the one-way official AVX2 implementation of Kyber [20]. As shown in Table 2, our 16-way NTT and INTT implementations outperform the one-way AVX2 implementation by 24.5% and 19.6%, respectively. Furthermore, our 32-way NTT, coefficient-wise multiplication, and INTT implementations achieve remarkable speed-ups of 115.3%, 20.9%, and 92.3%, respectively, compared to the one-way AVX2 implementation. These performance gains are primarily attributed to the reduction of permutation instructions and modular reductions. Our 16-way coefficient-wise multiplication exhibits a slightly slower performance compared to the one-way AVX2 implementation, but its overall impact on the Kyber.CCAKEM protocol is negligible due to its relatively small time proportion.

A comprehensive comparison of the throughput for Kyber.CCAKEM is also presented in Table 2. Our 16-way implementation of Kyber achieves 36.0%, 54.6%, and 25.9% performance gains over the one-way AVX2 implementation

Table 1. The comparison of throughput (k operations/s = 1000 operations/s) for the one-way and multi-way SHA3-related operations.The multi-way implementation of SHA3 is directly leveraged from the XKCP library.

Schemes[1]	One-way [20]	4-way [30]	Speed-up	8-way [30]	Speed-up
absorbing[2]	33272.3k	109006.7k	227.6%	177147.9k	432.4%
squeezing	3327.7k[3]	11327.8k[3]	241.3%	21308.9k[3]	540.4%
SHA3-256	2826.3k	9631.1k	240.8%	18875.0k	567.8%
SHA3-512	2824.3k	9630.7k	241.0%	18511.2k	555.4%

[1] SHA3-related operations are mainly made up of `absorbing` and `squeezing` operations.
[2] The input length of `absorbing` is 32 bytes.
[3] If we set the parameter to let one-way `squeezing` generate one 168-byte block, then the 4-way `squeezing` will generate 4 168-byte blocks and the 8-way `squeezing` will generate 8 168-byte blocks at a time. The number of one-way, 4-way, and 8-way here all represent the cost of generating one 168-byte block.

Table 2. The comparison of throughput (k operations/s = 1000 operations/s) for the core polynomial arithmetic and Kyber.CCAKEM of one-way and multi-way implementations.

Schemes	One-way [20]	16-way	Speed-up[1]	32-way	Speed-up
NTT	15055.7k	18737.4k	24.5%	32420.0k	115.3%
Basemul	43735.0k	43195.4k	-1.20%	52892.1k	20.9%
INTT	15503.9k	18545.7k	19.6%	29807.6k	92.3%
KEM.KenGen	102.5k	139.5k	36.0%	185.2k	80.6%
KEM.Encaps	77.5k	119.8k	54.6%	178.4k	130.3%
KEM.Decaps	101.3k	127.6k	25.9%	153.3k	51.3%

[1] The speed-up in this article is calculated as the ratio of the throughput achieved by the multi-way implementation to the cumulative throughput of the one-way implementation.

for `KeyGen`, `Encaps`, and `Decaps`, respectively. Furthermore, our 32-way implementation of Kyber yields more substantial speed-ups of 80.6%, 130.3%, and 51.3% compared to the one-way AVX2 implementation for `KeyGen`, `Encaps`, and `Decaps`, respectively. As discussed in Section 3, these remarkable speed-ups are attributed to various optimization techniques, including NTT/INTT, reducing coefficient rearrangement, and more efficient utilization of the multi-way SHA3 implementation. Overall, our multi-way Kyber implementation demonstrates higher throughput compared to the one-way AVX2 implementation, which can effectively alleviate the computational pressure and reduce server response time.

5 Discussion

Our multi-way implementation of Kyber is constant-time and does not have any secret dependent branching or secret dependent memory accesses. Each of the multiple ways in our implementation of Kyber is identical to the one-way Kyber implementation, which means our implementation inherits the same level of security as the C implementation of Kyber and its AVX2 one-way implementation, without introducing any additional attack surfaces. Additionally, the batched generated key pairs will be stored in the memory, which is protected by the operating system (OS) security measures.

Then, we discuss the applicability of our multi-way implementation methodology to realistic scenarios. Multi-recipient KEM (mKEM) is a variant of KEM which allows encapsulating a single key for multiple recipients. Based on mKEM, Alwen et al. [4] proposed the mKyber, a mKEM construction based on Kyber, and the amKyber, a primitive with IND-CCA security against an attacker that can adaptively leak mKEM secret keys. The pseudocode diagrams of mKyber and amKyber involve the execution of certain operations N times, (e.g., `CpaEncV` subroutine in mKyber.Encap and amKyber.Encap, which includes NTT, SHA3, and `Compress` subroutines), facilitating the utilization of our proposed multi-way parallel implementation methodology to parallelize these operations. Since our multi-way implementation methodology is applicable to mKEM, the proposed multi-way methodology can also be leveraged to benefit various scenarios, including secure group messaging (e.g., Messaging Layer Security (MLS) [6]) and the other related use cases.

