electronics

MDPI

Article

ARMv8/NEON Optimization of NCC-Sign for Mixed-Radix NTT: Cycle-Accurate Evaluation on Apple M1 Pro and Cortex-A72

Minwoo Lee [1], Minjoo Sim [1], Siwoo Eum [1] and Hwajeong Seo [2,*]

[1] Department of Information Computer Engineering, Hansung University, Seoul 02876, Republic of Korea
[2] Department of Convergence Security, Hansung University, Seoul 02876, Republic of Korea
* Correspondence: hwajeong@hansung.ac.kr; Tel.: +82-760-8033

Abstract

This paper presents an ARMv8/NEON-oriented implementation of NCC-Sign targeting the NTT-friendly trinomial parameter sets (NCC-Sign-1/3/5), whose dominant cost arises from mixed-radix NTT computations with $n = 2^a \cdot 3^b$. We design lane-local SIMD kernels—including a four-lane Montgomery multiply–reduce, a centered modular reduction pass, a fused stage-0 butterfly, and streamlined radix-2/radix-3 pipelines—and extend them with three further optimizations: (i) radix-2 multi-stage butterfly merging to halve intermediate load/store traffic, (ii) a stride-3 vectorization technique exploiting NEON structure load/store instructions (vld3q/vst3q) to fully vectorize small-len radix-3 stages that would otherwise fall back to scalar execution, and (iii) NEON-parallel pointwise Montgomery multiplication. Using cycle-accurate PMU measurements under identical toolchains for baseline and optimized builds on Apple M1 Pro, we observe geometric-mean speedups of 1.40× for key generation, 2.24× for signing, and 2.01× for verification across NCC-Sign-1/3/5, with per-kernel gains of up to 5–6× for NTT/INTT and 7.5× for pointwise multiplication. To contextualize these results, we provide a direct comparison with the NEON-optimized ML-DSA (Dilithium) implementation of Becker et al. on the same platform, a cross-platform evaluation on Arm Cortex-A72 (Raspberry Pi 4), a Montgomery-versus-Barrett microbenchmark supporting our design choice, and an empirical constant-time assessment via dudect confirming that no timing leakage is detected in any NEON kernel under 30 million measurements.

Keywords: NCC-Sign; ARMv8; NEON; NTT; mixed-radix; Montgomery reduction; performance optimization; benchmarking

Academic Editor: Aryya Gangopadhyay

Received: 4 March 2026
Revised: 25 March 2026
Accepted: 28 March 2026
Published: 31 March 2026

1. Introduction

Quantum-capable adversaries pose a concrete long-term confidentiality risk: encrypted traffic exfiltrated today may become decryptable once sufficiently large quantum computers emerge (the "harvest-now, decrypt-later" threat). This concern is grounded in Shor's polynomial-time algorithms for factoring and discrete logarithms and in Grover's quadratic speedup for search [1,2]. In response, governments and industry have shifted from exploratory research toward migration planning for post-quantum cryptography (PQC), guided by joint readiness advisories and roadmaps [3].

On the standardization front, NIST has issued its first PQC standards—ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205)—thereby providing concrete, interoperable targets for near-term deployment [4–7]. These standards emphasize not only

asymptotic security but also implementation quality, performance, memory footprint, and resistance to side-channel attacks in real systems.

In parallel with the NIST effort, Korea organized the KpqC competition to evaluate candidate KEMs and signature schemes for domestic standardization. Among the Round 1 candidates promoted to Round 2 was NCC-Sign, whereas the final selection announced on 16 January 2025 standardized AIMer and HAETAE for signatures and NTRU+ and SMAUG-T for KEM [8,9]. We nevertheless select NCC-Sign as a practical optimization target and focus on its NTT-friendly trinomial parameter sets (NCC-Sign-1/3/5), for which the dominant cost arises from mixed-radix NTT computations. Unlike the power-of-two cyclotomic settings common in lattice-based signatures, NCC-Sign's trinomial instantiations adopt $n = 2^a \cdot 3^b$, which introduces radix-3 stages and additional data-movement challenges. This structure defines a technically non-trivial ARMv8 workload, because performance depends strongly on SIMD lane locality and the cost of cross-lane shuffles. NCC-Sign also supports a non-cyclotomic instantiation and non-NTT polynomial multiplication (e.g., Toom–Cook/Karatsuba), but those variants lie outside the scope of this work. Our goal is to engineer efficient NEON kernels for the NTT-friendly sets and quantify their end-to-end impact through cycle-accurate measurements on Apple M1 Pro and Arm Cortex-A72.

Beyond NCC-Sign itself, mixed-radix NTT structures of the form $n = 2^a \cdot 3^b$ appear in other lattice-based settings that depart from power-of-two cyclotomic rings, yet systematic SIMD optimization studies for such transforms on ARMv8 remain scarce. The interplay between radix-3 butterfly stages and the fixed four-lane width of NEON poses vectorization challenges absent from the well-studied power-of-two case, making NCC-Sign's trinomial parameter sets a technically informative target.

We therefore focus on an ARMv8/NEON-oriented implementation of NCC-Sign for the NTT-friendly trinomial parameter sets. ARMv8 is ubiquitous in mobile and edge deployments, and NEON intrinsics expose 128-bit SIMD operations that match the structured, data-parallel arithmetic common in lattice-based cryptography [10]. Prior work has shown that NEON-tuned ARMv8 implementations can substantially reshape performance relative to scalar-C baselines across lattice schemes such as ML-DSA (Dilithium), Kyber, Saber, Falcon, and HAETAE, underscoring the broader value of architecture-conscious PQC implementations [11–13].

Accordingly, this paper examines how architecture-agnostic algorithmic kernels can be engineered for ARMv8/NEON to deliver end-to-end gains for a lattice-signature workload while preserving arithmetic correctness and stable kernel control flow. Section 2 reviews the KpqC competition and standardization context, provides a high-level description of NCC-Sign, and summarizes the relevant ARMv8/NEON concepts. Section 3 details our ARMv8-oriented implementation. Section 4 presents the evaluation, including cross-platform and cross-scheme comparisons, function-level profiling, constant-time analysis, and a design-choice microbenchmark. Section 5 discusses transferability, energy considerations, and limitations. Section 6 concludes.

Contributions

We summarize our main contributions as follows.

- **Comprehensive ARMv8/NEON optimization of mixed-radix NTT in NCC-Sign.** Beyond the baseline NEON kernels (four-lane Montgomery multiply–reduce, centered reduction, fused stage-0 butterfly, and radix-2/radix-3 pipelines), we apply radix-2 multi-stage butterfly merging—adapting the interleaved multi-layer approach of Becker et al. [11] to the mixed-radix setting—and propose a stride-3 vectorization technique that exploits NEON structure load/store instructions (vld3q/vst3q) to fully

vectorize small-len radix-3 stages previously handled by scalar fallback code. We also vectorize the pointwise Montgomery multiplication and integrate two-way parallel hashing (fips202x2) where the algorithm structure permits (Section 3).

