# Parameter-Aware and Instruction-Driven Dilithium Optimization on AVX2 and NEON

Shi Ya[1,2], Liu Bingqian[1,2], Lu Xianhui[1,2], Qian Wenfei[1,2], Liu Ying[1,2] and Wang Kungpeng[1,2]

[1] State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China,
{shiya,liubingqian,luxianhui,qianwenfei,liuying,wangkunpeng}@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

**Abstract.** We improve the performance of the lattice-based cryptosystem Dilithium on AVX2 and NEON by deeply exploiting its algorithmic properties, such as small coefficient bounds and high sparsity, with the distinct instruction-level profiles of the underlying architectures. On AVX2, we deploy a single-modulus 16-bit NTT for $c \cdot \mathbf{s}_i$ and a multi-moduli 16-bit NTT coupled with a vectorized CRT reconstruction for $c \cdot \mathbf{t}_0$. These instruction-level optimizations accelerate the respective computations by 2.4–2.5× and 1.2–1.3× over official AVX2 baselines, ultimately reducing the overall Dilithium signature generation time by 7% to 8%. Conversely, On ARMv8-A NEON, we retain the efficient 16-bit NTT for $c \cdot \mathbf{s}_i$, while proposing a Fast Sparse Polynomial Multiplication (Fast-SPM) method for $c \cdot \mathbf{t}_0$. By exploiting the extreme sparsity of the challenge polynomial, Fast-SPM entirely bypasses the NTT and converts the computation into highly efficient index-shifted additions. Across the tested NEON platforms (Cortex-A72 and Apple M1 Pro), this hybrid approach achieves a 1.9–2.1× speedup for $c \cdot \mathbf{s}_i$ alongside a 1.1–1.9× acceleration for $c \cdot \mathbf{t}_0$, which translates into a 10% to 13% reduction in the overall signature generation time.

**Keywords:** Lattice-based cryptography · Dilithium · Number Theoretic Transform · SIMD · AVX2 · NEON

## 1 Introduction

To counter the threat of quantum computers, the NIST [NIS] Post-Quantum Cryptography (PQC) standardization process has culminated in the selection of Dilithium [ABD+20] as the FIPS 204 digital signature standard. The computational bottleneck in Dilithium predominantly stems from the underlying polynomial multiplication.

To accelerate this critical arithmetic, prior optimization research has broadly evolved along three distinct routes. The first route focuses on ISA-optimized generic NTT implementations [Sei18, BHK+22, HZZ+22, XHL+24]. These works deeply couple the NTT algebraic structure with the specific instruction sets of target platforms to maximize throughput. However, this approach is parameter-independent: it processes all polynomial operations using general 32-bit arithmetic, without exploiting the unique algebraic properties and restricted coefficient bounds of specific Dilithium operands. The second route explores parameter-aware algebraic adaptations. Recognizing that the highly sparse challenge polynomial $c$ and strictly bounded secret vectors are significantly smaller than the primary modulus, researchers map these small-coefficient products to alternative domains. Methods in this category include Fermat-number algorithms [AHKS22] and Parallel Small Polynomial Multiplication (PSPM) [ZHS+22, ZZS+25], which leverage sparsity directly in the coefficient domain. The third route emphasizes instruction-driven optimizations.

Although initially pioneered for other lattice-based schemes such as NTRU [CHK+21] and Saber [ACC+22], this approach directly leverages the superior throughput of 16-bit instructions on platforms like x86 AVX2 or Armv7 to bypass the inherent bottlenecks of 32-bit arithmetic.

While these tracks provide essential foundational techniques, achieving optimal computational efficiency requires a holistic unification of parameter traits and instruction-level characteristics. Although this algorithmic-architectural co-design philosophy has been successfully realized on resource-constrained platforms (e.g., mapping Dilithium's small bounds to Cortex-M sub-word DSP instructions [HKK+25, HAZ+24]), optimizations on high-performance SIMD platforms have not yet simultaneously considered both parameter properties and Instruction Set Architecture (ISA) disparities. The fundamental hurdle lies in the rigid parallel data layout constraints of wide vector registers, coupled with the contradictory instruction latencies across different SIMD platforms. For instance, on the x86 AVX2 platform, constant-time 32-bit vector multiplication incurs a severe throughput penalty, whereas dedicated 16-bit instructions offer superior efficiency. While multi-modular decomposition elegantly exploits this 16-bit efficiency on scalar devices, bridging 16-bit intermediate NTTs with Dilithium's mandatory 32-bit CRT reconstructions across 256-bit vector lanes introduces severe parallel data layout conflicts. Conversely, the ARMv8-A NEON instruction set processes 16-bit and 32-bit multiplications with identical latencies, rendering the multi-moduli approach architecturally suboptimal due to redundant transform overhead. Meanwhile, existing SIMD parameter-aware attempts (e.g., PSPM) struggle to map sparsity efficiently to wide vectors, suffering from low register utilization and complex preprocessing. Consequently, a platform-optimal design that comprehensively unifies Dilithium's parameter traits with SIMD instruction disparities remains a critical unresolved challenge.

**Our Contributions.** Driven by these profound instruction-level disparities, we present a parameter-aware and instruction-driven optimization framework for Dilithium on the x86 (AVX2) and ARMv8-A (NEON) platforms. By jointly considering Dilithium's parameter characteristics (bounded values and sparsity) and the specific bottlenecks of each ISA, we formulate divergent, platform-optimal algebraic strategies.

Specifically, inspired by the multi-modular approaches applied to NTRU [CHK+21] and resource-constrained devices [HKK+25, HAZ+24], we adapt this methodology for Dilithium on the x86 AVX2 platform. To overcome the inherent 16-bit to 32-bit transition bottleneck during the final CRT reconstruction, we carefully design the parallel data layout across 256-bit lanes to resolve cross-precision reconstruction conflicts. Concurrently, we conduct precise algebraic bound analyses to guarantee that no register overflow occurs during these critical precision transitions. We deploy a single-modulus 16-bit NTT for $c \cdot \mathbf{s}_i$ and a multi-moduli 16-bit NTT coupled with a customized, vectorized CRT reconstruction for $c \cdot \mathbf{t}_0$. Compared to the official 32-bit AVX2 implementation [ABD+20], our single-modulus 16-bit NTT accelerates the computation of $c \cdot \mathbf{s}_i$ by 2.4–2.5×, while the multi-moduli 16-bit NTT coupled with a customized, vectorized CRT reconstruction accelerates $c \cdot \mathbf{t}_0$ by 1.2–1.3×.

Conversely, observing that multi-moduli techniques offer no architectural advantage on the ARMv8-A NEON platform, we adopt a hybrid optimization strategy inspired by prior Parallel Small Polynomial Multiplication (PSPM) schemes [ZHS+22]. To overcome the heavy multidimensional preprocessing, low register utilization, and data-dependent leakage risks inherent in existing PSPM designs, we propose the Fast Sparse Polynomial Multiplication (Fast-SPM) method. Fast-SPM bypasses the NTT entirely, directly converting the sparse computation into highly efficient, constant-time index-shifted additions. By deploying a highly efficient single-modulus 16-bit NTT for $c \cdot \mathbf{s}_i$ and utilizing the customized Fast-SPM method for $c \cdot \mathbf{t}_0$, we achieve architecture-optimal performance. Compared to state-of-the-art baselines [BHK+22], our single-modulus 16-bit NTT accelerates the

computation of $c \cdot \mathbf{s}_i$ by 1.9–2.1×, while the Fast-SPM method accelerates $c \cdot \mathbf{t}_0$ by 1.1–1.9×.

Ultimately, by integrating these platform-specific polynomial multiplication strategies into the complete Dilithium scheme, we effectively alleviate its predominant computational bottleneck. This parameter-aware and ISA-optimized co-design translates directly into substantial systemic performance gains. Our targeted strategies reduce the overall generation time of Dilithium signatures by 7% to 8% on AVX2 platforms and by 10% to 13% across the tested NEON microarchitectures (Cortex-A72 and Apple M1 Pro).

**Related Work.** Beyond specific optimizations for Dilithium, the broader software optimization of polynomial multiplication in lattice-based cryptography continues to be extensively studied. This includes exploring the vectorization of generic algebraic structures on high-performance platforms [BBCT22, LHP+23, NG21, HLY24, Hwa24] and balancing performance with strict memory footprints on embedded devices [GKS21, BMK+22].

**Structure of the Paper.** Section 2 provides the necessary background. Sections 3 and 4 describe the proposed optimization techniques for x86 and ARMv8-A, respectively. Section 5 presents the performance evaluation.

# 2 Preliminaries

## 2.1 Residue Number System and The Chinese Remainder Theorem

The Residue Number System (RNS) is a method of representing large integers by their residues with respect to a set of pairwise coprime moduli. Given coprime integers $p_1, p_2, \ldots, p_s$, an integer $x$ is uniquely represented by its residues $(x \bmod p_1, \ldots, x \bmod p_s)$. In lattice-based cryptography, a large modulus $q$ is decomposed using RNS to execute polynomial multiplication efficiently in smaller domains:

$$\mathbb{Z}_q[x]/(x^n+1) \longrightarrow \prod_{i=1}^{s} \mathbb{Z}_{q_i}[x]/(x^n+1).$$

The Chinese Remainder Theorem (CRT) provides the mechanism to reconstruct these parallel results back into modulo $q$. Researchers typically employ either the Explicit CRT [BS07] or the divided-difference interpolation CRT algorithm [CHK+21], which offers superior performance when the number of moduli $s$ is small.

