Optimized Implementation of ML-KEM on ARMv9-A with SVE2 and SME

Hanyu Wei[1], Wenqian Li[1], Shiyu Shen[2(✉)], Hao Yang[2], and Yunlei Zhao[1(✉)]

[1] Fudan University, Shanghai, China
hywei24@m.fudan.edu.cn, ylzhao@fudan.edu.cn
[2] City University of Hong Kong, Hong Kong, China
crypto@sher1e.dev

Abstract. As quantum computing continues to advance, traditional public-key cryptosystems face increasing vulnerability, necessitating a global transition toward post-quantum cryptography (PQC). A primary challenge for both cryptographers and system architects is the efficient integration of PQC into high-performance computing platforms. ARM, a dominant processor architecture, has recently introduced ARMv9-A to accelerate modern workloads such as artificial intelligence and cloud computing. Leveraging its Scalable Vector Extension 2 (SVE2) and Scalable Matrix Extension (SME), ARMv9-A provides sophisticated hardware support for high-performance computing. This architectural evolution motivates the need for efficient implementations of PQC schemes on the new architecture. In this work, we present a highly optimized implementation of ML-KEM, the post-quantum key encapsulation mechanism (KEM) standardized by NIST as FIPS 203, on the ARMv9-A architecture. We redesign the polynomial computation pipeline to achieve deep alignment with the vector and matrix execution units. Our optimizations encompass refined modular arithmetic and highly vectorized polynomial operations. Specifically, we propose two NTT variants tailored to the architectural features of SVE2 and SME: the vector-based NTT (VecNTT) and the matrix-based NTT (MatNTT), which effectively utilize layer fusion and optimized data access patterns. Experimental results on the Apple M4 Pro processor demonstrate that VecNTT and MatNTT achieve performance improvements of up to 7.18× and 7.77×, respectively, compared to the reference implementation. Furthermore, the matrix-vector polynomial multiplication, which is the primary computational bottleneck of ML-KEM, is accelerated by up to 5.27×. Our full ML-KEM implementation achieves a 52.47% to 60.09% speedup in key encapsulation across all security levels. To the best of our knowledge, this is the first work to implement and evaluate ML-KEM leveraging SVE2 and SME on real ARMv9-A hardware, providing a practical foundation for future PQC deployments on next-generation ARM platforms.

Keywords: Post-Quantum Cryptography · ML-KEM · NTT · SVE2 · SME · ARMv9-A

1 Introduction

The rapid advancement of quantum computing poses a significant threat to widely used public-key cryptosystems such as RSA and Elliptic Curve Cryptography (ECC), which rely on the hardness of integer factorization and discrete logarithm problems. To ensure long-term data security in the quantum era, the National Institute of Standards and Technology (NIST) initiated a global standardization process for post-quantum cryptography (PQC) in 2016. After several rounds of rigorous evaluation, NIST officially released the first set of PQC standards in 2024, including ML-KEM [19], ML-DSA [20], SLH-DSA [21], and FN-DSA (Falcon [9]). In 2025, NIST further announced HQC [11] as a backup KEM to enhance the diversity and robustness of the cryptographic ecosystem [22]. Among these algorithms, ML-KEM, formerly known as CRYSTALS-Kyber [4], is the first lattice-based KEM to be standardized. It is widely regarded as the primary choice for securing future communication protocols due to its excellent balance of security, efficiency, and communication overhead. Consequently, the efficient implementation and optimization of ML-KEM across diverse computing platforms have become a critical research priority for both industry and academia. ML-KEM relies on the hardness of the module learning with errors (MLWE) problem. As with other lattice-based schemes, its performance is largely determined by the efficiency of polynomial arithmetic, particularly polynomial multiplication based on the number-theoretic transform (NTT).

As artificial intelligence (AI) continues to advance, modern computing platforms are increasingly equipped with high-performance processors to handle growing data workloads. These platforms typically provide powerful vector processing capabilities that support the efficient execution of computationally intensive tasks. Since these processors are widely used in critical application domains such as healthcare, finance, and autonomous systems, the demand for high-performance PQC is becoming increasingly urgent. ARM processors have become a foundation of modern computing, supporting workloads ranging from edge AI to cloud infrastructure. The transition to the ARMv9-A architecture represents a significant milestone, introducing advanced features like Scalable Vector Extension 2 (SVE2) and Scalable Matrix Extension (SME) to handle next-generation computational demands.

The recent commercial availability of ARMv9-A hardware, such as the Apple M4 Pro SoC [2], MediaTek Dimensity 7200 [18], and Samsung Exynos 1580 [24], marks the beginning of a new era for high-performance mobile and desktop computing. However, the optimization of ML-KEM for these new extensions remains underexplored. While NTT has been extensively optimized for SIMD architectures like Neon and AVX2, the architectural shift to SVE2 and SME introduces both new challenges and opportunities, such as different vector length and matrix-oriented storage. Optimizing these core kernels is critical to achieving practical PQC performance on modern ARM platforms and bridging the gap between theoretical standards and real-world deployment.

1.1 Related Work

The optimization of ML-KEM (formerly Kyber) has been extensively explored across various computing platforms. In this section, we review the state-of-the-art implementations on different architectures, with a particular focus on ARM-based systems and the evolution of ARM processor capabilities.

The computational performance of ML-KEM has been investigated on high-end CPUs, GPUs, and hardware accelerators. On x86 architectures, Seiler [26] provided an early high-performance implementation using AVX2, which served as the basis for many subsequent optimizations. More recently, Yu et al. [30] further pushed the performance boundaries using AVX-512. For massively parallel scenarios, several works have utilized GPUs to accelerate Kyber, achieving high throughput for batch processing [13, 15, 31]. The RISC-V architecture, with its open-source nature and flexible extensions, has also become a prominent platform for PQC optimization. Li et al. [16] proposed a scalable SIMD RISC-V processor with customized vector extensions specifically for Kyber. More recently, Bevin et al. [6] introduced a specialized Montgomery reduction instruction set extension (ISE) to further accelerate the modular arithmetic in both ML-KEM and ML-DSA. Additionally, FPGA-based hardware implementations have focused on secure and efficient hardware designs, particularly for NTT-based polynomial operations [17, 27, 29].

ARM processors are a critical target for PQC deployment due to their ubiquity in mobile and embedded devices. On the low-power ARM M-profile cores, Botros et al. [7] and Abdulrahman et al. [1] utilized the DSP extension of the Cortex-M4 to optimize modular multiplication and the NTT. For the more powerful ARM A-profile processors, initial efforts focused on the Neon instruction set. Sanal et al. [25] provided a compact ARM64 implementation, while Becker et al. [5] introduced advanced techniques such as optimized Barrett multiplication and asymmetric multiplication to achieve significant speedups. Recent research has also begun exploring the ARMv9-A architecture, such as optimizations for the ML-DSA signature scheme using SVE2 [28], but a comprehensive treatment of ML-KEM on the latest ARM extensions, particularly SME, is still lacking.