- **Function-level profiling and Amdahl's-law analysis.** We decompose the cycle cost of each operation into individual kernels, derive the theoretical end-to-end speedup bound, and characterize the effect and limitations of each optimization in the mixed-radix pipeline (Section 4).
- **Two-platform evaluation and cross-scheme comparison.** We benchmark on Apple M1 Pro and Arm Cortex-A72 (Raspberry Pi 4), two architecturally distinct ARMv8 cores, and provide a direct cycle-level comparison with the NEON-optimized ML-DSA (Dilithium) of Becker et al. [11] under identical measurement conditions. A Montgomery-vs.-Barrett microbenchmark supports our modular-reduction design choice (Section 4).
- **Empirical constant-time verification via dudect.** We apply the fixed-vs-random *t*-test methodology of Reparaz et al. to all NEON kernels (NTT, INTT, pointwise multiplication) and confirm that no timing leakage is detected under 30 million measurements on Apple M1 Pro (Section 4).

To support reproducibility and follow-up research, we publicly released the optimized NCC-Sign implementation for ARMv8 platforms together with the benchmarking and validation harness: https://github.com/minunejip/PQC-ARMv8 (accessed on 27 March 2026). The repository additionally includes the cycle-counting adaptations for Cortex-A72, the dudect test harness, and all raw benchmark data to facilitate independent reproduction.

2. Related Work

2.1. KpqC Competition and Standardization Context

The Korean Post-Quantum Cryptography (KpqC) competition was launched to curate a domestically standardized portfolio of KEMs and digital signatures in parallel with international standardization, with an explicit remit that went beyond asymptotic security to include implementation maturity, constant-time discipline, and transition readiness across general-purpose and embedded platforms.

As sketched in Figure 1, an initial call in 2022 led to a Round 1 cohort comprising nine digital signatures and seven KEMs; on 7 December 2023, eight candidates advanced to Round 2—four signatures (AIMer, HAETAE, MQ-Sign, and NCC-Sign) and four KEMs (NTRU+, PALOMA, REDOG, and SMAUG+TiGER (merged)) [8]. The final decision on 16 January 2025 standardized *AIMer* and *HAETAE* for signatures and *NTRU+* and *SMAUG-T* for KEM, reflecting both cryptanalytic assurance and implementability considerations [9].

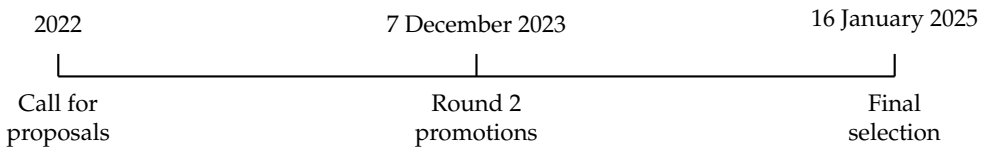

Figure 1. Schematic timeline of the KpqC competition.

Table 1 summarizes the final selections by family and hardness assumption. To situate these choices within deployment practice, complementary studies report reproducible benchmarking and constant-time code hygiene for Round 1 candidates [14], and integration of KpqC algorithms into production-grade TLS/X.509 stacks to quantify end-to-end overheads and deployment trade-offs [15]. Within this landscape, NCC-Sign is noted as a Round 2 signature candidate.

Table 1. KpqC final selections by family and hardness.

Category	Algorithm	Family	Hardness (One Line)
Signature	AIMer	Non-lattice (ZK/MITH)	OWF preimage resistance [16]
Signature	HAETAE	Lattice (module)	MLWE/MSIS [17]
KEM	NTRU+	Lattice (NTRU lineage)	NTRU problem [18]
KEM	SMAUG–T	Lattice (MLWE/MLWR)	MLWE/MLWR [19]

2.2. NCC-Sign: A Ring-LWE-Based Digital Signature Scheme

NCC-Sign is a lattice-based digital signature built on the Ring-Learning with Errors (Ring–LWE) assumption. It is intended as a lightweight yet structurally cautious alternative to mainstream lattice signatures such as ML-DSA (Dilithium). Unlike designs fixed to power-of-two cyclotomic rings, NCC-Sign uses a more flexible polynomial ring to reduce exploitable symmetries and to allow finer parameter tuning [20]. The algebraic setting is $R = \mathbb{Z}_q[X]/\phi(X)$ without restricting $\phi(X)$ to be cyclotomic. The specification provides two instantiations: (i) a *non-cyclotomic* option $\phi(X) = X^p - X - 1$ with prime p, which suppresses cyclotomic/subfield structure; and (ii) an *NTT-friendly trinomial* $\phi(X) = X^n - X^{n/2} + 1$ with $n = 2^a 3^b$ for fast transforms [20].

2.2.1. Parameters

Table 2 lists the conservative non-cyclotomic sets ($\eta = 2$). "Exp. reps." denotes the expected number of signing attempts due to rejection sampling (lower is better).

Table 2. NCC-Sign non-cyclotomic parameter sets (conservative).

Level	p	q	d	τ	γ_1	γ_2	η	β	ω	Exp. Reps.
I_c	1201	17,279,291	12	32	2^{19}	$(q-1)/70$	2	128	80	2.50
III_c	1607	17,305,741	13	32	2^{19}	$(q-1)/60$	2	128	80	3.02
V_c	2039	17,287,423	13	32	2^{19}	$(q-1)/58$	2	128	80	3.95

2.2.2. Protocol

NCC-Sign follows the Bai–Galbraith "Fiat–Shamir with aborts" pattern with rounding and hint reconciliation [21,22]. Key generation publishes a compressed $t_1 = \mathrm{HighBits}_q(t, 2^d)$ of $t = as_1 + s_2$ and retains $t_0 = t - t_1 \cdot 2^d$. Signing samples bounded y, computes w, and sets $w_1 = \mathrm{HighBits}_q(w, 2\gamma_2)$. It then derives a τ-sparse challenge $c \leftarrow \mathrm{SampleInBall}(H(\mu \| w_1))$, sets $z = y + c\,s_1$ and a hint h, and rejects unless $\|z\|_\infty < \gamma_1 - \beta$ and low-bit checks pass. Verification reconstructs $w_1' = \mathrm{UseHint}_q(h,\, az - c\,t_1 \cdot 2^d,\, 2\gamma_2)$ and accepts if $c = H(\mu \| w_1')$ and $\mathrm{wt}(h) \le \omega$. Secrets s_1, s_2 are sampled *uniformly* from bounded sets ($\eta \in \{1, 2\}$), avoiding Gaussian samplers and easing constant-time implementations.

For NTT-friendly settings, Table 3 summarizes the trinomial parameter sets used in practice.

Table 3. NCC-Sign trinomial parameter sets.