**Theorem 1.** *Let $p_i > 0$ be odd, pairwise co-prime ($\gcd(p_i, p_j) = 1$ for $1 \le i < j \le s$). An explicit solution $u$ of $u \equiv u_i \pmod{p_i}$, $i = 1...s$, where $|u_i| < p_i/2$, where $|u| < P/2 = \prod_{i=1}^{s} p_i$, is given by*

$$\begin{cases} y_1 = u_1 \\ y_2 = y_1 + \big((u_2 - y_1)\, m_2 \pmod{}^{\pm} p_2\big)\, p_1 \\ y_3 = y_2 + \big((u_3 - y_2)\, m_3 \pmod{}^{\pm} p_3\big)\, p_1 p_2 \\ \quad\vdots \\ u = y_s = y_{s-1} + \big((u_s - y_{s-1})\, m_s \pmod{}^{\pm} p_s\big)\, p_1 \cdots p_{s-1} \end{cases}$$

*where each $m_i := (p_1 \cdots p_{i-1})^{-1} \pmod{}^{\pm} p_i$.*

## 2.2 Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is a critical algorithmic component in lattice-based cryptography, reducing the overall computational complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$. It achieves this efficiency by mapping the intensive coefficient-domain convolution into a point-value domain, where the computation simplifies to an

element-wise product. For the standard NTT to be well-defined over the negacyclic polynomial ring $\mathbb{Z}_q[x]/\langle x^n+1\rangle$, the modulus $q$ must be a prime satisfying $q \equiv 1 \pmod{2n}$. This strict algebraic condition guarantees the existence of a primitive $2n$-th root of unity $\zeta$, which is fundamentally required to support the negacyclic convolution property of the ring.

**Butterfly Operations and Modular Reduction.** The NTT is typically computed using the Cooley-Tukey (CT) butterfly algorithm, which recursively divides the polynomial evaluation into smaller sub-problems. The fundamental CT butterfly operation computes $U' = U + \zeta V \pmod{q}$ and $V' = U - \zeta V \pmod{q}$. Conversely, the Inverse NTT (INTT) frequently employs the Gentleman-Sande (GS) butterfly, which computes $U' = U + V \pmod{q}$ and $V' = (U - V)\zeta^{-1} \pmod{q}$.

Within these butterfly networks, modular reduction constitutes the most expensive arithmetic step. Hardware implementations generally rely on two division-free algorithms. *Montgomery reduction* avoids division by $q$ by operating in the Montgomery domain, effectively replacing the modulo $q$ with operations modulo $R$ and division by $R$ (typically a power of 2). On SIMD architectures, this algorithm heavily relies on efficiently extracting both the low and high halves of a multiplication result. Alternatively, *Barrett reduction* precomputes an approximation of $1/q$ to estimate the quotient, followed by a subtraction to find the remainder. It strictly requires a multiply-high instruction with specific rounding behaviors to execute efficiently.

**Incomplete NTT.** When the chosen modulus $q$ does not satisfy the strict $q \equiv 1 \pmod{2n}$ condition but its maximum power-of-two divisibility satisfies $q \equiv 1 \pmod{2^{k+1}}$ where $k < \log_2(n)$, a full $n$-point transform over the negacyclic ring is mathematically impossible. In such scenarios, an *incomplete NTT* is utilized. The transform halts after exactly $k$ layers, mapping the original polynomial ring into a direct product of $2^k$ smaller rings: $\mathbb{Z}_q[x]/\langle x^n+1\rangle \cong \prod_{i=0}^{2^k-1} \mathbb{Z}_q[x]/\langle x^{n/2^k} - \gamma_i\rangle$. While this reduces the total number of butterfly operations, the pointwise multiplications in the transformed domain become polynomial multiplications of degree $n/2^k - 1$ rather than simple scalars.

**16-bit vs. 32-bit NTT.** The microarchitectural efficiency of the NTT strongly depends on the choice of modulus and the datapath width. Schemes like Kyber use 16-bit friendly moduli (e.g., 3329). On platforms like AVX2, constant-time 16-bit NTTs are highly efficient, processing sixteen 16-bit coefficients simultaneously within a 256-bit register. Conversely, Dilithium utilizes a 32-bit modulus $q = 8380417$. The 32-bit NTT is significantly slower on AVX2, not merely because the 256-bit register accommodates only eight 32-bit coefficients, but primarily because AVX2 lacks a high-throughput 32-bit parallel multiplier (e.g., the `vpmuldq` instruction processes only four elements per call and demands expensive shuffling). Reference [HAZ+24] show that Kyber's 16-bit NTT outperforms Dilithium's 32-bit NTT by a factor of approximately 4.11×.

**Multi-Moduli Reconstruction.** Given the inefficiency of 32-bit NTTs on certain microarchitectures, an effective algebraic reconstruction strategy is to decompose 32-bit data into multiple 16-bit parallel operations. By utilizing coprime 16-bit moduli (e.g., $q_1, q_2$), the transform can be computed using highly optimized 16-bit NTTs. The fragmented results are subsequently merged via the CRT. Chung et al. [CHK+21] successfully utilized multi-moduli NTT for Saber, NTRU, and LAC. Recent work [HAZ+24] demonstrated its feasibility for accelerating Dilithium on Cortex-M processors.

## 2.3 Dilithium Signature Algorithm

Dilithium is standardized by NIST and operates over the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^{256}+1)$ with $q = 8380417$. The parameters used in these algorithms are shown in Table 1.

**Table 1:** ML-DSA parameter sets

| **Parameters** | ML-DSA-44 | ML-DSA-65 | ML-DSA-87 |
|---|---|---|---|
| $q$ - modulus | 8380417 | 8380417 | 8380417 |
| $\zeta$ - a 512th root of unity in $\mathbb{Z}_q$ | 1753 | 1753 | 1753 |
| $d$ - # of dropped bits from $t$ | 13 | 13 | 13 |
| $\tau$ - # of $\pm 1$'s in polynomial $c$ | 39 | 49 | 60 |
| $\lambda$ - collision strength of $\tilde{c}$ | 128 | 192 | 256 |
| $\gamma_1$ - coefficient range of $\mathbf{y}$ | $2^{17}$ | $2^{19}$ | $2^{19}$ |
| $\gamma_2$ - low-order rounding range | $(q-1)/88$ | $(q-1)/32$ | $(q-1)/32$ |
| $(k, \ell)$ - dimensions of $\mathbf{A}$ | (4,4) | (6,5) | (8,7) |
| $\eta$ - private key range | 2 | 4 | 2 |
| $\beta = \tau \cdot \eta$ | 78 | 196 | 120 |
| $\omega$ - max # of 1's in the hint $\mathbf{h}$ | 80 | 55 | 75 |

The Dilithium protocol, as outlined in Algorithms 1, 2, and 3, comprises key generation, signature generation, and verification. The scheme extensively utilizes symmetric primitives based on SHAKE to expand the public matrix $\hat{\mathbf{A}}$ and sample various vectors via functions like ExpandA and ExpandS. Additionally, it employs specific operations such as Power2Round, UseHint and MakeHint to compress the public key by discarding the lower bits of polynomial coefficients. During the signing phase, the matrix-vector multiplication $\mathbf{Ay}$ undeniably dominates the overall execution time and is typically accelerated using the NTT. However, the subsequent polynomial convolutions between the sparse challenge polynomial $c$ and the vectors $\mathbf{s}_1, \mathbf{s}_2$, and $\mathbf{t}_0$ also introduce substantial computational overhead. Specifically, the secret vectors $\mathbf{s}_1$ and $\mathbf{s}_2$ are strictly bounded by $\eta$, the challenge $c$ contains exactly $\tau$ non-zero coefficients restricted to $\pm 1$, and the coefficients of $\mathbf{t}_0$ are bounded by $2^{12}$. These specific algebraic boundaries explicitly define the design space for bypassing costly multi-precision modular arithmetic and unnecessary NTT conversions in subsequent optimizations.

**Algorithm 1** Dilithium Key Generation (KeyGen)

**Output:** Public key $pk$, Secret key $sk$

1: $\zeta \leftarrow \{0,1\}^{256}$
2: $(\rho, \rho', K) \leftarrow H(\zeta)$
3: $\hat{\mathbf{A}} \in R_q^{k \times \ell} \leftarrow \text{ExpandA}(\rho)$
4: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \text{ExpandS}(\rho')$
5: $\hat{\mathbf{s}}_1 \leftarrow \text{NTT}(\mathbf{s}_1)$
6: $\mathbf{t} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}\hat{\mathbf{s}}_1) + \mathbf{s}_2$
7: $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \text{Power2Round}_q(\mathbf{t}, d)$
8: $tr \leftarrow H(\rho \| \mathbf{t}_1)$
9: **return** $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

## 2.4 Algebraic Foundations and Safety Bounds

Before implementing microarchitecture-specific optimizations, it is strictly necessary to mathematically formalize the algorithmic transformations. Transforming standard 32-bit

**Algorithm 2** Dilithium Signing (Sign)

**Input:** Secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$, Message $M$
**Output:** Signature $\sigma = (\mathbf{z}, \mathbf{h}, \tilde{c})$