In the OpenSSL framework, an OpenSSL ENGINE acts as a container that facilitates the integration of optimized cryptographic implementations into TLS applications without the need to modify the source code of OpenSSL or the TLS application itself. The two works [8,32] demonstrate the feasibility of batched key generation on NTRU Prime and X25519, respectively, and showcase their integration into TLS via the ENGINE APIs. Both ENGINEs presented in these two papers support the integration of batched key generation, enabling the effortless integration of our multi-way key generation into OpenSSL through ENGINE reuse. Consequently, our multi-way implementation methodology can benefit the Hybrid TLS 1.3 (e.g., Kyber hybridized with X25519 [29]) and is also suitable for KEMTLS, an alternative TLS handshake protocol.

Integrating multi-way encapsulation and decapsulation into OpenSSL typically requires the use of an asynchronous programming framework to accumulate a sufficient number of handshake requests. Fortunately, the QTLS paper [16] has laid the groundwork for OpenSSL to support asynchronous programming frameworks, making the integration of multi-way encapsulation and decapsulation into OpenSSL via such frameworks a promising area for future exploration.

6 Conclusion

This paper explores a multi-way KEM implementation and instantiates a 16-way implementation on AVX2 and a 32-way implementation on AVX-512 for

Kyber. Several optimization techniques are proposed for improving the performance of the multi-way Kyber implementation. Specifically, we leverage the parallelism of AVX2 and AVX-512 to implement multi-way NTT-based polynomial multiplication, eliminating the need for permutation instructions for coefficient rearrangement in the one-way Kyber implementation on AVX2. Additionally, our multi-way implementation methodology overcomes the parallelization limitations of SHA3 in one-way implementation, which parallelizes all SHA3-related operations, thereby significantly enhancing the throughput and resulting in better performance compared to the one-way Kyber implementation. Through these optimizations, our 16-way and 32-way implementations achieve speed-ups of 36.0%/54.6%/25.9% and 80.6%/130.3%/51.3%, respectively, for `KeyGen()`, `Encaps()` and `Decaps()` compared to the one-way Kyber implementation with AVX2. Finally, we validate the feasibility of our multi-way implementation methodology in real-world application scenarios.

Acknowledgements This work was partially supported by the National Natural Science Foundation of China under Grant 62071222 and the Joint funds of the National Natural Science Foundation of China under Grant U20A20176.

References

1. Abdulrahman, A., Hwang, V., Kannwischer, M.J., Sprenkels, A.: Faster Kyber and Dilithium on the Cortex-M4. In: Ateniese, G., Venturi, D. (eds.) Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13269, pp. 853–871. Springer (2022)
2. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R., et al.: Status report on the third round of the NIST post-quantum cryptography standardization process. US Department of Commerce, NIST (2022)
3. Alkim, E., Bilgin, Y.A., Cenk, M., Gérard, F.: Cortex-M4 optimizations for {R, M} LWE schemes. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 336–357 (2020)
4. Alwen, J., Kiltz, E., Massimo, J., Mularczyk, M., Prest, T., Schwabe, P.: How multi-recipient KEMs can help the deployment of post-quantum cryptography
5. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Kyber algorithm specifications and supporting documentation. NIST PQC Round **2**(4), 1–43 (2019)
6. Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., Cohn-Gordon, K.: The messaging layer security (MLS) protocol. RFC **9420**, 1–132 (2023). https://doi.org/10.17487/RFC9420, https://doi.org/10.17487/RFC9420
7. Becker, H., Kannwischer, M.J.: Hybrid scalar/vector implementations of Keccak and SPHINCS$^+$ on AArch64. In: Isobe, T., Sarkar, S. (eds.) Progress in Cryptology - INDOCRYPT 2022 - 23rd International Conference on Cryptology in India, Kolkata, India, December 11-14, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13774, pp. 272–293. Springer (2022). https://doi.org/10.1007/978-3-031-22912-1_12, https://doi.org/10.1007/978-3-031-22912-1_12