The ARM A-profile architecture has evolved from fixed-length SIMD towards more flexible and powerful vector and matrix extensions. The AArch64 instruction set introduced the 128-bit fixed-length Neon extension, which has been the standard for cryptographic acceleration for years. To address the limitations of fixed-length vectors, ARM introduced the Scalable Vector Extension (SVE) in ARMv8.2-A, enabling vector-length agnostic (VLA) programming. ARMv9-A further introduced SVE2. The most recent major advancement is the SME, which introduces ZA storage and outer-product operations to accelerate matrix-intensive computations. The Apple M4 Pro SoC [2] is the first consumer-grade processor to support both SVE2 and SME, offering a significant opportunity to redefine the performance limits of post-quantum cryptography on ARM platforms. Our work builds upon these prior advancements by providing the first comprehensive implementation of ML-KEM that leverages the full potential of both SVE2 and SME.

1.2 Contribution

In this work, we bridge the gap between emerging ARMv9-A hardware features and the performance requirements of post-quantum cryptography. To the best of our knowledge, this is the first work to provide a comprehensive optimization and evaluation of ML-KEM on ARMv9-A leveraging both SVE2 and SME. Our primary contributions are summarized as follows:

- **Instruction-Level Modular Arithmetic Optimization:** We optimize Montgomery and Barrett modular multiplication specifically for SVE2, enabling parallel execution of up to 32 operations per instruction. Our implementation minimizes register pressure and maximizes throughput for the underlying arithmetic of ML-KEM.
- **Tailored NTT Architectures:** We propose two distinct NTT strategies to exploit the different strengths of ARMv9-A: VecNTT and MatNTT. VecNTT leverages vector shuffle instructions to manage data movement within and across registers. MatNTT innovatively utilizes the SME matrix features to optimize data access patterns, combining computation with a hybrid vector-matrix processing approach.
- **Architecture-Aligned Pipeline Design:** We design a fine-grained instruction pipeline for polynomial operations that aligns with the out-of-order execution characteristics of modern ARMv9-A cores. By carefully interleaving ZA storage accesses with butterfly operations, we minimize data dependencies and maximize functional unit utilization.
- **Comprehensive Hardware Evaluation:** We conduct an extensive benchmarking of our optimizations on a physical Apple M4 Pro processor. Our results demonstrate substantial speedups over reference implementations, achieving a 7.77× speedup in MatNTT and a 52.47%-60.09% speedup in overall ML-KEM encapsulation, establishing a new state-of-the-art implementation for ARMv9-A.

2 Preliminaries

2.1 ML-KEM

ML-KEM [19] is a lattice-based key encapsulation mechanism derived from CRYSTALS-Kyber [4]. It defines three parameter sets: ML-KEM-512, ML-KEM-768, and ML-KEM-1024, tailored to satisfy the NIST security levels I, III, and V, respectively. The construction of ML-KEM follows a two-stage design. First, a public-key encryption (PKE) scheme, referred to as K-PKE, is constructed based on the Module Learning With Errors (MLWE) problem. Second, the resulting PKE is converted into a KEM using the Fujisaki-Okamoto (FO) transform [10, 14], which is widely believed to achieve IND-CCA2 security in the random oracle model.

Algorithm 1 K-PKE.KeyGen(d) [19]

Input: Randomness $d \in \mathbb{B}^{32}$
Output: Encryption key $ek_{\text{PKE}} \in \mathbb{B}^{384k+32}$ and decryption key $dk_{\text{PKE}} \in \mathbb{B}^{384k}$

$(\rho, \sigma) \leftarrow \mathsf{G}(d\|k)$ ▷ Expand to pseudorandom seeds
$N \leftarrow 0$
Generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_{256}^{q})^{k \times k}$ using ρ
for $i = 0$ to $k - 1$ **do**
  $\mathbf{s}[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
  $N \leftarrow N + 1$
end for
for $i = 0$ to $k - 1$ **do**
  $\mathbf{e}[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
  $N \leftarrow N + 1$
end for
$\hat{\mathbf{s}} \leftarrow \text{NTT}(\mathbf{s})$, $\hat{\mathbf{e}} \leftarrow \text{NTT}(\mathbf{e})$ ▷ Transform to NTT domain
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ Noisy linear system
$ek_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{t}})\|\rho$
$dk_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{s}})$
return $(ek_{\text{PKE}}, dk_{\text{PKE}})$

Algorithm 2 K-PKE.Encrypt(ek_{PKE}, m, r) [19]

Input: Encryption key $ek_{\text{PKE}} \in \mathbb{B}^{384k+32}$, message $m \in \mathbb{B}^{32}$, randomness $r \in \mathbb{B}^{32}$
Output: Ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$

$N \leftarrow 0$
$\hat{\mathbf{t}} \leftarrow \text{ByteDecode}_{12}(ek_{\text{PKE}}[0 : 384k])$
$\rho \leftarrow ek_{\text{PKE}}[384k : 384k + 32]$ ▷ Extract seed
Generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_{256}^{q})^{k \times k}$ using ρ
for $i = 0$ to $k - 1$ **do**
  $\mathbf{y}[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(r, N))$
  $N \leftarrow N + 1$
end for
for $i = 0$ to $k - 1$ **do**
  $\mathbf{e}_1[i] \leftarrow \text{SamplePolyCBD}_{\eta_2}(\text{PRF}_{\eta_2}(r, N))$
  $N \leftarrow N + 1$
end for
$e_2 \leftarrow \text{SamplePolyCBD}_{\eta_2}(\text{PRF}_{\eta_2}(r, N))$
$\hat{\mathbf{y}} \leftarrow \text{NTT}(\mathbf{y})$ ▷ Transform to NTT domain
$\mathbf{u} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}^{\top} \circ \hat{\mathbf{y}}) + \mathbf{e}_1$
$\mu \leftarrow \text{Decompress}_1(\text{ByteDecode}_1(m))$
$v \leftarrow \text{NTT}^{-1}(\hat{\mathbf{t}}^{\top} \circ \hat{\mathbf{y}}) + e_2 + \mu$
$c_1 \leftarrow \text{ByteEncode}_{d_u}(\text{Compress}_{d_u}(\mathbf{u}))$
$c_2 \leftarrow \text{ByteEncode}_{d_v}(\text{Compress}_{d_v}(v))$
return $c \leftarrow (c_1\|c_2)$