$\hat{\mathbf{A}} \in R_q^{k \times \ell} \leftarrow \text{ExpandA}(\rho)$; $\mu \leftarrow H(tr \| M)$; $\rho' \leftarrow H(K \| \mu)$
$\hat{\mathbf{s}}_1 \leftarrow \text{NTT}(\mathbf{s}_1); \hat{\mathbf{s}}_2 \leftarrow \text{NTT}(\mathbf{s}_2); \hat{\mathbf{t}}_0 \leftarrow \text{NTT}(\mathbf{t}_0)$
**repeat**
    $\mathbf{y} \leftarrow \text{ExpandMask}(\rho', \kappa)$
    $\hat{\mathbf{y}} \leftarrow \text{NTT}(\mathbf{y})$; $\mathbf{w} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}\hat{\mathbf{y}})$
    $(\mathbf{w}_1, \mathbf{w}_0) \leftarrow \text{HighBits}(\mathbf{w})$; $\tilde{c} \leftarrow H(\mu \| \mathbf{w}_1)$; $c \leftarrow \text{SampleBall}(\tilde{c})$
    $\hat{c} \leftarrow \text{NTT}(c)$; $\mathbf{z} \leftarrow \mathbf{y} + \text{NTT}^{-1}(\hat{c}\hat{\mathbf{s}}_1)$; $\mathbf{r} \leftarrow \mathbf{w} - \text{NTT}^{-1}(\hat{c}\hat{\mathbf{s}}_2)$
    $\mathbf{r}_0 \leftarrow \text{LowBits}(\mathbf{r})$
    **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **or** $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**
        reject and restart
    **else**
        $\mathbf{h} \leftarrow \text{NTT}^{-1}(\hat{c}\hat{\mathbf{t}}_0)$
        $\mathbf{h} \leftarrow \text{MakeHint}(-\mathbf{h}, \mathbf{r} + \mathbf{h})$
        **if** $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ or the number of 1's in $\mathbf{h}$ is greater than $\omega$ **then**
            reject and restart
        **else**
            **return** $\sigma = (\mathbf{z}, \mathbf{h}, \tilde{c})$
        **end if**
    **end if**
    $\kappa \leftarrow \kappa + \ell$
**until** signature accepted

**Algorithm 3** Dilithium Verification (Verify)

**Input:** $pk = (\rho, \mathbf{t}_1), M \in \{0,1\}^*$
**Input:** $\sigma = (\mathbf{z}, \mathbf{h}, \tilde{c})$
**Output:** Valid or Invalid

$\hat{\mathbf{A}} \in R_q^{k \times l} \leftarrow \text{ExpandA}(\rho)$
$\hat{\mathbf{z}} \leftarrow \text{NTT}(\mathbf{z})$
$\mu \in \{0,1\}^{384} \leftarrow \mathcal{H}(\mathcal{H}(\rho \parallel \mathbf{t}_1) \parallel M)$
$c \leftarrow \mathcal{H}_B(\tilde{c})$
$\mathbf{w}_1' \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}\hat{\mathbf{z}}) - 2^d c\mathbf{t}_1$
$\mathbf{w}_1' \leftarrow \text{UseHint}(\mathbf{h}, \mathbf{w}_1')$
**if** $\tilde{c} \neq \mathcal{H}(\mu \parallel \mathbf{w}_1')$ **or**
  $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **or**
  # 1's in $\mathbf{h} > \omega$ **then**
  **return** Invalid
**end if**
**return** Valid

arithmetic into 16-bit bounded operations introduces the risk of integer overflow during SIMD vector processing. In this section, we establish the rigorous dynamic range boundaries to prove the absolute safety of our multi-modulus reconstruction.

### 2.4.1 Bounding the Convolution $c \cdot \mathbf{t_0}$

To substitute the original 32-bit NTT with a 16-bit multi-modulus approach, the selected composite modulus $Q$ must completely encompass the maximum possible absolute value of the polynomial convolution.

According to the Dilithium specifications, the challenge polynomial $c$ contains exactly $\tau$ non-zero coefficients bounded in $\{-1, 1\}$. The vector $\mathbf{t}_0$ has coefficients strictly bounded within $[-2^{12}, 2^{12}]$. Thus, the maximum possible absolute value of a single coefficient in the resulting product polynomial is:

$$\gamma = \tau \times 2^{12} = 60 \times 4096 = 245760 \tag{1}$$

To safely compute this multiplication using the Residue Number System without losing precision due to modular wrap-around, the chosen composite modulus $Q$ must satisfy the inequality $Q > 2\gamma = 491520$.

By selecting two distinct prime moduli, $q_0 = 769$ and $q_1 = 3329$, we establish a composite modulus $Q = 769 \times 3329 = 2560001$. Since $2560001 > 491520$, the RNS representation holds absolute mathematical correctness.

### 2.4.2 Preventing Register Overflow During Incomplete NTT

A secondary, yet critical, microarchitectural constraint is that all intermediate coefficients generated during the multi-modulus NTT must strictly fit within the limits of a signed 16-bit integer (i.e., $[-32768, 32767]$) to utilize SIMD 16-bit datapaths without catastrophic overflow.

In our implementation, the Cooley-Tukey butterfly operations employ a signed Montgomery reduction. The mathematical property of this reduction guarantees that the coefficient growth per transformation layer is strictly bounded by the modulus $q_i$. Because $q_1 = 769$ and $q_2 = 3329$ do not support a full 8-layer decomposition, we employ an incomplete 7-layer NTT.

Over the course of 7 layers, the maximum cumulative coefficient growth evaluates to $7q_i$. Taking the larger modulus $q_2 = 3329$ as the worst-case scenario, the maximum intermediate coefficient magnitude is strictly bounded by:

$$\text{Max}_{bound} = 2^{12} + (7 \times 3329) = 4096 + 23303 = 27399 \tag{2}$$

Because $27399 < 32767$ (the maximum positive value of an `int16_t`), we theoretically guarantee that executing the incomplete NTT inside 16-bit vector registers is perfectly safe and immune to overflow vulnerabilities.

## 2.5 SIMD Instruction Architectures

Single Instruction Multiple Data (SIMD) architectures are critical for accelerating lattice-based cryptography. However, different extensions present unique constraints.

**AVX2** operates on 256-bit YMM registers. It provides dedicated instructions for 16-bit multiplication (`vpmullw`, `vpmulhw`) that elegantly extract the lower and upper halves of the product, crucial for Montgomery reductions. However, for 32-bit multiplications, it provides the `vpmuldq` instruction, which generates a 64-bit result, effectively halving register throughput and requiring subsequent shuffle instructions to repack the data. See Table 2 for relevant instructions.

**NEON** operates on 128-bit registers on ARM architectures. Crucially, NEON differs from AVX2 by processing 16-bit and 32-bit multiplications using identical fundamental instruction semantics. While a 128-bit register can hold eight 16-bit or four 32-bit integers, NEON does not penalize 32-bit multiplications with complex unpacking and repacking delays in the same manner as AVX2. See Table 3 for relevant instructions.

# 3 Dilithium Optimizations on x86 AVX2

This section details the customized optimization of the Dilithium signature algorithm using the AVX2 instruction set. We specifically analyze the instruction-level efficiency of

**Table 2:** Summary of instruction cycles on AVX2 with Skylake architecture.

| Instruction | Latency (cycles) | Throughput (cycles) |
|---|---|---|
| vpaddw/vpsubw (signed 16bit-integer) | 1 | 0.33 |
| vpmulhw/vpmullw (signed 16bit-integer) | 5 | 0.5 |
| vpmuldq (signed 32bit-integer) | 5 | 0.5 |
| vpmulld (signed 32bit-integer) | 10 | 1 |
| vpsllw (shift) | 1 | 0.5 |
| vpblendd | 1 | 0.33 |
| vmovdqa (load/store aligned) | 7/5 | 0.5/1 |

**Table 3:** Summary of instruction cycles on NEON with Cortex-A72.

| Instruction | Latency (cycles) | Throughput (cycles) | Utilized Pipelines |
|---|---|---|---|
| ADD/SUB (integer) | 3 | 2 | F0/F1 |
| MUL/SQRDMULH (Q-form) | 5 | 0.5 | F0 |
| MLA/MLS (Q-form) | 5 | 0.5 | F0 |
| SHL | 3 | 1 | F1 |
| LDR/LDP | 4 | 1 | L |
| STR/STP(w-form) | 1 | 1 | S |

replacing the standard 32-bit polynomial multiplications of $c$ and $\mathbf{s}$ with Number Theoretic Transforms (NTTs) under smaller moduli. The primary motivation for this precision transition stems from a striking performance disparity: based on the official reference implementations and the profiling results in [HAZ+24], the 32-bit NTT in Dilithium executes approximately 3-4× slower than the 16-bit NTT in Kyber.

To uncover the root cause of this severe performance gap, an analysis of the AVX2 instruction set reveals a fundamental efficiency bias toward 16-bit arithmetic. Specifically, the AVX2 instruction set provides highly optimized 16-bit multiplication instructions, such as `vpmullw` and `vpmulhw`, which directly isolate the lower or higher 16 bits of the product. This granular control is exceptionally convenient for the Montgomery reduction required in NTT butterfly computations. In contrast, while AVX2 supports lower 32-bit multiplication via `vpmulld`, it critically lacks a dedicated instruction for extracting the higher 32 bits. Consequently, full 32-bit modular reductions are forced to rely on the `vpmuldq` instruction, which computes the 64-bit product of only the even-indexed 32-bit elements. To perform parallel multiplication over all eight 32-bit coefficients within a YMM register, this architectural limitation necessitates executing two `vpmuldq` instructions interleaved with data shuffling (e.g., `vmovshdup`) to align the odd-indexed elements. Furthermore, a 256-bit YMM register can accommodate sixteen 16-bit coefficients but only eight 32-bit coefficients. Consequently, because specific small coefficients can be safely bounded within 16 bits, this substitution not only bypasses the heavy 32-bit instruction penalties but also effectively doubles the SIMD vectorization density.

## 3.1 Small NTTs for $cs_i$

### 3.1.1 Modulus Selection

Based on the preceding instruction-level analysis, we systematically apply the 16-bit NTT over the modulus 769 to compute $cs_i$ across all three Dilithium parameter sets. The rationale for this architectural decision is twofold: (1) The 16-bit NTT exhibits saturated optimization on the AVX2 instruction set, and empirical profiling indicates negligible performance variations among different 16-bit prime moduli. (2) As detailed subsequently, our optimization for $ct_0$ employs a multi-moduli NTT utilizing $q_0 = 769$ and $q_1 = 3329$. By

standardizing the $cs_i$ computations on $NTT_{769}$, the intermediate transformed vectors and precomputed twiddle factors residing in the L1 cache can be directly reused, significantly reducing computational and memory overhead.