Level	n	q	d	τ	γ_1	γ_2	η	β	ω	Exp. Reps.
1	1152	8,401,537	12	25	2^{18}	131,274	1	50	80	1.93
3	1536	8,397,313	12	29	2^{18}	131,208	1	58	80	2.76
5′ †	2048	8,380,417	11	32	2^{18}	130,944	1	64	80	4.49
5	2304	8,404,993	13	32	2^{19}	262,656	1	64	80	2.32

† Level 5′ uses $n = 2048 = 2^{11}$ (a power-of-two) and adopts the Dilithium modulus $q = 8{,}380{,}417$. Its NTT is purely radix-2 and does not require the mixed radix-2/3 kernels central to this paper; we therefore excluded it from evaluation.

2.2.3. Security

Unforgeability in the (Q)ROM is argued under RLWE/RSIS/SelfTargetRSIS using standard Fiat–Shamir-with-aborts techniques; rounding/hint mechanisms follow the established analysis of Lyubashevsky and Bai–Galbraith [21,22].

2.2.4. Artifact Sizes

Table 4 compares public-key, secret-key, and signature sizes across representative PQC schemes; NCC rows correspond to the conservative sets above [20]. Numbers for ML-DSA follow FIPS 204 [6], SLH-DSA follows FIPS 205 [7], and Falcon follows its v1.2 specification [23]; the SPHINCS+ framework itself was introduced at CCS 2019 [24].

Table 4. Sizes (bytes) for keys and signatures across selected PQC schemes.

Scheme	Public Key	Secret Key	Signature
ML-DSA-44	1312	2560	2420
ML-DSA-65	1952	4032	3309
ML-DSA-87	2592	4896	4627
NCC-Sign-I_c	1984	2800	3186
NCC-Sign-III_c	2443	3914	4251
NCC-Sign-V_c	3091	4940	5385
Falcon-512	897	1281	666
Falcon-1024	1793	2305	1280
SLH-DSA-128s	32	64	7856
SLH-DSA-128f	32	64	17,088
SLH-DSA-192s	48	96	16,224
SLH-DSA-192f	48	96	35,664
SLH-DSA-256s	64	128	29,792
SLH-DSA-256f	64	128	49,856

2.3. ARMv8 and NEON SIMD: Concepts and Rationale

This work targets the ARMv8-A execution environment in *AArch64* state. The platform exposes a bank of thirty-two 128-bit SIMD/FP registers $v0$–$v31$ that are shared by Advanced SIMD (*NEON*). Under the AAPCS64 procedure-call standard, $v0$–$v7$ carry vector arguments and return values across public interfaces, $v8$–$v15$ are callee-saved, and all other SIMD registers are caller-saved; these conventions shape inlining boundaries and the way our kernels spill vector state when needed [25].

NEON provides a fixed-width (128-bit) data-parallel model with integer and floating-point lanes and low-cost permutation primitives (zip/unzip/extract). For lattice-oriented arithmetic (e.g., NTT butterflies, modular reductions, and coefficient reordering), this model enables regular, predictable control flow and memory access, which is desirable for constant-time implementations and cache behavior. In this paper we focus on 32-bit integer lanes and avoid using optional extensions (e.g., SVE/SVE2) to keep portability across mainstream ARMv8-A cores.

A less commonly exploited feature of NEON is its structure load/store family (`vld2q`/`vld3q`/`vld4q` and the corresponding stores), which deinterleaves or interleaves data at the memory interface with no additional ALU cost. For example, `vld3q_s32` reads twelve contiguous 32-bit words and distributes them across three registers in a stride-3 pattern, effectively performing a hardware-level transpose. In Section 3, we show that this mechanism maps naturally to the data-access pattern of radix-3 NTT butterflies, enabling full vectorization of stages whose length falls below the four-lane SIMD width.

We program NEON through the Arm C Language Extensions (*ACLE*) intrinsics (`<arm_neon.h>`), which provide a standardized, compiler-agnostic mapping from C/C++ to A64 vector instructions. Compared to handwritten assembly, intrinsics preserve portability (GCC/Clang/armclang), enable the compiler to schedule around loads/stores, and make constant-time discipline easier to audit; Section 3 details which intrinsics realize each kernel [26,27].

Empirically, prior work reports substantial speedups from NEON vectorization for lattice schemes on ARMv8-A (e.g., Kyber, Saber, NTRU, ML-DSA (Dilithium)), confirming that the above programming model is effective for polynomial transforms and modular arithmetic on contemporary cores [11,28,29]. Among these, Becker et al. [11] introduced interleaved multi-layer butterflies that process up to four radix-2 NTT stages per memory pass, along with a Barrett-multiplication technique for one-known-factor reduction. These advances target power-of-two cyclotomic NTTs; their adaptation to the mixed-radix setting of NCC-Sign is discussed in Section 3.

3. Optimization Methodology

3.1. Design Choice: Montgomery vs. Barrett Reduction

Several ARMv8/NEON implementations of lattice-based schemes adopt Barrett multiplication as a fast modular reduction strategy [11]. In our setting, we choose Montgomery multiplication for three practical reasons. First, NCC-Sign's NTT-friendly trinomial parameter sets use $n = 2^a \cdot 3^b$, requiring mixed-radix stages (notably radix-3) and introducing additional data-movement and shuffle overhead compared to power-of-two cyclotomic NTTs. Our optimization focus is therefore not only the arithmetic cost per residue, but also the SIMD dataflow that keeps lanes local and memory accesses streaming. Second, Montgomery multiplication maps naturally to 32-bit lane arithmetic when combined with ARM's signed high-multiply intrinsics (`vqdmulhq_s32`), enabling an efficient multiply–reduce pipeline without resorting to wider per-lane operations. Barrett, by contrast, requires a multiply-by-reciprocal and one or more shift/correction steps per residue. Third, we integrate a centered reduction step in the pipeline to cap dynamic ranges and prevent edge cases in signed high-multiply operations (Lemma 2), which improves robustness of the SIMD arithmetic path. Overall, while Barrett and Montgomery are both viable modular reduction strategies, our choice is guided by the mixed-radix workload characteristics of NCC-Sign and the resulting SIMD dataflow on ARMv8/NEON. Section 4.6 provides a microbenchmark on Apple M1 Pro confirming this choice empirically: in a NEON radix-2 butterfly, the Montgomery path costs 0.74 cycles per butterfly versus 0.95 for Barrett, a 23% advantage attributable to Montgomery's shorter five-instruction critical path and lower register pressure.

Figure 2 provides an overview of the signing pipeline with optimization targets highlighted. Sections 3.2–3.6 describe the baseline NEON kernels that were part of our initial implementation. Sections 3.7–3.9 then present three additional optimizations developed during revision: radix-2 multi-stage butterfly merging (Section 3.7), stride-3 vectorization of small-len radix-3 stages using NEON structure load/store instructions (Section 3.8), and pointwise Montgomery vectorization with parallel hashing (Section 3.9).