8. Bernstein, D.J., Brumley, B.B., Chen, M., Tuveri, N.: OpenSSLNTRU: Faster post-quantum TLS key exchange. In: Butler, K.R.B., Thomas, K. (eds.) 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022. pp. 845–862. USENIX Association (2022)
9. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - Kyber: A CCA-Secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018. pp. 353–367. IEEE (2018)
10. Botros, L., Kannwischer, M.J., Schwabe, P.: Memory-efficient high-speed implementation of Kyber on Cortex-M4. In: Progress in Cryptology–AFRICACRYPT 2019: 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9–11, 2019, Proceedings 11. pp. 209–228. Springer (2019)
11. Cheng, H., Fotiadis, G., Großschädl, J., Ryan, P.Y.A.: Highly vectorized SIKE for AVX-512. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(2), 41–68 (2022). https://doi.org/10.46586/TCHES.V2022.I2.41-68, https://doi.org/10.46586/tches.v2022.i2.41-68
12. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Mathematics of computation **19**(90), 297–301 (1965)
13. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1666, pp. 537–554. Springer (1999)
14. Gentleman, W.M., Sande, G.: Fast fourier transforms: for fun and profit. In: Proceedings of the November 7-10, 1966, fall joint computer conference. pp. 563–578 (1966)
15. Greconici, D.: Kyber on RISC-V. Master's thesis (2020)
16. Hu, X., Wei, C., Li, J., Will, B., Yu, P., Gong, L., Guan, H.: QTLS: high-performance TLS asynchronous offload framework with intel® quickassist technology. In: Hollingsworth, J.K., Keidar, I. (eds.) Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019. pp. 158–172. ACM (2019). https://doi.org/10.1145/3293883.3295705, https://doi.org/10.1145/3293883.3295705
17. Huang, J., Adomnicai, A., Zhang, J., Dai, W., Liu, Y., Cheung, R.C.C., Koç, Ç.K., Chen, D.: Revisiting Keccak and Dilithium implementations on ARMv7-M. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2024**(2), 1–24 (2024). https://doi.org/10.46586/TCHES.V2024.I2.1-24, https://doi.org/10.46586/tches.v2024.i2.1-24
18. Huang, J., Zhang, J., Zhao, H., Liu, Z., Cheung, R.C.C., Koç, Ç.K., Chen, D.: Improved Plantard arithmetic for lattice-based cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(4), 614–636 (2022)
19. Huang, J., Zhao, H., Zhang, J., Dai, W., Zhou, L., Cheung, R.C.C., Koç, Ç.K., Chen, D.: Yet another improvement of Plantard arithmetic for faster Kyber on low-end 32-bit IoT devices. IEEE Trans. Inf. Forensics Secur. **19**, 3800–3813 (2024). https://doi.org/10.1109/TIFS.2024.3371369, https://doi.org/10.1109/TIFS.2024.3371369
20. Kyber-team: The Kyber's optimized implementation using AVX2. https://github.com/pq-crystals/kyber/tree/a621b8dde405cc507cbcfc5f794570a4f98d69cc/avx2 (2023), Accessed: 2023-07-03

21. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography **75**(3), 565–599 (2015)
22. Mike, W.: High traffic surges got your website down? Here's why, https://www.keysight.com/blogs/tech/software-testing/2022/11/15/high-traffic-surges-got-your-website-down
23. National Institute of Standards and Technology: Module-lattice-based digital signature standard (2023). https://doi.org/10.6028/NIST.FIPS.204.ipd
24. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard (2023). https://doi.org/10.6028/NIST.FIPS.203.ipd
25. National Institute of Standards and Technology: Stateless hash-based digital signature standard (2023). https://doi.org/10.6028/NIST.FIPS.205.ipd
26. Roy, S.S.: Saberx4: High-throughput software implementation of Saber key encapsulation mechanism. In: 37th IEEE International Conference on Computer Design, ICCD 2019, Abu Dhabi, United Arab Emirates, November 17-20, 2019. pp. 321–324. IEEE (2019). https://doi.org/10.1109/ICCD46524.2019.00050, https://doi.org/10.1109/ICCD46524.2019.00050
27. Seiler, G.: Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. IACR Cryptol. ePrint Arch. p. 39 (2018)
28. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Rev. **41**(2), 303–332 (1999)
29. wolfSSL: Coming soon: Kyber (ML-KEM) hybridized with X25519 in wolfSSH, https://www.wolfssl.com/coming-soon-kyber-ml-kem-hybridized-with-x25519-in-wolfssh/
30. XKCP-team: The eXtended Keccak Code Package. https://github.com/XKCP/XKCP/tree/86110a0be2c5463c8278807da8395e5fd9912f20 (2023), Accessed: 2024-05-10
31. Zhang, J., Huang, J., Liu, Z., Roy, S.S.: Time-memory trade-offs for Saber+ on memory-constrained RISC-V platform. IEEE Trans. Computers **71**(11), 2996–3007 (2022)
32. Zhang, J., Huang, J., Zhao, L., Chen, D., Çetin Kaya Koç: ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. In: 33st USENIX Security Symposium, USENIX Security 2024
33. Zheng, J., Zhu, H., Song, Z., Wang, Z., Zhao, Y.: Optimized vectorization implementation of CRYSTALS-Dilithium. CoRR **abs/2306.01989** (2023), https://doi.org/10.48550/arXiv.2306.01989