### 3.1.2 Layer Merging

Memory access latency constitutes a dominant bottleneck in SIMD NTT execution. In the official reference implementation of Dilithium [ABD+20], layers 2 through 8 of the 8-layer NTT are merged to improve temporal data locality. We adapt this identical methodology, applying aggressive layer merging to the 16-bit NTT framework. To minimize external memory operations to the absolute theoretical limit, we partition the transform into two distinct operational sessions: a 1+6 forward NTT and a 6+1 INTT. The microarchitectural justification for this specific division lies in the register file capacity, the maximum capacity of an AVX2 SIMD register is 256 bits. Processing a 6-layer NTT sub-tree requires only in-register data permutations, entirely bypassing L1 cache accesses during these dense arithmetic stages.

### 3.1.3 Incomplete NTT and Lazy Reduction

The factorization pattern of $q_0 = 769$ directly mandates the use of an incomplete NTT, where the transform depth is reduced from the standard $\log_2(256) = 8$ layers to just $\log_2(128) = 7$ layers. The truncation of an entire layer in both the forward and inverse transforms provides a primary source of computational speedup. However, this structural efficiency necessitates a compensatory adjustment during the pointwise multiplication phase. Because the incomplete transform halts at layer 7, the output elements reside in rings isomorphic to $\mathbb{Z}_q[Y]/(Y^2 - \gamma_j)$, rendering the pointwise multiplication a degree-1 polynomial operation rather than a simple scalar product. Despite this added complexity, the computational savings from eliminating three full NTT layers vastly outweigh the pointwise overhead. Ultimately, this design represents a highly favorable trade-off, heavily validated by the parameter choices in Kyber [BDK+18].

Furthermore, as mathematically bounded in Section 2.4, the coefficient growth of $c$ and $s$ remains strictly within safe limits. Consequently, the Barrett reduction algorithm is unnecessary during the modular addition and subtraction operations within the NTT's butterfly stage. By eliminating this redundant computation, the implementation of the NTT is substantially accelerated.

## 3.2 Multi-Moduli NTT for $ct_0$

### 3.2.1 Range Analysis

When applying the multi-moduli NTT to the Dilithium scheme, strict algebraic constraints must be satisfied. As established in Section 2.4, the maximum absolute coefficient value in the product polynomial $c \cdot t_0$ is analytically bounded by $\gamma = 245760$. To safely compute this convolution in the integer domain before recovering the correct result [CHK+21, HAZ+24], the selected composite modulus $Q$ must satisfy $Q > 2\gamma = 491520$. Consequently, $ct_i$ does not need to be calculated using the original Dilithium modulus. We define the composite modulus $Q = 769 \times 3329 = 2560001$, which strictly satisfies $2560001 > 491520$. Utilizing the Residue Number System (RNS), the computation is decomposed into two independent 16-bit moduli channels.

Crucially, to ensure compatibility with AVX2 16-bit multiplication instructions, intermediate values must not exceed the 16-bit signed integer limit. Because our forward and inverse NTT processes utilize a signed Montgomery reduction, the maximum coefficient growth is bounded by $q_i$ per butterfly layer. As theoretically guaranteed in Section 2.4, after seven transform layers, the maximum coefficient magnitude is bounded

by $2^{12} + 7 \times 3329 < 2^{15}$. This mathematical guarantee prevents overflow and ensures exact correctness.

### 3.2.2 NTT with Composite Modulus

Based on the RNS analysis and the properties of incomplete NTTs, the explicit isomorphism is established as:

$$\mathbb{Z}_Q \cong \mathbb{Z}_{769} \times \mathbb{Z}_{3329}; \tag{3}$$

Overall, the polynomial multiplication over $\mathbb{Z}_Q[X]/(X^{256}+1)$ is separately computed utilizing the faster 16-bit NTTs over $R_{q_i} = \mathbb{Z}_{q_i}[X]/(X^{256}+1)$. Following the 16-bit NTTs, pointwise multiplications, and INTTs in their respective domains, the explicit CRT is executed to reconstruct the 32-bit results. The detailed implementation is provided in Algorithm 4.

Building upon the implementation in [HAZ+24], we structurally modify the **UnpackSK** and **PolyChallenge** functions to generate two explicit copies of $c$ and $t_0$, pre-storing them as 16-bit arrays mapped to the bottom and top halves of $c_{32}$ and $t_{32}$. The advantage of this memory layout is that a significantly higher density of coefficients can be loaded into the RAM simultaneously, drastically increasing cache hit rates and mitigating memory access latency.

**Algorithm 4** Multi-moduli NTT for computing 32-bit NTT with AVX2

**Input:** Declare arrays: `int32_t c_32[256], t_32[256]`
**Output:** `int32_t res_32[256]`

1: `c_32[256]` $\leftarrow [c|c]$; `t_32[256]` $\leftarrow [t|t]$ ▷ Pre-store two copies as 16-bit array to $c_{32}$ $t_{32}$
2: `cl_16, ch_16` $=$ DoubleNTT$_{q_0q_1}$(`c_32[256]`) ▷ $\hat{c}_0 = \text{NTT}_{q_0}(c)$;$\hat{c}_1 = \text{NTT}_{q_1}(c)$
3: `tl_16, th_16` $=$ DoubleNTT$_{q_0q_1}$(`t_32[256]`) ▷ $\hat{t}_0 = \text{NTT}_{q_0}(t)$; $\hat{t}_1 = \text{NTT}_{q_1}(t)$
4: `tmp_32[256]` $=$ DoubleBasemul$_{q_0q_1}$(`cl_16, ch_16, tl_16, th_16`)
5: `tmpl_16, tmph_16` $=$ DoubleINTT$_{q_0q_1}$(`tmp_32[256]`)
6: `res_32[256]` $=$ CRT(`tmpl_16, tmph_16`) ▷ CRT(INTT$_{q_0}(\hat{c}_0 \cdot \hat{t}_0)$, INTT$_{q_1}(\hat{c}_1 \cdot \hat{t}_1)$)
7: **return** `res_32`

### 3.2.3 Vectorized Implementation of CRT

Chung et al. [CHK+21] proposed the explicit CRT algorithm based on divided-difference interpolation, demonstrating its superiority over traditional Lagrangian methods [MS90] for lattice cryptography schemes when the number of moduli is minimal. While Huang et al. [HAZ+24] successfully adapted this structure for Dilithium on the Cortex-M4, their implementation heavily relies on the Plantard reduction, which is strictly tailored for Armv7-M architectures. Because our RNS decomposition utilizes exactly two moduli, the explicit divided-difference CRT remains mathematically optimal for our design. However, to efficiently map this reconstruction onto wide SIMD platforms, we substitute the Plantard reduction with the Montgomery reduction. This critical algorithmic shift aligns perfectly with the AVX2 architecture, as Montgomery reduction natively leverages the highly optimized 16-bit vector multiplication instructions (e.g., `vpmullw` and `vpmulhw`). Consequently, building upon these foundations, we design a customized, fully vectorized CRT reconstruction strategy for Dilithium. By explicitly orchestrating AVX2 and SSE registers, our implementation not only processes multiple coefficients concurrently but also elegantly resolves the complex 16-bit to 32-bit cross-precision layout conflicts inherent in the final multiply-add stage.

The instruction sequence of the CRT on AVX2 vectorizes 256 pairs of coefficients ($u_0$ and $u_1$) to calculate $u = u_0 + ((u_1 - u_0)m_1 \pmod{\pm q_1})q_0$. To seamlessly return the

result from the Montgomery domain to the normal integer domain, we pre-multiply $m_1$ by the unit length $\beta$, and pre-compute the constant $m_1' = m_1 \cdot q^{-1}$. This precomputation explicitly eliminates one parallel multiplication instruction per loop iteration. Following this modular reduction, the intermediate value, denoted as $tmp = (u_1 - u_0)m_1 \pmod{\pm q_1}$, is tightly bounded within the range $(-q_1, q_1)$. The detailed implementation is provided in Algorithm 5

**Algorithm 5** Using AVX2 vectorisation for CRT

**Input:** $u_0 = u \pmod{q_0}$, $u_1 = u \pmod{q_1}$, $m_1 = q_0^{-1}\beta \pmod{\pm q_1}$, $m_1' = m_1 \cdot q_1^{-1} \pmod{\beta}$, $\beta = 2^{16}$
**Output:** $u = u_0 + ((u_1 - u_0)m_1 \pmod{\pm q_1})q_0$

1: **vpsubw** $u_0$ $u_1$ $tmp$ ▷ $tmp = u_1 - u_0$
2: **vpmulhw** $tmp$ $m_1$ $a_1$
3: **vpmullw** $tmp$ $m_1'$ $t$
4: **vpmulhw** $t$ $q_1$ $t$
5: **vpsubw** $t$ $a_1$ $result$ ▷ $result = (u_1 - u_0)m_1 \pmod{\pm q_1}$
6: **MLA** $result$ $q_0$ $u_0$ $result$ ▷ $result = u_0 + ((u_1 - u_0)m_1 \pmod{\pm q_1})q_0$

However, during the final step of the CRT reconstruction ($u = u_0 + tmp \cdot q_0$), a severe bottleneck at the instruction level emerges. AVX2 intrinsically lacks a dedicated vector instruction to multiply and accumulate values across different precisions. The intermediate product $tmp \cdot q_0$ fundamentally requires 32 bits of precision to prevent overflow, yet the highly efficient `vpmullw` and `vpmulhw` instructions only isolate the lower and upper 16 bits of the product into separate registers. Concurrently, the elements of the base vector $u_0$ must be explicitly expanded with sign bits from 16 bits to 32 bits to accommodate carry propagation during the final addition. Resolving this data layout conflict across different widths efficiently, without triggering value spilling or memory overhead, is the core challenge of our CRT vectorization.