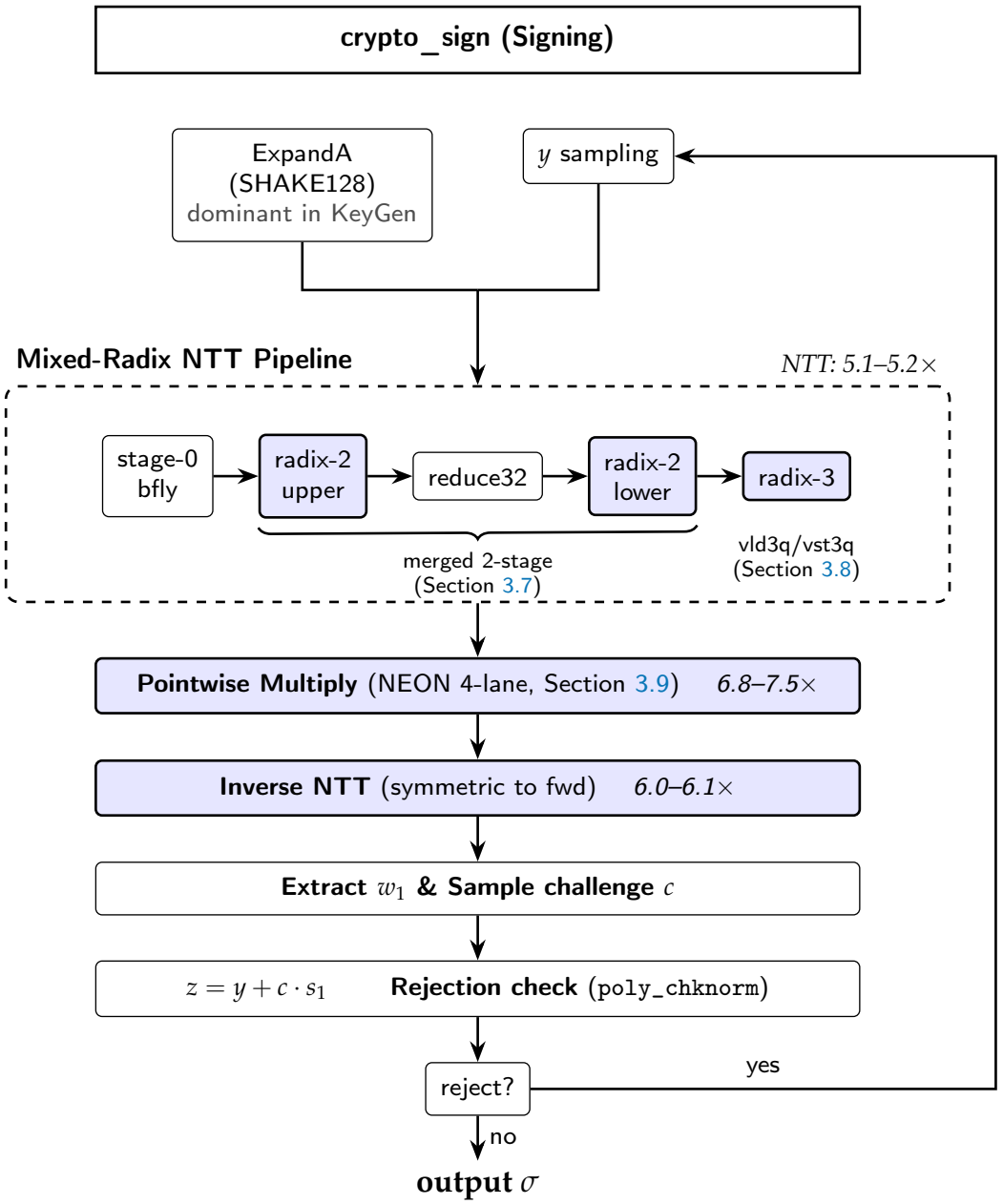

Figure 2. Overview of the NCC-Sign signing pipeline with optimization targets highlighted. Shaded blocks indicate components accelerated by our NEON kernels, with section references and per-kernel speedups on Apple M1 Pro. Unshaded blocks (ExpandA, rejection check) remain unchanged and constitute the primary remaining bottleneck.

3.2. Building Blocks

We present the kernels in NEON intrinsic notation to make the mapping between algorithmic steps and the ARMv8 instruction stream explicit.

3.2.1. Montgomery Reduction Kernel (4-Lane NEON)

We implement Montgomery reduction for signed 32-bit lanes with radix $R = 2^{32}$ and an odd modulus $Q < 2^{31}$. Let Q^{-1} satisfy $Q \cdot Q^{-1} \equiv -1 \pmod{2^{32}}$. For a broadcast twiddle z and a 4-lane vector b, the scalar reference

$$r = \left\lfloor z \cdot b \cdot R^{-1} \right\rfloor \bmod Q$$

is realized lane-wise using the NEON "doubling-high" primitive $\mathrm{H}(x, y) = \lfloor 2xy/2^{32} \rfloor$ and an arithmetic halving: we form $h_1 = \mathrm{H}(z, b)$, a correction $t = (b \cdot Q^{-1}) \bmod 2^{32}$, $m = z \cdot t$, then $h_2 = \mathrm{H}(m, Q)$, and output $r = (h_1 - h_2)/2$. This mapping avoids cross-lane traffic and remains branch-free for secret-dependent values; given $|z|, |Q| < 2^{31}$, it does not trigger saturation in the doubling-high instruction. Our vector kernel emits *five* NEON instructions for *four* results: `vqdmulhq_s32`, `vmulq_s32`, `vmulq_s32`, `vqdmulhq_s32`, `vhsubq_s32`. The algorithmic complexity remains $O(1)$ per residue; the gain is from instruction- and lane-level parallelism.

Correctness and Range Analysis

Let $R = 2^{32}$ and let Q' satisfy $Q \cdot Q' \equiv -1 \pmod{R}$. Our kernel implements a *signed* Montgomery product: given a twiddle z and coefficient b, it returns $r \equiv z\, b\, R^{-1} \pmod{Q}$ with r in a centered range.

Lemma 1 (Montgomery correctness)**.** *Define* $h_1 = \mathrm{H}(z, b)$, $t = (b \cdot Q') \bmod R$, $m = z \cdot t$, $h_2 = \mathrm{H}(m, Q)$, *and* $r = (h_1 - h_2)/2$, *where* $\mathrm{H}(x, y) = \lfloor 2xy/2^{32} \rfloor$ *is the signed doubling-high multiply. Then,* $r \equiv z\, b\, R^{-1} \pmod{Q}$ *and* $|r| < Q$ *whenever* $|z|, |b| \le Q$.