Algorithm 3 K-PKE.Decrypt(dk_{PKE}, c) [19]

Input: Decryption key $dk_{\text{PKE}} \in \mathbb{B}^{384k}$, ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$
Output: Message $m \in \mathbb{B}^{32}$
1: $c_1 \leftarrow c[0 : 32d_u k]$, $c_2 \leftarrow c[32d_u k : 32(d_u k + d_v)]$ ▷ Extract ciphertext components
2: $\mathbf{u}' \leftarrow \text{Decompress}_{d_u}(\text{ByteDecode}_{d_u}(c_1))$
3: $v' \leftarrow \text{Decompress}_{d_v}(\text{ByteDecode}_{d_v}(c_2))$
4: $\hat{\mathbf{s}} \leftarrow \text{ByteDecode}_{12}(dk_{\text{PKE}})$ ▷ Decode secret key
5: $w \leftarrow v' - \text{NTT}^{-1}(\hat{\mathbf{s}}^\top \circ \text{NTT}(\mathbf{u}'))$ ▷ Recover message polynomial
6: $m \leftarrow \text{ByteEncode}_1(\text{Compress}_1(w))$
7: **return** m

The core component K-PKE consists of three algorithms: key generation (Algorithm 1), encryption (Algorithm 2), and decryption (Algorithm 3). The computational efficiency of these algorithms is heavily reliant on polynomial arithmetic within R_q. Specifically, matrix-vector multiplications and inner products are performed in the Number-Theoretic Transform (NTT) domain to accelerate polynomial multiplication. For instance, during encryption, the matrix $\hat{\mathbf{A}}$ and vectors $\hat{\mathbf{y}}$ are represented in the NTT domain, allowing the operations $\hat{\mathbf{A}}^\top \circ \hat{\mathbf{y}}$ and $\hat{\mathbf{t}}^\top \circ \hat{\mathbf{y}}$ to be computed via efficient point-wise multiplications.

2.2 Modular Arithmetic

Modular arithmetic over a prime modulus q is fundamental in lattice-based cryptography. Efficient techniques such as Montgomery and Barrett eliminate costly division by replacing it with multiplication, addition, and bit-shifting.

Montgomery-Based Modular Multiplication. Montgomery reduction facilitates modular multiplication by operating in the Montgomery domain, where an integer a is represented as $a' = a\beta \bmod q$, with $\beta = 2^k > q$. This allows for modular reduction without explicit division by q. As detailed in Algorithm 4, the process utilizes shift-based approximations and a precomputed constant $T = -q^{-1} \bmod \beta$. This method is particularly efficient for general modular multiplication in SIMD environments.

Barrett-Based Modular Multiplication. Barrett reduction provides an alternative approach by approximating the quotient a/q. It utilizes a precomputed constant $v = \lfloor \frac{2^k \beta}{q} \rfloor$ to estimate $p = \lfloor \frac{a}{q} \rfloor$ via $\hat{p} = \lfloor \frac{av}{2^k \beta} \rfloor$, followed by $r = a - \hat{p} \cdot q$. This technique is highly effective when one of the multiplicands is known (e.g., twiddle factors in NTT), as the constant v can be precomputed to further accelerate the operation. The procedure is summarized in Algorithm 5.

2.3 NTT-based Multiplication

Polynomial multiplication in the ring $R_q = \mathbb{Z}_q[x]/(x^{256} + 1)$ is most efficiently performed using the number-theoretic transform (NTT). For the ML-KEM mod-

Algorithm 4 Montgomery Multiplication

Input: Modulus q, $\beta = 2^k > q$, $a, b \in \mathbb{Z}_q$, $T = -q^{-1} \bmod \beta$
Output: $\mathsf{Mont}_q(a, b) = ab\beta^{-1} \bmod q$
1: $z \leftarrow \lfloor \frac{ab}{\beta} \rfloor$
2: $k \leftarrow ab \cdot T \bmod \beta$
3: $c \leftarrow \lfloor \frac{kq}{\beta} \rfloor$
4: **return** $z - c$

Algorithm 5 Barrett Multiplication

Input: Modulus q, $\beta = 2^k > q$, $a, b \in \mathbb{Z}_q$, precomputed $v = \left\lfloor \frac{b \cdot \beta}{q} \right\rfloor$
Output: $\mathsf{Bar}_q(a, b) = ab \bmod q$
1: $z \leftarrow ab$
2: $t \leftarrow \lfloor \frac{a \cdot v}{\beta} \rfloor$
3: $r \leftarrow z - t \cdot q$
4: **return** r

ulus $q = 3329$, where $q - 1 = 2^8 \cdot 13$, the base field $\mathbb{Z}_q$ contains primitive 256-th roots of unity but lacks primitive 512-th roots. Consequently, the defining polynomial $x^{256} + 1$ does not factor completely into linear terms but instead decomposes into 128 quadratic factors modulo q. Thus, the NTT of a polynomial $f \in R_q$ is a vector of 128 polynomials of degree one.

The in-place implementations of the NTT often output results in bit-reversed order to optimize performance; we adopt this convention for our implementation. Let $\zeta = 17$ be a primitive 256-th root of unity modulo q. The polynomial $x^{256}+1$ can be factored as:

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

where $\mathrm{br}_7(i)$ denotes the 7-bit bit reversal of i. The NTT of $f \in R_q$ is defined as the vector of residues:

$$\mathrm{NTT}(f) = (f \bmod x^2 - \zeta^{2\mathrm{br}_7(0)+1}, \ldots, f \bmod x^2 - \zeta^{2\mathrm{br}_7(127)+1}).$$

This representation facilitates efficient multiplication: given polynomials $f, g \in R_q$, their product $h = f \cdot g$ is computed as $h = \mathrm{NTT}^{-1}(\mathrm{NTT}(f) \circ \mathrm{NTT}(g))$, where $\circ$ denotes basecase multiplication of the 128 pairs of linear polynomials modulo $(x^2 - \zeta^{2\mathrm{br}_7(i)+1})$. The NTT is computed using divide-and-conquer butterfly operations, primarily the Cooley-Tukey (CT) [8] and Gentleman-Sande (GS) [12] butterfly. As illustrated in Algorithm 6 and Algorithm 7, the CT butterfly is typically used for the forward NTT, while the GS butterfly is employed for the inverse NTT to restore the coefficients to their natural order.