Algorithm 6 provides the detailed implementation of our custom MLA macro, which computes $u = a \cdot b + c$ in 32-bit precision from 16-bit inputs. The macro processes four independent coefficient groups in sequence, each containing 16 int16 values. For each group, the addend $c$ stored in register $rh$ is first split into two 128-bit halves via `vextracti128`, and each half is sign-extended from $8 \times$ int16 to $8 \times$ int32 using `vpmovsxwd`, yielding 16 int32 values across two registers. The multiplicand $a$ in register $rl$ is then multiplied by the shared factor $b$ in register $zl$ using parallel `vpmullw` and `vpmulhw` instructions, which produce the low and high 16-bit halves of each 32-bit product respectively. Since these two halves reside in separate registers, we interleave them into correct 32-bit alignment using `vpunpcklwd` and `vpunpckhwd`, followed by a cross-lane `shuffle8` permutation to ensure that the resulting 16 int32 products are distributed consecutively across two registers. Finally, the `vpaddd` instruction adds the sign-extended addend to the interleaved products, completing the fused multiply-add for 16 coefficients in parallel.

# 4 Dilithium Optimizations on ARMv8-A NEON

This section introduces the customized optimization of the Dilithium signature algorithm using the NEON instruction set on ARM platforms. According to the parameters at the instruction level detailed in Table 2 and Table 3, significant disparities exist between AVX2 and NEON. Unlike AVX2, the AArch64 NEON instruction set lacks a dedicated vector instruction to extract the higher half of a multiplication natively, and it handles operations across 16 bits and 32 bits using identical fundamental latencies. Consequently, while packing 16-bit elements on NEON theoretically doubles the data processing throughput compared to 32-bit elements. This symmetry at the instruction level necessitates a

**Algorithm 6** The explicit MLA macro definition.

**Input:** $a \in rl$ (16×int16), $b \in zl$ (16×int16), $c \in rh$ (16×int16)
**Output:** $u = a \cdot b + c$ (16×int32, stored in two registers $r_L$, $r_H$)

▷ *Step 1: sign-extend addend c to 32-bit*
1: **vextracti128** $rh \to (rh_L, rh_H)$ ▷ split 16×int16 into two 8×int16 halves
2: **vpmovsxwd** $rh_L \to rh_L$; **vpmovsxwd** $rh_H \to rh_H$ ▷ 8×int16 → 8×int32 each
▷ *Step 2: full-width 16×16 multiplication*
3: **vpmullw** $zl, rl \to lo$ ▷ low 16 bits of $a \cdot b$
4: **vpmulhw** $zl, rl \to hi$ ▷ high 16 bits of $a \cdot b$
▷ *Step 3: interleave into 32-bit layout*
5: **vpunpcklwd** $hi, lo \to r_L$ ▷ pair low words: $[lo_0|hi_0, lo_1|hi_1, \ldots]$
6: **vpunpckhwd** $hi, lo \to r_H$ ▷ pair high words
7: **shuffle8** $r_L, r_H \to r_L, r_H$ ▷ cross-lane permute to consecutive int32
▷ *Step 4: fused add*
8: **vpaddd** $r_L, rh_L \to r_L$; **vpaddd** $r_H, rh_H \to r_H$ ▷ $u = a \cdot b + c$ in 16×int32

bifurcated optimization strategy tailored to the specific algebraic bounds of the Dilithium operands:

For the computation of $c \cdot \mathbf{s}_i$, the strictly bounded coefficients guarantee that the product fits entirely within a single 16-bit modulus. Consequently, employing a single-modulus 16-bit NTT directly translates the doubled vector density into a pure performance gain. For the computation of $c \cdot \mathbf{t}_0$, the larger coefficient bounds necessitate a decomposition across multiple moduli. Due to the aforementioned symmetry, the computational overhead of executing multiple NTT passes and subsequently reconstructing the results completely offsets the theoretical throughput gains. Therefore, we abandon the NTT framework for this specific operand and pivot to deeply optimize the Parallel Small Polynomial Multiplication (PSPM) algorithm [ZHS+22] using highly customized NEON instructions.

## 4.1 Small NTT for $cs_i$

### 4.1.1 NTT Optimization

Under the ARM architecture, we systematically retain the high-level algorithmic optimizations applied in the x86 environment, including aggressive layer fusion, delay reduction, and incomplete transform decomposition. However, guided by the exhaustive profiling in [BHK+22], replacing Montgomery multiplication with Barrett multiplication during NTT/INTT operations yields faster performance on the NEON pipeline. Consequently, we implemented the corresponding Barrett reduction optimizations throughout our 16-bit NTT implementation on ARM.

### 4.1.2 Asymmetric Multiplication Optimization

Because the reference work [BHK+22] utilized a fully expanded 32-bit NTT, their proposed Asymmetric Multiplication technique was never applied to Dilithium. Since our architecture forces $cs$ into an incomplete 16-bit NTT over a smaller modulus, we adapt this technique to accelerate the $c \cdot \mathbf{s}$ convolution.

Normally, Asymmetric Multiplication is designed for matrix-vector operations in Kyber. In Dilithium, we are multiplying a multidimensional polynomial vector ($\mathbf{s}$) with a one-dimensional polynomial ($c$). For precomputations, we precompute the pointwise product of the single polynomial $c$ with the corresponding NTT zeta values, storing the result in a localized $c_{heavy}$ array. We then execute polynomial multiplication between $c_{heavy}$ and each

discrete dimension of the $\mathbf{s}$ polynomial vector natively, significantly reducing redundant modular reductions and accelerating the implementation, as detailed in Algorithm 7.

**Algorithm 7** Asymmetric $2 \times 2$ Pointwise Multiplication

**Input:** Registers `c`, `l`, `h`, `t`, `a0`, `a1`, `T0`, `T1`
**Input:** Constants $q$, $q' = -q^{-1} \bmod 2^{16}$
**Output:** $c_0 + c_1 x = (\boldsymbol{a} * \boldsymbol{b})\,\mathrm{R}^{-1} \bmod (x^2 - \zeta)$

1: Map coefficients: `c` $\leftarrow c_0$, (`a0`, `a1`) $\leftarrow (a_0, a_1)$, (`T0`, `T1`) $\leftarrow (b_0, \zeta b_1)$
2: **smull** `l`, `a0`, `T0`
3: **smull2** `h`, `a0`, `T0`
4: **smlal** `l`, `a1`, `T1`
5: **smlal2** `h`, `a1`, `T1`
6: **uzp1** `t`, `l`, `h` ▷ NEON Montgomery Multiplication Step 6-10
7: **mul** `t`, `t`, $q'$
8: **smlal** `l`, `t`, $q$
9: **smlal2** `h`, `t`, $q$
10: **uzp2** `c`, `l`, `h`
11: **Rname** (`T0`, `T1`) $\leftarrow (b_1, b_0)$, `c` $\leftarrow c_1$ and **Repeat** Steps 4–10

## 4.2 Fast Sparse Polynomial Multiplication for $ct_0$

### 4.2.1 Scheme Analysis

Having established that multi-moduli decompositions offer no instruction-level advantage on the ARMv8-A architecture, we must explore algorithmic alternatives for the $c \cdot \mathbf{t}_0$ convolution that bypass the heavy overhead of standard NTT conversions. The state-of-the-art NEON baseline by Becker et al. [BHK+22] computes a full 32-bit NTT for this operation. However, executing such a dense transform fundamentally fails to exploit the extreme sparsity of the challenge polynomial $c$, whose coefficients are strictly bounded to $\{-1, 0, 1\}$ with exactly $\tau$ non-zero entries. To maximize computational efficiency, the arithmetic strategy must shift from the transform domain back to the coefficient domain. Recently, Zheng et al. [ZHS+22] proposed the Parallel Small Polynomial Multiplication (PSPM) method, bypassing the NTT entirely to compute convolutions directly via additions and subtractions. While this algorithmic shift theoretically reduces arithmetic complexity, our in-depth analysis identifies substantial room for structural refinement and extended register utilization when adapting their design to advanced AArch64 processors:

- **Heavy Preprocessing Overhead and Scalability Bottlenecks:** To achieve parallelism, PSPM necessitates a complex preprocessing phase where multidimensional vector coefficients are mathematically offset, shifted, and bit-packed into a single 64-bit word. As Dilithium scales to higher security categories, this repeated packing and unpacking introduces substantial data manipulation latency, which largely offsets the computational savings gained from bypassing the NTT.
- **Restricted Register Utilization:** The existing PSPM implementation explicitly designs its working set around a restricted baseline of only sixteen 128-bit NEON registers. However, modern ARMv8-A processors operating in the AArch64 [Arm] execution state (e.g., Cortex-A72, Apple M1) natively provide an expanded architectural state of 32 vector registers ($v0$–$v31$). Confining the inner arithmetic loop to this restricted register space artificially induces severe register pressure, forcing frequent and expensive memory load/store operations (register spilling). Remapping the algorithmic data flow to fully exploit the 32-register AArch64 state presents a critical optimization opportunity to eliminate these structural bottlenecks.

- **Data-Dependent Execution and Timing Leakage:** Their implementation iterates dynamically over $c$ and employs conditional branches (`if-else`) to execute distinct arithmetic logic depending on whether the coefficient is $1, 0$, or $-1$. In cryptographic engineering, such data-dependent branching induces variable execution latencies, inevitably introducing timing leakage vulnerabilities that are highly undesirable in modern secure deployments.