Proof. By construction $h_1 = \lfloor 2zb/R \rfloor$ and $h_2 = \lfloor 2mQ/R \rfloor$. Since $m \equiv z\, b\, Q' \pmod{R}$, we have $mQ \equiv -zb \pmod{R}$, so $zb + mQ \equiv 0 \pmod{R}$; dividing by R gives an integer congruent to $zb\, R^{-1} \pmod{Q}$. The halving subtract $r = (h_1 - h_2)/2$ recovers this quotient in the signed representable range. Under $|z|, |b| \le Q < 2^{31}$, the intermediate products stay within the 64-bit range, and the final result satisfies $|r| < Q$. □

Lemma 2 (No saturation in signed high-multiply)**.** *The ARM instruction* `vqdmulhq_s32` *saturates only when* $\mathrm{H}(-2^{31}, -2^{31})$ *is requested. If a centered reduction step (*`reduce32_vec`*, Section 3.2.2) is applied mid-pipeline so that every input lane satisfies* $|x| < Q < 2^{31}$, *no lane can equal* -2^{31}, *and the saturation condition is never met.*

Proof. The two's-complement corner case arises only for $x = y = -2^{31}$, since $\lfloor 2 \cdot 2^{62}/2^{32} \rfloor = 2^{31}$ exceeds the maximum signed 32-bit value. By construction, `reduce32_vec` maps each lane into $(-Q/2, Q/2] \subset (-2^{22}, 2^{22}]$, which is far from $\pm 2^{31}$. Hence, no NEON lane in the pipeline triggers the saturation path. □

Lemma 1 justifies the algebraic correctness of Algorithm 1, while Lemma 2 formalizes the role of the mid-pipeline centered reduction in preventing edge-case saturation.

Algorithm 1 `montgomery_mul4_sqdmulh` (4-lane, NEON; $R = 2^{32}$)

Require: `b`, `vZ`, `vQ`, `vQINV` *(int32x4_t)*
Ensure: `r` $\equiv z \cdot b \cdot R^{-1} \pmod{Q}$

```
high_ab ← vqdmulhq_s32(vZ, b)          ▷ ⌊2zb/2^32⌋ per lane
b_qinv ← vmulq_s32(b, vQINV)           ▷ (b · Q^-1) mod 2^32
a_times ← vmulq_s32(vZ, b_qinv)
high_corr ← vqdmulhq_s32(a_times, vQ)  ▷ ⌊2(a_timesQ)/2^32⌋
r ← vhsubq_s32(high_ab, high_corr)     ▷ arith. halving of (h1 − h2)
return r
```

3.2.2. Centered Modular Reduction (`reduce32_vec`)

We normalize signed 32-bit coefficients using a vectorized Barrett-style step tuned for a modulus Q close to 2^{23} (e.g., $Q < 2^{23}$). For a lane value x, we compute

$$t = \left\lfloor \frac{x + 2^{22}}{2^{23}} \right\rfloor, \qquad r = x - t \cdot Q.$$

This realizes a centered reduction that returns r in a narrow signed interval around 0. The operation is branch-free for secret-dependent values; it uses a single right-shift to approximate division and one multiply by Q for the correction. Per four residues the kernel issues four vector ALU instructions: `vaddq_s32`, `vshrq_n_s32`, `vmulq_s32`, `vsubq_s32`—i.e., exactly 1.0 instruction per result.

3.3. Radix-2 Butterfly Pipeline

The forward NTT begins with the *stage-0 butterfly*, which consumes the first twiddle $z =$ `zetas[0]` and processes the top split (a, b) with $n_2 = N/2$. Each 4-lane iteration computes $t = \text{Mont}(z \cdot b)$ and writes $(a' = a + t,\ b' = a + b - t)$. Note that the stage-0 output differs from the standard Cooley–Tukey butterfly $(a + t,\ a - t)$ used in negacyclic NTTs. This is because the trinomial $\varphi(X) = X^n - X^{n/2} + 1$ factors over two roots ζ and $1 - \zeta$ of $t^2 - t + 1 = 0$, yielding the CRT split $a' = a + \zeta b$ and $b' = a + (1 - \zeta)b = a + b - \zeta b$. The cost is 8 vector ALU ops per 8 residues (1.0/residue).

After stage 0, two radix-2 kernels handle the remaining lengths in a Cooley–Tukey pattern. The `radix2_upper_neon` kernel processes lengths *len* from $N/4$ down to (but not including) `radix2_redlen_ntt`; the `radix2_lower_neon` kernel continues from `radix2_redlen_ntt` down to $3 \cdot$ `radix3_len`, after which the mixed radix-3 stage takes over. For each (a, b) pair at a given *len*, we compute

$$a' = a + t, \qquad b' = a - t, \qquad t = \text{Mont}(z \cdot b),$$

using the 4-lane Montgomery kernel (Algorithm 1). Per 4-lane pair (eight residues), we perform two loads, one 5-op Montgomery kernel, one `vaddq_s32`, one `vsubq_s32`, and two stores; this totals 7 vector ALU ops, i.e., $7/8 = 0.875$ ALU ops per residue.

Between the upper and lower kernels we insert a mid-pipeline centered reduction sweep (Section 3.2.2; see Figure 3) to cap coefficient growth and maintain the non-saturating input conditions for `vqdmulhq_s32` (Lemma 2). The lower kernel (`radix2_lower_neon`) follows the same structure as Algorithm 2 with a different `len` range; we omit a separate listing. All arithmetic-kernel code paths are branch-free with respect to secret-dependent data.

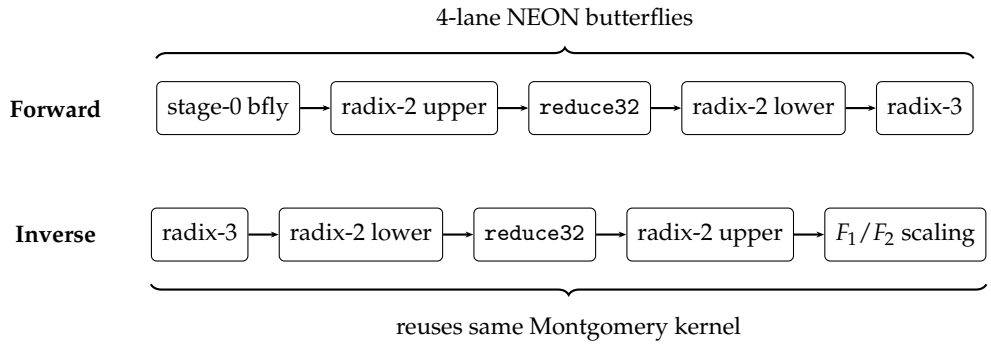

Figure 3. Forward and inverse NTT pipelines with mid-pipeline vector `reduce32` sweep (sign1/5 modes).

3.4. Mixed Radix-3 Pipeline

The mixed-radix parameter sets of NCC-Sign ($n = 2^a \cdot 3^b$) require radix-3 stages after the radix-2 cascade. For a triplet (a_0, a_1, a_2) at a given stride *len*, with stage twiddles z_1, z_2, we compute

$$t_1 = z_1 \cdot a_1,\ \ t_2 = z_2 \cdot a_2, \quad u_1 = w \cdot t_1,\ \ u_2 = w^2 \cdot t_2,$$

and output

$$a_0' = a_0 + (t_1 + t_2), \qquad a_1' = a_0 + (u_1 + u_2), \qquad a_2' = a_0 - \big((t_1 + t_2) + (u_1 + u_2)\big),$$

where all products are Montgomery products modulo Q with radix $R = 2^{32}$ (Algorithm 1). The vector kernel maps these operations lane-wise; no cross-lane permutation is needed. The 4-lane NEON implementation of the forward mixed radix-3 stage is described in Algorithm 3.