2.4 SVE2 and SME Extensions

The ARMv9-A architecture introduces SVE2 and SME to significantly enhance support for data-parallel workloads. SVE2 provides 32 scalable vector registers (`z0`–`z31`) with implementation-defined lengths (VL) ranging from 128 to 2048 bits. Unlike traditional fixed-width SIMD, SVE2 supports vector-length agnostic (VLA) programming, allowing code to run unmodified on different hardware

Algorithm 6 Forward NTT

Input: Polynomial $f \in \mathbb{Z}_q[x]/(x^{256}+1)$, twiddle factors $\zeta^{2\mathrm{br}_7(k)+1} \bmod q$ for $k = 1, \ldots, 127$
Output: Polynomial $\hat{f} = \mathrm{NTT}(f)$

$k \leftarrow 1, len \leftarrow 128$
while $len >= 2$ **do**
  for $j = 0$ to 255 by len **do**
    $\zeta \leftarrow \zeta^{\mathrm{br}_7(k)}$
    $k \leftarrow k + 1$
    for $i = j$ to $j + len - 1$ **do**
      $tmp \leftarrow \zeta \cdot f_{i+len}$
      $f_{i+len} \leftarrow f_i - tmp$
      $f_i \leftarrow f_i + tmp$
    end for
  end for
  $len \leftarrow len/2$
end while
return $\hat{f}$

Algorithm 7 Inverse NTT

Input: Polynomial $\hat{f} \in \mathbb{Z}_q[x]/(x^{256}+1)$, twiddle factors $\zeta^{\mathrm{br}_7(k)} \bmod q$ for $k = 127, \ldots, 1$
Output: Polynomial $f = \mathrm{NTT}^{-1}(\hat{f})$

$k \leftarrow 127$, $len \leftarrow 2$
while $len <= 128$ **do**
  for $j = 0$ to 255 by len **do**
    $\zeta \leftarrow \zeta^{\mathrm{br}_7(k)}$
    $k \leftarrow k - 1$
    for $i = j$ to $j + len - 1$ **do**
      $tmp \leftarrow \hat{f}_i$
      $\hat{f}_i \leftarrow \frac{1}{2}(\hat{f}_i + \hat{f}_{i+len})$
      $\hat{f}_{i+len} \leftarrow \frac{\zeta}{2}((tmp - \hat{f}_{i+len}))$
    end for
  end for
  $len \leftarrow 2 \cdot len$
end while
return f

configurations. Predicate registers (`p0`–`p15`) enable per-lane masking, facilitating complex control flow within vectorized loops.

SME further extends these capabilities by introducing the streaming SVE mode and ZA storage, which is a dedicated 2D matrix register file. This is particularly beneficial for matrix-intensive operations. However, entering streaming mode via `smstart` disables the standard Neon units, necessitating a purely SVE2/SME-based approach for the computation kernel. In this work, we target the Apple M4 Pro SoC [2], the first mass-market processor to feature both extensions, with a SVE2 vector length of 512 bits.

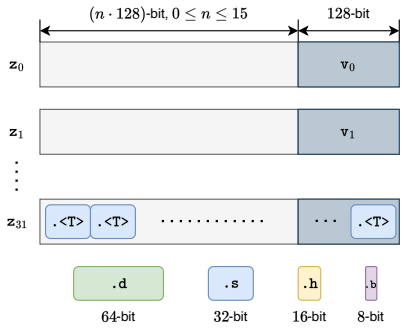

Fig. 1. Structural overview of SVE2 scalable vector registers.

3 Implementation

This section details our optimized implementation of ML-KEM on the ARMv9-A architecture, specifically leveraging SVE2 and SME. We first present an architectural pipeline design tailored for out-of-order execution. Then, we describe the optimization of vectorized modular arithmetic. Finally, we propose two distinct NTT methods (VecNTT and MatNTT) along with their respective instruction scheduling strategies.

3.1 Pipeline Design

As shown in Figure 2, the processor pipeline consists of two major components: instruction delivery and instruction processing [3]. The instruction delivery component handles instruction fetch and decode, as well as branch prediction, to ensure that the instruction buffer remains saturated and ready for execution. The instruction processing component supports out-of-order execution with a deep instruction window. Instructions are fetched sequentially and stored in a buffer, from which the processor selects ready instructions those whose operands are already available and issues them out of order. This enables concurrent execution of multiple instructions across different functional units. The architecture features three main execution units: integer, SIMD/floating-point (FP), and load/store. To fully exploit out-of-order execution, our implementation interleaves instructions across these units wherever possible. Load/store instructions such as `ld1w`, `ld2w`, and `st1w` are dispatched to the load/store unit; scalar arithmetic instructions like `add` and `sub` on general-purpose registers are handled by the integer unit; and SVE2 vector instructions, including `add`, `sub`, `mls`, and `sqrdmulh`, are executed by the SIMD unit.

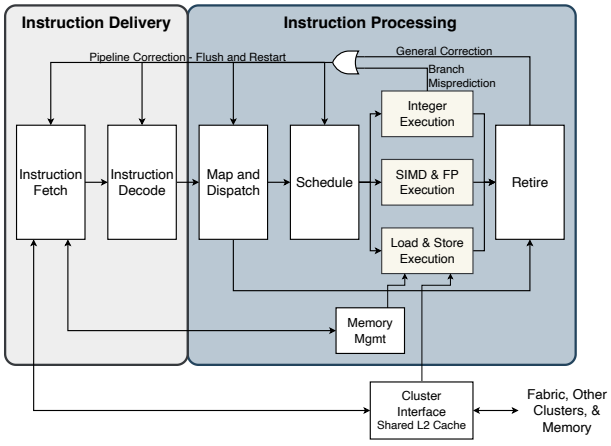

Fig. 2. Abstract View of the Processor Pipeline [3].

Although the underlying microarchitecture supports out-of-order execution, we carefully schedule vector instructions to minimize data dependencies and balance the usage of execution ports. This arrangement improves pipeline utilization and maximizes throughput under the out-of-order execution model. By organizing instructions in a recurring pattern that begins with `ld1w` to load data, continues with SVE instructions such as `mul` and `sqrdmulh` for parallel arithmetic operations, and ends with `st1w` to store results, we ensure that the processor loads a new group of data while performing computations on the previous one. Since load and store operations are independent of ongoing arithmetic computations, this interleaving exposes greater instruction-level parallelism, resulting in improved pipeline efficiency and overall performance.