Motivated by these limitations, we retain the core algebraic insight of exploiting $c$'s sparsity, but completely redesign the computational flow from an instruction-level perspective.

### 4.2.2 Fast Sparse Polynomial Multiplication (Fast-SPM)

**Fast-SPM Preprocessing** Our approach drastically simplifies preprocessing compared to Zheng's scheme. Initially, we scan $c$ to extract and record only the raw array indices of the non-zero coefficients. This introduces negligible efficiency overhead. Secondly, to resolve timing side-channel vulnerabilities, we fundamentally alter the memory layout. During the de-encapsulation generation of $t_0$, we forcefully allocate a buffer three times the original size and pre-expand the polynomial into a continuous memory block formatted as $[t_0 \mid -t_0 \mid t_0]$ as shown in Algorithm 8. Although preprocessing is still required, this contiguous memory mapping effectively eliminates massive bit-packing overhead and guarantees zero branch divergence during fetches.

**Algorithm 8** Fast-SPM Preprocessing

**Input:** Polynomial vectors $\mathbf{t}_0 \in \mathbb{Z}^{K \times N}$ where $N = 256$, sparse challenge $c$ defined by index sets $I_{\mathsf{pos}}$ and $I_{\mathsf{neg}}$

**Output:** Extended buffers $buf \in \mathbb{Z}^{K \times 3N}$, unified index array $U$ of length $\tau$

*Phase 1: Extended Buffer Construction*

**for** $k = 0$ **to** $K - 1$ **do**
  $buf[k][0 \dots N-1] \leftarrow \mathbf{t}_0[k]$ ▷ region $[0, 255]$: positive copy
  $buf[k][N \dots 2N-1] \leftarrow -\mathbf{t}_0[k]$ ▷ region $[256, 511]$: negated copy
  $buf[k][2N \dots 3N-1] \leftarrow \mathbf{t}_0[k]$ ▷ region $[512, 767]$: positive copy
**end for**

*Phase 2: Unified Index Encoding*

$U[0 \dots |I_{\mathsf{pos}}| - 1] \leftarrow I_{\mathsf{pos}}$ ▷ positive: index unchanged, reads $+\mathbf{t}_0$
$U[|I_{\mathsf{pos}}| \dots \tau - 1] \leftarrow I_{\mathsf{neg}} + N$ ▷ negative: offset by $N$, reads $-\mathbf{t}_0$

**The Fast Sparse Polynomial Multiplication** Unlike Zheng et al.'s approach, which packs multi-dimensional vector coefficients into 64-bit elements and suffers from underutilized bit-width, our implementation maximizes register efficiency. To fully exploit the NEON register file, we divide the resulting polynomial into four 64-coefficient tiles, designating 16 registers as permanent 32-bit accumulators to store the outputs. Given the bounded ranges of $c$ and $t_0$ in Section 2.4, a 32-bit representation securely prevents overflow.

The mathematical elegance of the Fast-SPM algorithm lies in transforming the negacyclic convolution into pure index-shifted additions. Let $v$ represent the expanded array generated during preprocessing, covering indices from $-2n$ to $n-1$:

$$v_i = \begin{cases} t_i & \text{if } 0 \le i \le n-1 \\ -t_{n+i} & \text{if } -n \le i \le -1 \\ t_{2n+i} & \text{if } -2n \le i \le -1-n \end{cases}$$

The standard negacyclic convolution for the $i$-th coefficient is defined as $u_i = \sum_{j=0}^{n-1} c_j \cdot v_{i-j}$. Because the non-zero coefficients of $c$ are strictly bounded to $\pm 1$, we completely eliminate multiplication. For $c_j = 1$, the algorithm directly accumulates $v_{i-j}$. For $c_j = -1$, subtracting $v_{i-j}$ is algebraically equivalent to adding $v_{i-j-n}$. Consequently, the accumulation formula simplifies to:

$$u_i = \sum_{c_j=1} v_{i-j} + \sum_{c_j=-1} v_{i-j-n}$$

The inner loop iterates through the non-zero indices of $c$. Based on the contiguous $[t_0 \mid -t_0 \mid t_0]$ layout, the registers utilize deterministic index offsets to retrieve the correctly sign-adjusted $t_0$ coefficients without branching. Storing the $t_0$ coefficients as 16-bit arrays allows the system to load 64 coefficients using only eight registers, minimizing memory load overhead.

To further accelerate the accumulation process, we implement **Batch Computation** as shown in Algorithm 9. Direct accumulation relies on the widening `saddw` instruction, which processes only four coefficients per operation, leaving half of the 128-bit register bandwidth unused. Instead, we safely pre-accumulate natively within intermediate 16-bit registers using the high-throughput `add` instruction (eight coefficients per cycle). Mathematically, because $7 \times 2^{12} = 28672 < 32767$, we can safely pre-accumulate across seven consecutive indices without triggering an integer overflow. We execute exactly seven 16-bit `add` instructions before issuing a single `saddw` instruction, heavily amortizing the widening cost and doubling the SIMD throughput.

**Algorithm 9** Batch Computation for $c \cdot t_0$

**Input:** Center pointer $ctr$ into $[+t_0 \mid -t_0 \mid +t_0]$; unified index $U[0..\tau-1]$; count $\tau$
**Output:** $target[0..N-1]$: output coefficients (int32)

**for** $t = 0$ **to** 3 **do** ▷ 4 tiles × 64 coefficients = $N$
  $base \leftarrow ctr + t \cdot 64$; $\mathbf{acc}_{32}[0..15] \leftarrow \mathbf{0}$ ▷ 16 int32 persistent accumulators
  $p \leftarrow 0$; $rem \leftarrow \tau$
  **while** $rem > 0$ **do** ▷ $\lceil \tau/7 \rceil$ batches, constant-time
    $B \leftarrow \min(7, rem)$ ▷ overflow-safe: $7 \times 2^{12} = 28672 < 2^{15}-1$
    $\mathbf{acc}_{16}[0..7] \leftarrow \mathbf{0}$ ▷ 8 int16 batch accumulators (64 coefficients)
    **for** $i = 0$ **to** $B - 1$ **do**
      $a \leftarrow base - U[p+i]$
      **for** $r = 0$ **to** 7 **do**
        $\mathbf{acc}_{16}[r]$ += $\texttt{load}_{8\times i16}(a + 16r)$ ▷ `add v.8h`: 8-wide accumulation
      **end for**
    **end for**
    $\mathbf{acc}_{32}[0..15]$ += `saddw/saddw2`($\mathbf{acc}_{16}[0..7]$) ▷ widen: 8 × (int16 → int32)
    $p \leftarrow p + B$; $rem \leftarrow rem - B$
  **end while**
  `store` $\mathbf{acc}_{32}[0..15] \rightarrow target[64t\,..\,64t{+}63]$
**end for**

During **Two-Way Loop Unrolling**, the implementation exploits the instruction-level parallelism of the ARM NEON processor as shown in Algorithm 10. The inner loop body is unrolled by a factor of two, forcing the CPU to process two distinct indices of $c$ simultaneously. This microarchitectural configuration drastically increases intra-loop instruction density and halves the execution frequency of the loop instruction (`b.ne`). Furthermore, processing two streams concurrently expands the temporal distance between consecutive writes to the identical accumulator register to exactly 15 instructions, flawlessly eliminating Read-After-Write (RAW) data hazards. Crucially, loop unrolling is integrated

with **Software Pipelining**. One pair of coefficients is preloaded before the loop begins. Inside the loop body, the address computation instructions (`ldrsh`, `sub`) for the next coefficient pair are interleaved between the accumulation instructions (`add`) of the current pair, while `prfm` instructions prefetch the next iteration's data into L1 cache. This overlap hides load-to-use latencies behind independent arithmetic execution, effectively eliminating pipeline stalls.

**Algorithm 10** Two-Way Unrolled Inner Loop (Lines 8–10 of Algorithm 9)

**Input:** Base address *base*; index array $U$; offset $p$; batch size $B$; accumulators $\mathbf{acc}_{16}[0..7]$
**Output:** Updated $\mathbf{acc}_{16}$; advanced offset $p$

1: $pairs \leftarrow \lfloor B/2 \rfloor$ ▷ `lsr`: halve iteration count
▷ — Prologue: preload first pair to fill pipeline —
2: $a_1 \leftarrow base - U[p]$; $a_2 \leftarrow base - U[p{+}1]$; $p \leftarrow p + 2$
3: $pairs \leftarrow pairs - 1$

4: **for** $i = 1$ **to** $pairs$ **do** ▷ 2×-unrolled, software-pipelined kernel
5: $\mathbf{d} \leftarrow$ `ldp` $[a_1{+}0]$, $[a_1{+}32]$ ▷ $\text{idx}_1$ front half: load 32 int16
6: $a'_1 \leftarrow base - U[p]$; $a'_2 \leftarrow base - U[p{+}1]$; $p \leftarrow p + 2$ ▷ pipeline: preload next pair
7: $\mathbf{acc}_{16}[0..3]$ += $\mathbf{d}$ ▷ `add v.8h`: 8-wide, latency of Line 7 hidden
8: $\mathbf{d} \leftarrow$ `ldp` $[a_1{+}64]$, $[a_1{+}96]$ ▷ $\text{idx}_1$ back half
9: $\mathbf{acc}_{16}[4..7]$ += $\mathbf{d}$
10: $\mathbf{d} \leftarrow$ `ldp` $[a_2{+}0]$, $[a_2{+}32]$ ▷ $\text{idx}_2$ front half
11: `prfm` $[a'_1]$, $[a'_2]$ ▷ L1 prefetch for next iteration
12: $\mathbf{acc}_{16}[0..3]$ += $\mathbf{d}$ ▷ $v_{16}$ write gap $\geq 15$ instr. $\Rightarrow$ RAW-free
13: $\mathbf{d} \leftarrow$ `ldp` $[a_2{+}64]$, $[a_2{+}96]$ ▷ $\text{idx}_2$ back half
14: $\mathbf{acc}_{16}[4..7]$ += $\mathbf{d}$
15: $a_1 \leftarrow a'_1$; $a_2 \leftarrow a'_2$ ▷ advance to next iteration
16: **end for**
▷ — Epilogue: last pair, no prefetch —
17: $\mathbf{acc}_{16}[0..7]$ += `ldp` from $[a_1]$; $\mathbf{acc}_{16}[0..7]$ += `ldp` from $[a_2]$
18: **if** $B \bmod 2 = 1$ **then** ▷ tail: odd remainder
19: $\mathbf{acc}_{16}[0..7]$ += `ldp` from $[base - U[p]]$; $p \leftarrow p + 1$
20: **end if**

Because the Fast-SPM solution is also applicable to the x86 AVX2 architecture, we provide an implementation utilizing AVX2 instructions. Although this variant achieves higher signature efficiency for Dilithium than the official reference implementation, it does not reach the performance level of the multi-mode NTT optimization discussed in the previous section. A detailed performance comparison is provided in Chapter 5.