Algorithm 2 `radix2_upper_neon` (4-lane, NEON; $len : N/4 \rightarrow$ `radix2_redlen_ntt` + 1)

Require: `Out`, `zetas`, `pk`, `vQ`, `vQINV`

for $len \leftarrow N/4$ **down to** `radix2_redlen_ntt` $+ 1$ **by** $len \leftarrow len/2$ **do**
  for $start \leftarrow 0$ **to** $N - 1$ **step** $2 \cdot len$ **do**
    $z \leftarrow$ `zetas[ *pk++ ];  vZ` $\leftarrow$ `vdupq_n_s32(`z`)`
    $j \leftarrow start$; $j\_end \leftarrow start + len$
    while $j + 4 \leq j\_end$ **do**
      `a` $\leftarrow$ `vld1q_s32(&Out[j])`
      `b` $\leftarrow$ `vld1q_s32(&Out[j + len])`
      `t` $\leftarrow$ `montgomery_mul4_sqdmulh(b, vZ, vQ, vQINV)`
      `vst1q_s32(&Out[j + len], vsubq_s32(a, t))` ▷ $b' = a - t$
      `vst1q_s32(&Out[j], vaddq_s32(a, t))` ▷ $a' = a + t$
      $j \leftarrow j + 4$
    end while
    while $j < j\_end$ **do** ▷ scalar tail
      standard scalar butterfly with `montgomery_reduce`
      $j \leftarrow j + 1$
    end while
  end for
end for

Algorithm 3 `radix3_forward_mixed_neon` (4-lane, NEON; sign1/5)

Require: `Out`, `zetas`, `pk`, `vQ`, `vQINV`

`vW` $\leftarrow$ `vdupq_n_s32(Wmont);  vW2` $\leftarrow$ `vdupq_n_s32(W2mont)`
for $len \leftarrow$ `radix3_len` **down to** 1 **by** $len \leftarrow len/3$ **do**
  for $start \leftarrow 0$ **to** $N - 1$ **step** $3 \cdot len$ **do**
    $z_1 \leftarrow$ `zetas[ *pk++ ];` $z_2 \leftarrow$ `zetas[ *pk++ ]`
    `vZ1` $\leftarrow$ `vdupq_n_s32(`z_1`);  vZ2` $\leftarrow$ `vdupq_n_s32(`z_2`)`
    $j \leftarrow start$; $j\_end \leftarrow start + len$
    while $j + 4 \leq j\_end$ **do**
      `a0` $\leftarrow$ `vld1q_s32(&Out[j])`
      `a1` $\leftarrow$ `vld1q_s32(&Out[j + len])`
      `a2` $\leftarrow$ `vld1q_s32(&Out[j + 2*len])`
      `t1` $\leftarrow$ `montgomery_mul4_sqdmulh(a1, vZ1, vQ, vQINV)`
      `t2` $\leftarrow$ `montgomery_mul4_sqdmulh(a2, vZ2, vQ, vQINV)`
      `t12`$\leftarrow$ `vaddq_s32(t1, t2)`
      `u1` $\leftarrow$ `montgomery_mul4_sqdmulh(t1, vW, vQ, vQINV)`
      `u2` $\leftarrow$ `montgomery_mul4_sqdmulh(t2, vW2, vQ, vQINV)`
      `t34`$\leftarrow$ `vaddq_s32(u1, u2)`
      `a0n`$\leftarrow$ `vaddq_s32(a0, t12)`
      `a1n`$\leftarrow$ `vaddq_s32(a0, t34)`
      `a2n`$\leftarrow$ `vsubq_s32(a0, vaddq_s32(t12, t34))`
      `vst1q_s32(&Out[j], a0n)`
      `vst1q_s32(&Out[j + len], a1n)`
      `vst1q_s32(&Out[j + 2*len], a2n)`
      $j \leftarrow j + 4$
    end while
    while $j < j\_end$ **do** ▷ scalar tail (bit-identical to reference)
      standard scalar radix-3 butterfly
      $j \leftarrow j + 1$
    end while
  end for
end for

Cost

Per 4-lane triplet (12 residues): three loads, four calls to the Montgomery kernel ($4 \times 5 = 20$ vector ALU ops), four add/sub, and three stores—overall 24 vector ALU ops, or about 2.0 ops per residue. Lengths $len \in \{3, 1\}$ commonly leave a scalar tail; the vector path handles any full 4-lane chunks.

3.5. Inverse Path

The inverse pipeline mirrors the forward path and stays lane-local. For sign1/5 we reuse the same 4-lane Montgomery primitive and compose three steps in reverse order: (i) an in-place inverse butterfly (`inv_bfly4_store`), (ii) a mixed radix-3 inverse kernel (`radix3_inverse_mixed_neon`), and (iii) a final fixed-twiddle scaling with (F_1, F_2). For sign3, the driver keeps a reference-compatible dispatch for part of the inverse path; reported speedups therefore reflect both optimized kernels and unchanged reference components. The inverse kernels mirror their forward counterparts; the principal differences are reversed stage order and the use of inverse twiddles. We omit separate algorithm listings as the kernel structure is symmetric to the forward path; see the public repository for full details.

Cost (Summary)

`inv_bfly4_store`: about 7 vector ALU ops per 8 residues (≈0.875/residue). Radix-3 inverse triplet: about 25 ALU ops per 12 residues (≈2.08/residue). Final scaling: about 18 ALU ops per 8 residues (≈2.25/residue). All described arithmetic kernels are data-independent with respect to secret-dependent control flow.

3.6. Integration Notes

3.6.1. NEON Intrinsics Summary

Table 5 lists the ARMv8 NEON intrinsics used across all kernels, together with their roles. All intrinsics operate on `int32x4_t` (four 32-bit signed lanes).

Table 5. Summary of ARMv8 NEON intrinsics used in our kernels.