3.2 Modular Arithmetic

Barrett Multiplication. When one of the multiplicands is known in advance, we adopt Barrett multiplication, which is particularly common in NTT computations due to frequent multiplications with fixed twiddle factors. Barrett multiplication requires only three SVE instructions, as shown in Algorithm 8, and achieves 32-way parallelism. Similar to Neon, however, SVE does not provide an instruction to directly obtain the high n bits of a multiplication result. Instead, the `sqrdmulh` instruction performs a saturating rounded doubling multiply-high, which effectively computes $2ab$ and returns the high bits of the result. Therefore, we must pre-divide the precomputed data of the known operand by 2 during preprocessing to compensate for the behavior of `sqrdmulh`. In all subsequent NTT-related computations where one operand is constant, we consistently employ Barrett multiplication to maximize performance.

Algorithm 8 Barrett Multiplication with One Constant Operand

Input: Coefficient register `a` $= (a_0, \ldots, a_{31}) \in \mathbb{Z}_q$, constant operand `b` $= (b_0, \ldots, b_{31}) \in \mathbb{Z}_q$, precomputed `bR` $= (b'_0, \ldots, b'_{31})$ with $b'_i = \lfloor \frac{b_i \cdot 2^{16}}{2q} \rceil$, modulus register `q`, temporary register `tmp`, predicate register `p0`.

Output: Product register `c` $= (a_0 b_0 \bmod q, \ldots, a_{31} b_{31} \bmod q)$

```
1: mul  c.h, a.h, b.h
2: sqrdmulh  tmp.h, a.h, bR.h
3: mls  c.h, p0/m,tmp.h, q.h
```

Montgomery Multiplication. When both multiplicands are unknown, Montgomery multiplication is used to compute modular products in $\mathbb{Z}_q$, entirely within registers to avoid unnecessary memory access. Algorithm 9 presents the instruction sequence. Unlike prior Neon-based implementations [5] that rely on `uzp` for vector unpacking, we utilize the `trn` (transpose) instruction to interleave even and odd lanes. This approach aligns better with our data layout and streamlines the instruction sequence.

Algorithm 9 Montgomery Multiplication of Two SVE2 Registers

Input: Coefficient register $\mathtt{a} \in \mathbb{Z}_q, \mathtt{b} \in \mathbb{Z}_q$ with each element 16-bit, parameters register `q`, `qinv` $= (q^{-1} \bmod R, \ldots, q^{-1} \bmod R)$ with each element 16-bit, where $R = 2^{16}$, temporary register `tmp`, `tmp2`, `tmp3`.

Output: Product register $\mathtt{c} = (a_0 b_0 R^{-1}, \ldots, a_{31} b_{31} R^{-1}) \in \mathbb{Z}_q$, where $R^{-1} = 2^{-16} \bmod q$.

```
1: smullb  tmp.s, a.h, b.h
2: smullt  tmp2.s, a.h, b.h
3: trn1  tmp3.h, tmp.h, tmp2.h
4: mul  tmp3.h, tmp3.h, qinv.h
5: smlslb  tmp.s, tmp3.h, q.h
6: smlslt  tmp2.s, tmp3.h, q.h
7: trn2  c.h, tmp.h, tmp2.h
```

Asymmetric Element-wise Multiplication. In ML-KEM, the NTT decomposition over R_q results in 128 quadratic factors of the form $x^2 - \zeta$. This requires 128 pairs of linear polynomial multiplications to perform element-wise multiplication in the NTT domain. We adopt the asymmetric method from [5] and adapt it for the SVE2 instruction set. For example, consider the polynomial multiplication $(a_0+a_1x)(b_0+b_1x) \bmod x^2 - \zeta = (a_0b_0+a_1b_1\zeta)+(a_0b_1+a_1b_0)x$. We precompute $b_1\zeta$ and store it in a table, denoted as b_1'. Since b_1' is a known value, we only need to compute $(a_0b_0 + a_1b_1') + (a_0b_1 + a_1b_0)x$. Leveraging the properties of Barrett and Montgomery multiplications described above, we can perform the modular multiplication using $(\mathsf{Mont}(a_0b_0) + \mathsf{Bar}(a_1b_1')) + (\mathsf{Mont}(a_0b_1) + \mathsf{Mont}(a_1b_0))x$. We further optimize the data movement by utilizing the `ld2h` instruction, which automatically de-interleaves the even and odd coefficients into separate registers during the load stage, thereby avoiding expensive shuffle operations.

3.3 Vectorized NTT

During the vectorized forward NTT, all $n = 256$ coefficients are sequentially loaded into SVE2 vector registers using the `ld1h` instruction. Eight registers (`z0`-`z7`), each holding 512 bits, are used to accommodate the entire input. All seven NTT layers are then computed in-place before writing the final results back to memory. This approach maximizes register utilization and minimizes memory access overhead. The full computation process is illustrated in Figure 3. The first three layers correspond to CT butterfly operations, where coefficient pairs reside at the same relative positions in two separate registers. These layers operate at strides of 128, 64, and 32, respectively. For example, the first layer processes pairs from `z0` and `z4`, the second from `z0` and `z2`, and the third from `z0` and `z1`. The assembly-level sequences for both CT and GS butterfly operations are shown in Algorithm 10 and Algorithm 11. In the remaining four layers, butterfly operations must be performed on elements located within the same register. To maintain parallelism and avoid underutilizing register lanes, we interleave data across registers using `zip` and `trn` instructions. This transformation enables

cross-register computation, effectively converting intra-register operations into inter-register ones. The resulting strides align with the final four layers of the CT butterfly. After completing all seven layers, coefficients are reordered back to their original layout to preserve memory alignment and simplify storage.

Algorithm 10 CT Butterfly Operation of Two SVE2 Register Elements

Input: Coefficient register `a_low`, `a_high` with each element in $\mathbb{Z}_q$, twiddle factor register `zeta`, pre-computed factor `zetaR`, parameters register `q`, temporary register `c`, `t`.

Output: CT butterfly result register `a_low`, `a_high`.

```
1: barrett_mul a_high, zeta, zetaR, c, t, q
2: sub  a_high.h, a_low.h, c.h
3: add  a_low.h, a_low.h, c.h
```

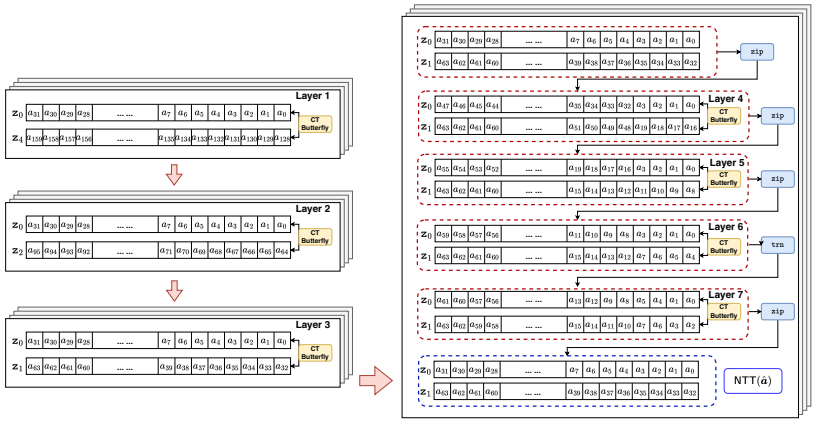

Fig. 3. Implementation of Vectorized Forward NTT (VecNTT).