# 5 Results and Comparisons

## 5.1 Benchmark setting

We will describe the benchmark test results of dilithium in skylake using AVX2 and Cortex-A72, and M1 Pro using NEON.

**Benchmarking setup for Skylake** The cycle counts for AVX2 were obtained on a Intel Core i7-8750U (Skylake) processor with a base frequency of 2.2 GHz. As usual we disable TurboBoost and hyperthreading. We compile our implementations with gcc version 11.4.0 and use the compiler flags -O3, -fomit-frame-pointer, -march=native, -mtune=native. All cycle counts are the average cycle counts of 100 000 executions.

**Benchmarking setup for the Cortex A72** To evaluate performance on the ARMv8-A architecture, we utilize a Raspberry Pi 4 Model B equipped with a quad-core Broadcom BCM2835 processor operating at 1.6 GHz. For the underlying symmetric cryptographic operations, our benchmarking environment integrates the specialized NEON-based SHA-3 and SHAKE implementations provided by Becker et al. [BHK+22]. The source code is compiled using GCC version 11.4.0 with the -O3 optimization flag. To ensure statistical reliability, all reported measurements represent the median cycle count obtained across 10 000 independent executions

**Benchmarking setup for the M1 Pro** For the evaluation on the Apple silicon platform, we deploy a 2021 MacBook Pro featuring the M1 Pro system-on-chip. This processor comprises eight high-performance Firestorm cores alongside two energy-efficient Icestorm cores. The symmetric hashing operations rely on the Keccak implementation authored by Westerbaan [Wes], which explicitly leverages the dedicated cryptographic instructions available on this architecture. To obtain precise cycle measurements, we adopt the profiling methodology established by Becker et al. [BHK+22] and utilize a customized adaptation of their testing tool. All binaries are generated via Clang version 15.0.0 with the -O3 optimization level. Following this established framework, the documented results reflect the average cycle count derived from 10 000 continuous executions.

## 5.2 Performance of Core Polynomial Arithmetic

This section evaluates the core polynomial multiplications involving the sparse challenge polynomial $c$ using the AVX2 and NEON instruction sets. We first benchmark the foundational arithmetic primitives, comparing the performance of the customized 16-bit NTT, INTT, and pointwise operations against the standard 32-bit reference implementations. Subsequently, we assess the overall performance of the small polynomial multiplications, specifically $c \cdot \mathbf{s}_i$ and $c \cdot \mathbf{t}_0$. To demonstrate the efficacy of the proposed optimizations, we compare the benchmarking results against established baselines. On the AVX2 platform, we evaluate the implementation against the designs from [ABD+20]. On the NEON platforms (Cortex-A72 and M1 Pro), we benchmark the proposed schemes against the highly optimized baselines presented in [BHK+22].

### 5.2.1 NTT-related functions

Table 4 presents the benchmarking results for the core NTT-related functions across different architectures. On the AVX2 platform (Skylake), migrating from the standard 32-bit arithmetic [ABD+20] to the single-modulus 16-bit implementations yields substantial performance gains. Specifically, the 16-bit operations over $q = 3329$ and $q = 769$ achieve speedups ranging from 4.08× to 4.16× for the NTT, 1.99× to 2.50× for the INTT, and 2.17× to 2.26× for the pointwise multiplications. For the multi-moduli approach ($q_1 \times q_2$), the dual 16-bit NTTs execute in 448 cycles, demonstrating a 1.96× acceleration over the 32-bit baseline. The subsequent combined phase—encompassing the dual INTTs, pointwise multiplications, and the CRT reconstruction—consumes a total of 1085 cycles, of which the CRT step explicitly accounts for 132 cycles. Although this complete reconstruction phase (1085 cycles) incurs a slight overhead compared to the 991 cycles required by the standard 32-bit INTT and pointwise operations, the overall polynomial multiplication delivers superior performance compared to the 32-bit approach. Furthermore, the implementation introduces a fully vectorized divided-difference CRT strategy customized for Dilithium, establishing a highly competitive alternative on AVX2.

On the ARMv8-A NEON platforms, utilizing the 16-bit single-modulus NTT ($q = 3329$) consistently outperforms the highly optimized 32-bit implementations provided by Becker et al. [BHK+22]. On the Cortex-A72, the modification yields speedups of 1.89×, 2.25×,

and 1.45× for the NTT, INTT, and pointwise operations, respectively. Similarly, on the Apple M1 Pro platform, the proposed method maintains its advantage, delivering 1.88×, 2.31×, and 1.17× speedups for the corresponding NTT, INTT, and pointwise operations relative to the state-of-the-art baseline.

**Table 4:** Performance Comparison of NTT-Related Functions Across AVX2 and NEON Platforms

| Platform | Modulus $q$ | Ref. | NTT | INTT | Pointwise | CRT |
|---|---|---|---|---|---|---|
| Skylake | 8380417 | [ABD+20] | 877 | 826 | 165 | – |
| | 3329 | This work | 215 | 331 | 76 | – |
| | 769 | This work | 211 | 416 | 73 | – |
| | $769 \times 3329$ | This work | 448 | | 1085† | 132 |
| Cortex-A72 | 8380417 | [BHK+22] | 1797 | 2532 | 600 | – |
| | 3329 | This work | 952 | 1123 | 413 | – |
| M1 Pro | 8380417 | [BHK+22] | 426 | 527 | 117 | – |
| | 3329 | This work | 226 | 228 | 100 | – |

† Total combined cycles for dual INTTs, pointwise multiplications, and CRT.

### 5.2.2 Multiplications with the challenge polynomial

Table 5 details the cycle counts for the polynomial multiplications $c \cdot \mathbf{s}_i$ and $c \cdot \mathbf{t}_0$, which constitute the primary computational bottlenecks within the Dilithium signature generation loop. To maximize performance across diverse target microarchitectures, we eschew a monolithic algorithmic design in favor of platform-tailored algebraic strategies.

For the products $c \cdot \mathbf{s}_1$ and $c \cdot \mathbf{s}_2$, we uniformly deploy our highly optimized 16-bit single-modulus NTT. On the AVX2 (Skylake) microarchitecture, this reduced-precision arithmetic yields substantial 2.4–2.5× speedups across all security levels relative to the official 32-bit baseline [ABD+20]. Similarly, on ARMv8-A platforms, our 16-bit approach markedly outperforms the 32-bit Barrett-based NTT/INTT methodology presented in [BHK+22]. Specifically, we achieve speedups of 2.0–2.1× on the Cortex-A72 and 1.9–2.0× on the Apple M1 Pro.

For the computation of $c \cdot \mathbf{t}_0$, where larger coefficient bounds necessitate distinct algebraic handling, our optimization strategies diverge based on hardware characteristics. On Skylake, the 16-bit multi-moduli NTT, coupled with a fully vectorized CRT reconstruction, proves highly efficient, delivering consistent 1.2–1.3× improvements over the 32-bit NTT baseline. Conversely, on NEON pipelines, where the multi-moduli approach is architecturally constrained, our proposed Fast-SPM serves as a direct, NTT-free alternative. This tailored approach accelerates $c \cdot \mathbf{t}_0$ by 1.1–1.8× on the Cortex-A72 and 1.1–1.9× on the M1 Pro compared to the state-of-the-art [BHK+22]. Ultimately, these results empirically validate that tightly co-designing algorithms parameters with the underlying SIMD pipelines systematically outperforms architecture-agnostic implementations.

## 5.3 Performance of Dilithium Signature Generation

As our microarchitecture-tailored optimizations primarily accelerate the polynomial arithmetic within the rejection sampling loop, our system-level evaluation focuses exclusively on the Dilithium signature generation phase. Table 6 details the aggregate cycle counts and relative performance gains across the targeted platforms.