Intrinsic	Per-Lane Semantics	Used in
`vqdmulhq_s32`	Signed doubling high-multiply	Montgomery kernel (Section 3.2)
`vmulq_s32`	Lane-wise multiply (low 32 bits)	Montgomery/correction (Section 3.2)
`vhsubq_s32`	Halving subtract $(a - b)/2$	Montgomery output (Section 3.2)
`vaddq_s32`	Lane-wise add	Butterflies/reduce32 (Sections 3.2–3.4)
`vsubq_s32`	Lane-wise subtract	Butterflies/reduce32 (Sections 3.2–3.4)
`vshrq_n_s32`	Arithmetic right shift	reduce32 (Section 3.2)
`vdupq_n_s32`	Broadcast scalar to all lanes	Twiddles, Q, Q^{-1} (Sections 3.2–3.4)
`vld1q_s32`	Contiguous 128-bit load	Streaming loads (Sections 3.3–3.6)
`vst1q_s32`	Contiguous 128-bit store	In-place writes (Sections 3.3–3.6)
`vld3q_s32`	Stride-3 deinterleave (12 words → 3 regs)	Radix-3 len = 1 (Section 3.8)
`vst3q_s32`	Stride-3 interleave (3 regs → 12 words)	Radix-3 len = 1 (Section 3.8)
`vld2q_s32`	Stride-2 deinterleave (8 words → 2 regs)	Twiddle pair load (Section 3.8)
`vld1q_lane_s32`	Single-lane insert from memory	Pointwise gather (Section 3.9)

3.6.2. Normalization Strategy

Instead of normalizing after every butterfly, we apply a *mid-pipeline* centered reduction using `reduce32_vec` (Algorithm 4) between the radix-2 upper and lower spans (and symmetrically in the inverse path). This "reduce-late" policy minimizes arithmetic while preserving numerical safety. Without reduction, magnitude can grow roughly by a factor ≤ 2 per level; a single $O(N)$ sweep with 1.0 vector ALU op per residue caps all inputs well within the non-saturating domain (Lemma 2). Final normalization is deferred to the inverse-path epilogue, where scaling with (F_1, F_2) canonicalizes outputs.

Algorithm 4 `reduce32_vec` (4-lane, NEON; centered reduction)

Require: `x` *(int32x4_t)*, constants Q, $C = 2^{22}$
Ensure: `r` $\approx x \bmod Q$ in a centered interval

```
add ← vdupq_n_s32(C)        ▷ broadcast 2^22
qv ← vdupq_n_s32(Q)         ▷ broadcast Q
t ← vaddq_s32(x, add)       ▷ x + 2^22
t ← vshrq_n_s32(t, 23)      ▷ t = ⌊(x + 2^22)/2^23⌋
tq ← vmulq_s32(t, qv)       ▷ t · Q (low 32 bits)
r ← vsubq_s32(x, tq)        ▷ centered subtraction
return r
```

3.6.3. Memory and Throughput

We operate *in place* with two contiguous loads and two stores per 4-lane butterfly (stage-0/radix-2), and three loads/stores per radix-3 triplet. Twiddles are broadcast once per `start`-block (`vdupq_n_s32`); Q and Q^{-1} are hoisted out of the loops using `restrict` pointers. Accesses over (a, b) halves at stride *len* preserve streaming patterns, and scalar tails handle non-multiples of four without divergence. For best throughput, `Out` should be 16-byte aligned; `vld1q_s32`/`vst1q_s32` still function correctly if unaligned, with minor penalties.

3.7. Radix-2 Multi-Stage Butterfly Merging

In the baseline implementation, each radix-2 NTT stage performs a full pass over the coefficient array: loading every element, computing the butterfly, and storing the result before the next stage begins. For k consecutive radix-2 stages this incurs k round trips to memory, and on both Apple M1 Pro and Cortex-A72, the load/store cost dominates the arithmetic cost of the butterfly itself.

Becker et al. [11] showed that for power-of-two NTTs, multiple radix-2 layers could be interleaved within NEON registers so that intermediate results are never written back to memory. We adapted this idea to the radix-2 span of NCC-Sign's mixed-radix NTT pipeline, merging two consecutive stages into a single memory pass. The proposed two-stage merged radix-2 forward NEON kernel is given in Algorithm 5.

Concretely, let len_o and $len_i = len_o/2$ denote the outer and inner stage lengths. For each group of four coefficient positions $\{j,\ j + len_i,\ j + len_o,\ j + len_o + len_i\}$, the merged kernel:

1. Loads four values a, b, c, d with four `vld1q_s32` instructions;
2. Performs the outer butterfly in registers: $t_1 = \mathrm{Mont}(\zeta_{\mathrm{outer}}, c)$, $u = a + t_1$, $v = a - t_1$, and likewise for (b, d);
3. Immediately performs the inner butterfly on the results: $t_2 = \mathrm{Mont}(\zeta_{\mathrm{inner},0}, w)$, $a' = u + t_2$, $c' = u - t_2$, and likewise with $\zeta_{\mathrm{inner},1}$;
4. Stores the four results with four `vst1q_s32` instructions.

The net effect is that each coefficient is loaded and stored once instead of twice, halving the memory traffic for the merged stages. The twiddle factors for both stages are consumed in sequence: outer twiddles occupy indices k_{start} through $k_{\mathrm{start}} + num\_outer - 1$, and inner twiddles follow immediately. When the total number of radix-2 stages in a span is odd, the last stage falls back to the unmerged single-stage kernel.

We apply this merging to both the forward and inverse radix-2 paths. In the inverse direction the butterfly order is reversed (inner before outer), and the twiddle consumption order is adjusted accordingly. The mid-pipeline `reduce32` sweep (Section 3.2.2) is retained at the boundary between the upper and lower radix-2 spans to prevent coefficient growth from reaching the saturation threshold of `vqdmulhq_s32` (Lemma 2).

Algorithm 5 `radix2_merged2_fwd_neon` (2-stage merged, 4-lane NEON)

Require: `Out`, `zetas`, `pk`, `vQ`, `vQINV`, len_o (outer stage length)
Ensure: Two consecutive radix-2 NTT stages in one memory pass

$len_i \leftarrow len_o/2$
for $start \leftarrow 0$ **to** $N-1$ **step** $2 \cdot len_o$ **do**
  $z_o \leftarrow$ `zetas[*pk++];` $vZ_o \leftarrow$ `vdupq_n_s32`(z_o)
  $z_{i0} \leftarrow$ `zetas[*pk++];` $vZ_{i0} \leftarrow$ `vdupq_n_s32`(z_{i0})
  $z_{i1} \leftarrow$ `zetas[*pk++];` $vZ_{i1} \leftarrow$ `vdupq_n_s32`(z_{i1})
  for $j \leftarrow start$ **to** $start + len_i - 1$ **step** 4 **do**
    $a \leftarrow$ `vld1q_s32`$(\&Out[j])$
    $b \leftarrow$ `vld1q_s32`$(\&Out[j + len_i])$
    $c \leftarrow$ `vld1q_s32`$(\&Out[j + len_o])$
    $d \leftarrow$ `vld1q_s32`$(\&Out[j + len_o + len_i])$
    ▷ *Outer butterfly (stage* len_o*)*
    $t_{ac} \leftarrow$ `montgomery_mul4`$(c, vZ_o,$ `vQ, vQINV`$)$
    $t_{bd} \leftarrow$ `montgomery_mul4`$(d, vZ_o,$ `vQ, vQINV`$)$
    $u \leftarrow$ `vaddq_s32`(a, t_{ac}); $v \leftarrow$ `vsubq_s32`(a, t_{ac})
    $w \leftarrow$ `vaddq_s32`(b, t_{bd}); $x \leftarrow$ `vsubq_s32`(b, t_{bd})
    ▷ *Inner butterfly (stage* len_i*)*
    $t_w \leftarrow$ `montgomery_mul4`$(w, vZ_{i0},$ `vQ, vQINV`$)$
    $t_x \leftarrow$ `montgomery_mul4`$(x, vZ_{i1},$ `vQ, vQINV`$)$
    `vst1q_s32`$(\&Out[j],$ `vaddq_s32`$(u, t_w))$
    `vst1q_s32`$(\&Out[j + len_i],$ `vsubq_s32`$(u, t_w))$
    `vst1q_s32`$(\&Out[j + len_o],$ `vaddq_s32`$(v, t_x))$
    `vst1q_s32`$(\&Out[j + len_o + len_i],$ `vsubq_s32`$(v, t_x))$
  end for
end for