Differences in INVNTT. In the vectorized inverse NTT, the first layer uses a GS butterfly with a coefficient stride of 2. Coefficients are sequentially loaded into registers using the `ld1h` instruction. Since the coefficient pairs for the GS butterfly operation reside within the same register, we first apply data shuffling to rearrange the coefficient order, which is the inverse of the forward NTT. Specifically, we use `uzp` and `trn` instructions to perform the shuffle, followed by the GS butterfly transformation as shown in Algorithm 11. For the remaining three layers, the GS butterfly coefficient pairs are located at corresponding positions across different registers, allowing direct inter-register GS butterfly operations without additional shuffling.

Algorithm 11 GS Butterfly Operation of Two SVE2 Register Elements

Input: Coefficient register `a_low`, `a_high` with each element in $\mathbb{Z}_q$, twiddle factor register `zeta`, pre-computed factor `zetaR`, parameters register `q`, temporary register `c`, `t`.

Output: GS butterfly result register `a_low`, `a_high`.

```
1: sub  t.h, a_low.h, a_high.h
2: add  a_low.h, a_low.h, a_high.h
3: barrett_mul t, zeta, zetaR, a_high, c, q
```

Pipeline Scheduling for VecNTT. To hide instruction latencies and exploit the out-of-order engine, we adopt a round-robin scheduling strategy across register pairs. While the butterfly unit is busy with one pair (e.g., `z0`/`z4`), the scheduling logic issues independent instructions for another pair (e.g., `z1`/`z5`). In the shuffle-heavy layers, we interleave the data rearrangement instructions (`zip`, `trn`) with modular multiplications to maximize functional unit utilization, as shown in Figure 4.

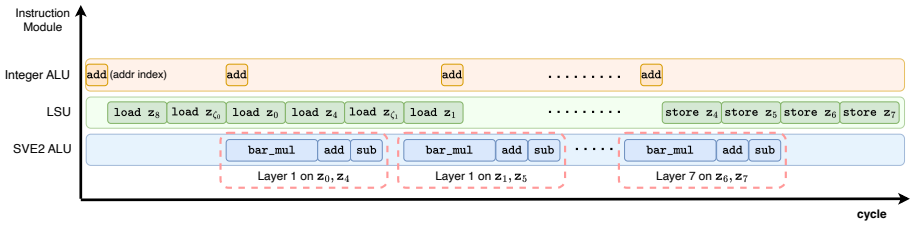

Fig. 4. Pipeline Design of Vectorized NTT. Here, `bar_mul` denotes Barrett multiplication, and $\mathtt{z}_{\zeta_i}$ represents the twiddle factor associated with the coefficient at the current layer.

3.4 Matrix-based NTT

For the first three layers, as in the VecNTT approach, polynomial coefficients are sequentially loaded into 8 SVE2 registers, with CT butterfly operations performed between register pairs. The updated coefficients in registers `z0-z7` are then written into the ZA storage in row-major order. Using SME instructions, we perform a vertical load from ZA back into the SVE2 registers, which effectively transposes the matrix without the overhead of shuffling operations. However, when loading vertically, the unequal width and height of the matrix result in only one-quarter of the SVE2 register space being utilized, leading to resource underutilization. To address this issue, we propose a strategy that combines the coefficient strides required for subsequent butterfly layers. We perform an inter-register shuffle to fill registers to half capacity, enabling inter-register butterfly

operations. After completing the fourth layer CT butterfly transformation, we apply coefficient shuffling to fully populate the registers, eliminating resource waste. The final three layers can be computed directly via inter-register butterfly operations without additional shuffling.

This approach leverages the horizontal and vertical access patterns of matrix operations, effectively combining inter-register and intra-register butterfly transformations to maximize instruction set architecture resource utilization. Furthermore, unlike VecNTT, which performs intra-register shuffling that cannot achieve sequential ordering due to instruction set limitations and thus complicates twiddle factor arrangement, MatNTT maintains sequentially ordered coefficients within each register, providing a more favorable arrangement for twiddle factors. Moreover, the twiddle factors can be reused across multiple register pairs within the same layer, significantly reducing both the size of the twiddle factor table and the overhead associated with loading these factors. This overall process is illustrated in Figure 5.

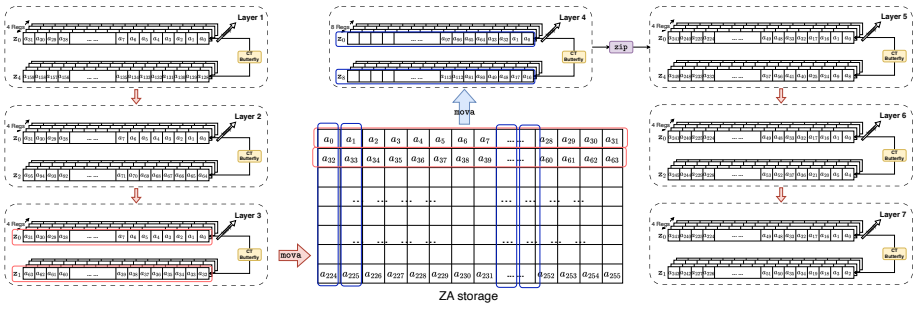

Fig. 5. Implementation of Matrix-based Forward NTT (MatNTT) leveraging SME ZA storage.

Pipeline Scheduling for MatNTT. The MatNTT scheduling focuses on overlapping ZA storage accesses with computation. Since writing to and reading from ZA storage are adjacent to the fourth butterfly layer, we interleave the SME memory instructions with the CT butterfly calculations. This effectively hides the latency of matrix transposition and ensures that the SME and SVE2 units operate in parallel, significantly increasing overall throughput.

4 Evaluation

4.1 Benchmarking Setup

All benchmarks were performed on a MacBook Pro with an Apple M4 Pro SoC. Code was compiled using `Apple clang` 16.0 with `-march=armv9.2-a+sme` and `-O3` to enable SVE2/SME instructions and high-level optimizations. All SVE2

and SME assembly codes were executed in streaming SVE mode, managed via `smstart` and `smstop`. We use the implementation from liboqs [23] as the baseline, comparing it with our optimized version. All reported cycle counts represent the average of 10,000 runs of the algorithm, measured in clock cycles.