Appendix A Format sequence conversion

The following figure illustrates the process of format sequence conversion between `one-way format` and `16-way format`, using the first 16 coefficients of each polynomial as an example. In the figure, every letter denotes an independent polynomial, with the subscript indicating the coefficient index. The notation `shuffleN` depicted in the figure refers to the macro defined in our implementation, whereas `vpshufb` and `vpermq` are two SIMD instructions.

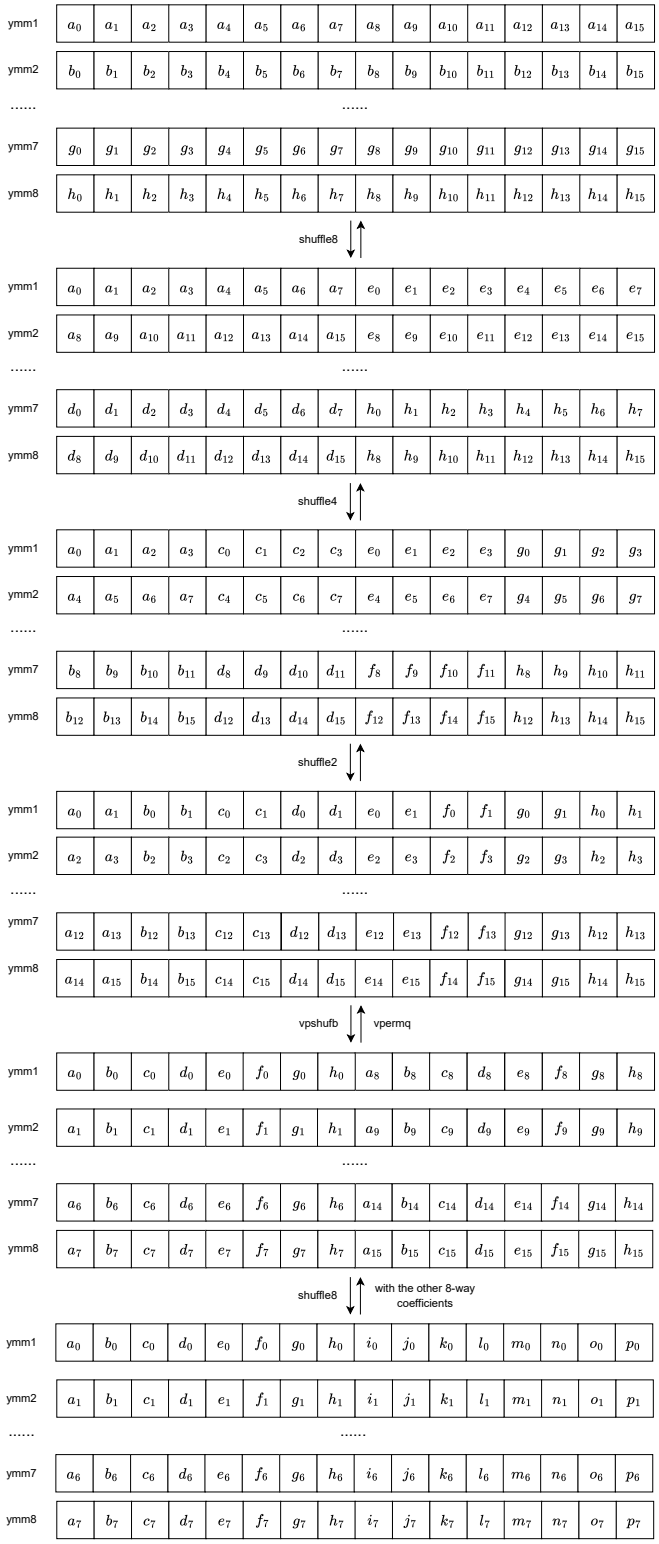


Fig. 3. The process of format conversion subroutine