**Table 5:** Cycle counts for small polynomial multiplications in Dilithium signature generation.

| | | **Dilithium2** | | | **Dilithium3** | | | **Dilithium5** | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Platform** | **Operation** | Prior | Ours | × | Prior | Ours | × | Prior | Ours | × |
| Skylake [ABD+20] | $c \cdot \mathbf{s}_1$ | 8391 | **3334** | 2.5 | 10143 | **4099** | 2.5 | 13935 | **5682** | 2.5 |
| | $c \cdot \mathbf{s}_2$ | 8310 | **3368** | 2.5 | 12004 | **4853** | 2.5 | 15822 | **6492** | 2.4 |
| | $c \cdot \mathbf{t}_0$ | | **6610** | 1.3 | | **9689** | 1.2 | | **12749** | 1.2 |
| Cortex-A72 [BHK+22] | $c \cdot \mathbf{s}_1$ | 22535 | **11108** | 2.0 | 28059 | **13595** | 2.1 | 38485 | **18580** | 2.1 |
| | $c \cdot \mathbf{s}_2$ | 22800 | **11082** | 2.1 | 33522 | **16053** | 2.1 | 43126 | **21067** | 2.1 |
| | $c \cdot \mathbf{t}_0$ | | **13008** | 1.8 | | **22978** | 1.5 | | **38249** | 1.1 |
| M1 Pro [BHK+22] | $c \cdot \mathbf{s}_1$ | 4876 | **2430** | 2.0 | 5988 | **2977** | 2.0 | 8215 | **4071** | 2.0 |
| | $c \cdot \mathbf{s}_2$ | 4909 | **2429** | 2.0 | 6682 | **3525** | 1.9 | 8966 | **4617** | 1.9 |
| | $c \cdot \mathbf{t}_0$ | | **2612** | 1.9 | | **4960** | 1.3 | | **8467** | 1.1 |

*Note:* For the baseline implementations [ABD+20, BHK+22], the convolutions $c \cdot \mathbf{s}_2$ and $c \cdot \mathbf{t}_0$ exhibit identical cycle counts. This is mathematically expected, as both operand vectors share the same dimension $k$, and the standard 32-bit NTT arithmetic executes in constant time regardless of the differing coefficient bounds ($\eta$ vs. $2^{13} - 1$).

On the Skylake (AVX2) microarchitecture, we benchmark our implementation against the official reference [ABD+20] and the current state-of-the-art [XHL+24]. By optimizing the core polynomial multiplications, our design reduces the overall signature generation latency by 8.2%, 7.6%, and 7.9% for Dilithium2, Dilithium3, and Dilithium5, respectively, relative to the official baseline. Furthermore, our approach systematically outperforms the highly optimized implementation in [XHL+24], setting a new state-of-the-art performance benchmark for Dilithium signature generation on this architecture. Crucially, as the algorithmic enhancements proposed in [XHL+24] are largely orthogonal to our microarchitecture-driven arithmetic strategies, these optimizations can be seamlessly integrated into our design to potentially unlock further performance gains

On the ARMv8-A NEON platforms, we benchmark our integrated implementation against the highly optimized baseline established by Becker et al. [BHK+22]. The incorporation of our Fast-SPM algorithm alongside the 16-bit single-modulus NTT yields a systematic reduction in execution cycles during signature generation. Specifically, our design achieves overall performance improvements ranging from 9.7% to 13.2% on the Cortex-A72, and 11.6% to 12.9% on the Apple M1 Pro across the three security levels.

## 6 Conclusion

In this work, we presented a SIMD optimization strategy for Dilithium polynomial multiplication on AVX2 and ARMv8-A NEON platforms. The central conclusion of this study is that efficient implementation in this setting requires jointly considering operand-specific parameter characteristics and architecture-specific SIMD execution characteristics, rather than applying a uniform acceleration method across all polynomial products. Based on this principle, we replaced the conventional 32-bit NTT for $c \cdot \mathbf{s}$ with a more efficient 16-bit NTT under carefully selected moduli, and we adopted different techniques for $c \cdot \mathbf{t}_0$ according to the target platform: a multi-moduli incomplete NTT with vectorized divided-difference CRT reconstruction on AVX2, and Fast Sparse Polynomial Multiplication (Fast-SPM) on NEON. In particular, the optimized implementations provide substantial acceleration for the targeted polynomial multiplications and reduce the overall signing time by 8% to 13%

**Table 6:** Performance comparison of Dilithium signature generation.

| | | Dilithium2 | | Dilithium3 | | Dilithium5 | |
|---|---|---|---|---|---|---|---|
| **Platform** | **Work** | Cycles | Impr. | Cycles | Impr. | Cycles | Impr. |
| Skylake (AVX2) | [ABD+20] | 263045 | 8.2% | 421953 | 7.9% | 504319 | 7.9% |
| | [ZZS+25] | 295523 | 18.3% | 460942 | 15.7% | 559587 | 17.0% |
| | PQClean | 272263 | 11.3% | 471953 | 17.7% | 531481 | 12.6% |
| | [XHL+24] | 253852 | 4.9% | 405604 | 4.2% | 479968 | 3.3% |
| | Fast-SPM(AVX2) | 251763 | 4.1% | 403454 | 3.7% | 477346 | 2.7% |
| | This work | **241485** | | **388409** | | **464352** | |
| Cortex-A72 (NEON) | [ZHS+22][a] | 3880847 | | 6786351 | | 10190913 | |
| | [BHK+22] | 651312 | 13.2% | 794791 | 10.5% | 1252408 | 9.7% |
| | This work | **565036** | | **711422** | | **1131520** | |
| M1 Pro (NEON) | [ZHS+22][a] | 1581485 | | 2729018 | | 2207382 | |
| | [BHK+22] | 240232 | 12.9% | 372057 | 12.7% | 451740 | 11.6% |
| | This work | **209298** | | **324905** | | **399284** | |

[a] *Note:* The implementation in [ZHS+22] specifies the usage of 16 NEON registers, indicating it was developed for the AArch32 architecture. In contrast, our implementation natively leverages the AArch64 architecture (which provides 32 NEON registers), contributing to the significant performance disparity.

on representative x86 and ARM processors. More broadly, this work indicates that the efficient software realization of post-quantum schemes can benefit significantly from combining tight algebraic bounds with detailed awareness of microarchitectural behavior. We expect that the same design perspective will be useful for further optimization of Dilithium and other lattice-based cryptographic schemes on modern SIMD-enabled platforms.

## References

[ABD+20] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/dilithium/.

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR TCHES*, 2022(1):127–151, 2022.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 2022*, volume 13269 of *LNCS*, pages 853–871. Springer, Cham, June 2022.

[Arm] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture (DDI 0487)*. Arm Limited. Section B1.2: Registers in AArch64 Execution state. Available at https://developer.arm.com/documentation/ddi0487/mb/-Part-B-The-AArch64-Application-Level-Architecture/

-Chapter-B1-The-AArch64-Application-Level-Programmers--Model/-B1-2-Registers-in-AArch64-Execution-state?lang=en.

[BBCT22] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 845–862, Boston, MA, August 2022. USENIX Association.

[BDK+18] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. Crystals - kyber: A cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 353–367, 2018.

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR TCHES*, 2022(1):221–244, 2022.

[BMK+22] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhede. Polynomial multiplication on embedded vector architectures. *IACR TCHES*, 2022(1):482–505, 2022.

[BS07] Daniel J. Bernstein and J. Sorenson. Modular exponentiation via the explicit chinese remainder theorem. *Math. Comput.*, 76:443–454, 2007.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings. *IACR TCHES*, 2021(2):159–188, 2021.

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR TCHES*, 2021(1):1–24, 2021.

[HAZ+24] Junhao Huang, Alexandre Adomnicai, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting Keccak and Dilithium implementations on ARMv7-M. *IACR TCHES*, 2024(2):1–24, 2024.

[HKK+25] Ottó Hanyecz, Alexander Karenin, Elena Kirshanova, Péter Kutas, and Sina Schaeffler. Constant time lattice reduction in dimension 4 with application to SQIsign. *IACR TCHES*, 2025(2):511–534, 2025.

[HLY24] Vincent Hwang, Chi-Ting Liu, and Bo-Yin Yang. Algorithmic views of vectorized polynomial multipliers - NTRU prime. In Christina Pöpper and Lejla Batina, editors, *ACNS 2024, Part II*, volume 14584 of *LNCS*, pages 24–46. Springer, Cham, March 2024.

[Hwa24] Vincent Hwang. Pushing the limit of vectorized polynomial multiplications for NTRU prime. In Tianqing Zhu and Yannan Li, editors, *ACISP 24, Part II*, volume 14896 of *LNCS*, pages 84–102. Springer, Singapore, July 2024.

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR TCHES*, 2022(4):614–636, 2022.

[LHP+23] Douwei Lei, Debiao He, Cong Peng, Min Luo, Zhe Liu, and Xinyi Huang. Faster implementation of ideal lattice-based cryptography using avx512. *ACM Trans. Embed. Comput. Syst.*, 22(5), September 2023.

[MS90] Peter L. Montgomery and Robert D. Silverman. An fft extension to the $p-1$ factoring algorithm. *Mathematics of Computation*, 54(190):839–854, 1990.

[NG21] Duc Tri Nguyen and Kris Gaj. Fast NEON-based multiplication for lattice-based NIST post-quantum cryptography finalists. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*, pages 234–254. Springer, Cham, 2021.

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018.

[Wes] Bas Westerbaan. Keccak-f1600 on aarch64. https://github.com/bwesterb/armed-keccak.

[XHL+24] Runqing Xu, Debiao He, Min Luo, Cong Peng, and Xiangyong Zeng. Optimizing dilithium implementation with avx2/-512. *ACM Trans. Embed. Comput. Syst.*, 23(6), September 2024.

[ZHS+22] Jieyu Zheng, Feng He, Shiyu Shen, Chenxi Xue, and Yunlei Zhao. Parallel Small Polynomial Multiplication for Dilithium: A Faster Design and Implementation. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, pages 304–317, 2022. https://doi.org/10.1145/3564625.3564629.

[ZZS+25] Jieyu Zheng, Haoliang Zhu, Zhenyu Song, Zheng Wang, and Yunlei Zhao. Optimized vectorization implementation of crystals-dilithium. *IEEE Transactions on Dependable and Secure Computing*, 22(5):5730–5743, 2025.