4.2 Polynomial Arithmetic

We applied the proposed optimization techniques to both polynomial and polynomial vector operations. The cycle counts before and after optimization were measured to evaluate the performance. The results are summarized in Table 1.

Table 1. Performance comparison between the baseline and our optimized polynomial arithmetic kernels.

Algorithm	NTT	INVNTT	MatrixVectorMul			InnerProd (Enc)			InnerProd (Dec)		
Security Level			I	III	V	I	III	V	I	III	V
Ref	3248	5102	27223	46818	70504	9606	11486	13517	15955	21229	26332
VecNTT	695	710	5708	9537	13359	1703	2093	2327	4020	5555	6388
	(4.67×)	(7.18×)	(4.76×)	(4.90×)	(5.27×)	(5.64×)	(5.48×)	(5.81×)	(3.97×)	(3.82×)	(4.12×)
MatNTT	727	657	5345	9541	14263	1620	2050	2365	3743	5612	7327
	(4.47×)	(7.77×)	(5.09×)	(4.91×)	(4.94×)	(5.93×)	(5.60×)	(5.71×)	(4.26×)	(3.78×)	(3.59×)

NTT Performance Analysis. Compared to the reference implementation, our VecNTT achieves a 4.67× speedup for the forward NTT, while MatNTT achieves 4.47×. For the inverse NTT (INVNTT), VecNTT and MatNTT deliver 7.18× and 7.77× speedups, respectively. While both methods exhibit comparable computational throughput, MatNTT offers a superior memory footprint. Specifically, the twiddle factor table for MatNTT is approximately 65% the size of that required for VecNTT, owing to the reuse of factors across different register pairs within the matrix structure. Furthermore, the sequential data access pattern in MatNTT simplifies twiddle factor arrangement and minimizes cache misses. Consequently, we select MatNTT as the default NTT engine for our full ML-KEM implementation.

Polynomial Vector Operations. The most performance-critical operations in ML-KEM involve polynomial vectors, specifically the *Matrix-Vector Multiplication* and *Inner Product*. By optimizing the underlying modular arithmetic and NTT kernels, we achieve substantial speedups in these composite operations. As shown in Table 1, MatrixVectorMul is accelerated by up to 5.27× across all security levels. Similarly, the Inner Product operations used in encryption and decryption see speedups ranging from 3.59× to 5.93×. These results demonstrate that the effectiveness of our optimized NTT and asymmetric multiplication.

4.3 Performance of Scheme

The proposed polynomial optimizations are integrated into the ML-KEM algorithm, with performance benchmarking conducted using the MatNTT method on the Apple M4 Pro. Table 2 compares the results with the reference implementation [19], highlighting significant performance improvements. Notably, the key encapsulation, critical to the algorithm, shows substantial speedup: ML-KEM-512 achieves a 52.47% speedup, ML-KEM-768 improves by 58.61%, and ML-KEM-1024 delivers a 60.09% speedup. These results demonstrate the effectiveness of our SVE2/SME-optimized implementation and pipeline design for ML-KEM applications.

Table 2. Overall performance comparison of ML-KEM on the Apple M4 Pro.

Parameter		KeyGen	Encaps	Decaps
ML-KEM-512	Ref	56218	65903	85270
	Ours	28511	31322	38711
		(49.28%)	(52.47%)	(54.60%)
ML-KEM-768	Ref	95497	103983	130925
	Ours	41398	43042	56561
		(56.65%)	(58.61%)	(56.80%)
ML-KEM-1024	Ref	147435	151319	186344
	Ours	58644	60385	80823
		(60.22%)	(60.09%)	(56.63%)

5 Conclusion

In this work, we presented the first highly optimized implementation of ML-KEM on the ARMv9-A architecture, leveraging the advanced SVE2 and SME extensions. By redesigning the polynomial computation pipeline to align with out-of-order execution and utilizing the ZA storage of SME for efficient matrix-oriented operations, we achieved significant performance gains. Our evaluation on the Apple M4 Pro processor demonstrates speedups of up to 7.77× for core NTT operations and an overall reduction in encapsulation of up to 60.09% compared to the baseline implementations.

These findings establish a new performance baseline for post-quantum cryptography on high-performance ARM platforms. Future work will explore extending these optimization strategies to other cryptographic primitives, such as the digital signature scheme FN-DSA, and investigating further throughput enhancements through deeper SME integration. Our results confirm that the innovative repurposing of AI-centric hardware features like SVE2 and SME is vital for achieving the high-performance post-quantum security required for next-generation computing environments.

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). https://doi.org/10.1007/978-3-031-09234-3_42, https://doi.org/10.1007/978-3-031-09234-3_42
2. Apple: Apple introduces m4 chip. https://www.apple.com/newsroom/2024/05/apple-introduces-m4-chip/ (2024)
3. Apple: Apple silicon cpu optimization guide. https://developer.apple.com/documentation/apple-silicon/cpu-optimization-guide (2024)
4. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D., et al.: Crystals-kyber algorithm specifications and supporting documentation. NIST PQC Round **2**(4), 1–43 (2019)
5. Becker, H., Hwang, V., Kannwischer, M.J., Yang, B., Yang, S.: Neon NTT: faster dilithium, kyber, and saber on cortex-a72 and apple M1. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(1), 221–244 (2022). https://doi.org/10.46586/TCHES.V2022.I1.221-244
6. Bevin, R., Khalid, A., Imran, M., O'Neill, M.: Accelerating crystals-kyber and dilithium via a single montgomery reduction ISE on RISC-V. In: Zhao, D., Hofmann, K. (eds.) 38th IEEE International System-on-Chip Conference, SOCC 2025, Dubai, United Arab Emirates, September 29 - Oct. 1, 2025. pp. 1–6. IEEE (2025). https://doi.org/10.1109/SOCC66126.2025.11235487, https://doi.org/10.1109/SOCC66126.2025.11235487
7. Botros, L., Kannwischer, M.J., Schwabe, P.: Memory-efficient high-speed implementation of kyber on cortex-m4. In: Buchmann, J., Nitaj, A., Rachidi, T. (eds.) Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11627, pp. 209–228. Springer (2019). https://doi.org/10.1007/978-3-030-23696-0_11, https://doi.org/10.1007/978-3-030-23696-0_11
8. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Mathematics of computation **19**(90), 297–301 (1965)
9. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z., et al.: Falcon: Fast-fourier lattice-based compact signatures over ntru. Submission to the NIST's post-quantum cryptography standardization process **36**(5), 1–75 (2018)
10. 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). https://doi.org/10.1007/3-540-48405-1_34, https://doi.org/10.1007/3-540-48405-1_34
11. Gaborit, P., Aguilar-Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Persichetti, E., Zémor, G., Bos, J., et al.: Hamming quasi-cyclic (hqc). NIST Post-Quantum Standardization, 4th Round; National Institute of Standards and Technology: Gaithersburg, MD, USA (2025)
12. Gentleman, W.M., Sande, G.: Fast fourier transforms: For fun and profit. In: Proceedings of the November 7-10, 1966, Fall Joint Computer Conference. p.

563âĂŞ578. AFIPS '66 (Fall), Association for Computing Machinery, New York, NY, USA (1966). https://doi.org/10.1145/1464291.1464352
13. Gupta, N., Jati, A., Chauhan, A.K., Chattopadhyay, A.: PQC acceleration using gpus: Frodokem, newhope, and kyber. IEEE Trans. Parallel Distributed Syst. **32**(3), 575–586 (2021). https://doi.org/10.1109/TPDS.2020.3025691, https://doi.org/10.1109/TPDS.2020.3025691
14. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the fujisaki-okamoto transformation. In: Kalai, Y., Reyzin, L. (eds.) Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10677, pp. 341–371. Springer (2017). https://doi.org/10.1007/978-3-319-70500-2_12, https://doi.org/10.1007/978-3-319-70500-2_12
15. Ji, X., Dong, J., Deng, T., Zhang, P., Hua, J., Xiao, F.: Hi-kyber: A novel high-performance implementation scheme of kyber based on GPU. IEEE Trans. Parallel Distributed Syst. **35**(6), 722–736 (2024). https://doi.org/10.1109/TPDS.2024.3379734, https://doi.org/10.1109/TPDS.2024.3379734
16. Li, H., Mentens, N., Picek, S.: A scalable SIMD RISC-V based processor with customized vector extensions for crystals-kyber. In: Oshana, R. (ed.) DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022. pp. 733–738. ACM (2022). https://doi.org/10.1145/3489517.3530552, https://doi.org/10.1145/3489517.3530552
17. Mandal, S., Roy, D.B.: Kid: A hardware design framework targeting unified NTT multiplication for crystals-kyber and crystals-dilithium on FPGA. In: 37th International Conference on VLSI Design and 23rd International Conference on Embedded Systems, VLSID 2024, Kolkata, India, January 6-10, 2024. pp. 455–460. IEEE (2024). https://doi.org/10.1109/VLSID60093.2024.00082
18. MediaTek: vivo v27 powered by mediatek dimensity 7200. https://www.mediatek.com/tek-talk-blogs/vivo-v27-powered-by-mediatek-dimensity-7200 (March 28, 2023)
19. NIST: Fips 203: Module-lattice-based key-encapsulation mechanism standard. https://csrc.nist.gov/pubs/fips/203/final (2024), https://csrc.nist.gov/pubs/fips/203/final
20. NIST: Fips 204: Module-lattice-based digital signature standard. https://csrc.nist.gov/pubs/fips/204/final (2024), https://csrc.nist.gov/pubs/fips/204/final
21. NIST: Fips 205: Stateless hash-based digital signature standard. https://csrc.nist.gov/pubs/fips/205/final (2024), https://csrc.nist.gov/pubs/fips/205/final
22. NIST: Post-quantum cryptography, pqc standardization process. https://csrc.nist.gov/pqc-standardization (2025), https://csrc.nist.gov/pqc-standardization
23. Safe, O.Q.: liboqs: C library for prototyping and experimenting with quantum-resistant cryptography. https://github.com/open-quantum-safe/liboqs (2025), version 0.15.0
24. SAMSUNG: Exynos 1580. https://semiconductor.samsung.cn/processor/mobile-processor/exynos-1580/ (2024)
25. Sanal, P., Karagoz, E., Seo, H., Azarderakhsh, R., Kermani, M.M.: Kyber on ARM64: compact implementations of kyber on 64-bit ARM cortex-a processors. In: García-Alfaro, J., Li, S., Poovendran, R., Debar, H., Yung, M. (eds.) Security and Privacy in Communication Networks - 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6-9, 2021, Proceedings,

Part II. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 399, pp. 424–440. Springer (2021). https://doi.org/10.1007/978-3-030-90022-9_23, https://doi.org/10.1007/978-3-030-90022-9_23

26. Seiler, G.: Faster AVX2 optimized NTT multiplication for ring-lwe lattice cryptography. IACR Cryptol. ePrint Arch. p. 39 (2018), http://eprint.iacr.org/2018/039
27. Wang, T., Zhang, C., Zhang, X., Gu, D., Cao, P.: Optimized hardware-software co-design for kyber and dilithium on RISC-V soc FPGA. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2024**(3), 99–135 (2024). https://doi.org/10.46586/TCHES.V2024.I3.99-135
28. Wei, H., Li, W., Shen, S., Yang, H., Guo, W., Zhao, Y.: Vectorized SVE2 optimization of the post-quantum signature ML-DSA on armv9-a architecture. IACR Cryptol. ePrint Arch. p. 2208 (2025), https://eprint.iacr.org/2025/2208
29. Wu, Z., Chen, R., Wang, Y., Wang, Q., Peng, W.: An efficient hardware implementation of crystal-dilithium on FPGA. In: Zhu, T., Li, Y. (eds.) Information Security and Privacy - 29th Australasian Conference, ACISP 2024, Sydney, NSW, Australia, July 15-17, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14896, pp. 64–83. Springer (2024). https://doi.org/10.1007/978-981-97-5028-3_4
30. Yu, X., Zhang, J., Huang, J., Chen, D., Zhou, L.: Multi-way high-throughput implementation of kyber. In: Mouha, N., Nikiforakis, N. (eds.) Information Security - 27th International Conference, ISC 2024, Arlington, VA, USA, October 23-25, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 15258, pp. 41–60. Springer (2024). https://doi.org/10.1007/978-3-031-75764-8_3, https://doi.org/10.1007/978-3-031-75764-8_3
31. Zhou, T., Zheng, F., Fan, G., Wan, L., Tang, W., Song, Y., Bian, Y., Lin, J.: Convkyber: Unleashing the power of AI accelerators for faster kyber with novel iteration-based approaches. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2024**(2), 25–63 (2024). https://doi.org/10.46586/TCHES.V2024.I2.25-63, https://doi.org/10.46586/tches.v2024.i2.25-63