We note that extending this approach across the radix-2/radix-3 boundary was also investigated: merging the last radix-2 stage with the first radix-3 stage would avoid one additional memory round trip. However, the stride-3 access pattern of the radix-3 stage is incompatible with the contiguous four-lane loads used in the radix-2 path, and the gather/scatter overhead required to reconcile the two patterns exceeds the load/store saving. We therefore kept the radix-2 and radix-3 pipelines separate.

3.8. Stride-3 Vectorization of Radix-3 Stages

In the baseline radix-3 kernel (Section 3.4), the inner loop processes triplets (a_0, a_1, a_2) in chunks of four lanes using `vld1q_s32` and the Montgomery kernel. When the stage length *len* is four or greater, this works well. However, the final radix-3 stages of NCC-Sign-1 and NCC-Sign-5 operate at $len = 3$ and $len = 1$, both below the NEON four-lane width. In the baseline code these iterations fall through to a scalar tail that executes the Montgomery reduction one coefficient at a time, forfeiting the SIMD advantage precisely where the radix-3 cost concentrates.

A stage-merging approach analogous to Section 3.7 does not help here: the small array segments already reside in L1 cache, so eliminating one load/store round trip has negligible effect. The bottleneck is not memory traffic but the per-coefficient cost of the scalar Montgomery reduction, which processes one residue at a time instead of four.

We resolve this with a data-layout transformation that maps the radix-3 access pattern onto NEON structure load/store instructions.

3.8.1. $len = 1$ (Four-Block Gather)

Figure 4 illustrates the data-layout transformation for this case. When $len = 1$ each radix-3 block spans three contiguous coefficients, and successive blocks are laid out consecutively in memory. The NEON instruction `vld3q_s32` reads twelve contiguous 32-bit

words and distributes them into three registers with a hardware stride-3 deinterleave at no additional ALU cost:

$$\texttt{vld3q\_s32}(\&Out[start]) \text{ produces } \begin{aligned} val[0] &= (Out[0],\ Out[3],\ Out[6],\ Out[9]),\\ val[1] &= (Out[1],\ Out[4],\ Out[7],\ Out[10]),\\ val[2] &= (Out[2],\ Out[5],\ Out[8],\ Out[11]), \end{aligned}$$

which groups the first, second, and third elements of four independent triplets into separate NEON registers. Because each block uses different twiddle factors, we load these as vectors as well: `vld2q_s32` on the twiddle array separates the (ζ_1, ζ_2) pairs for four blocks in one instruction. The four-lane Montgomery kernel then processes all four triplets simultaneously. After the butterfly, `vst3q_s32` performs the inverse interleave and writes the twelve results back. The loop advances by four blocks (twelve coefficients) per iteration; a scalar tail handles any remainder. To fully vectorize the len = 1 radix-3 stage, we reorganize the data layout as shown in Algorithm 6.

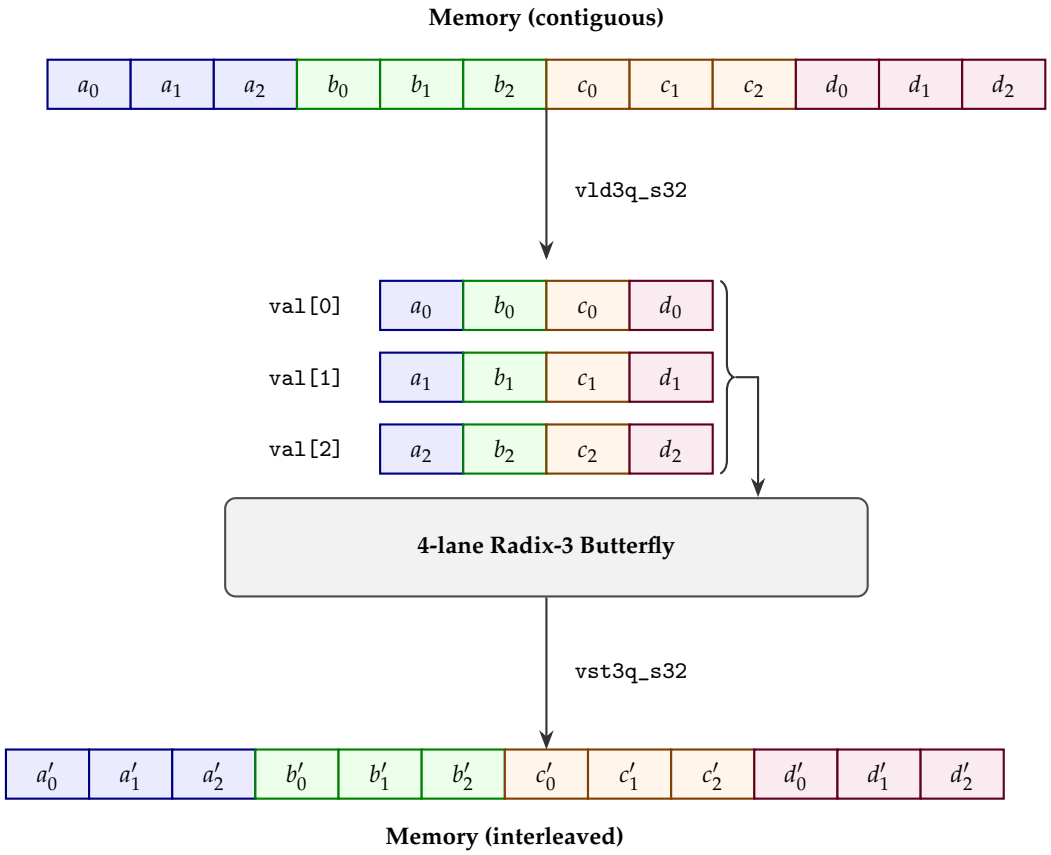

Figure 4. Data-layout transformation for the $len = 1$ radix-3 vectorization. `vld3q_s32` deinterleaves twelve contiguous coefficients from four independent triplets into three NEON registers, enabling a four-lane parallel radix-3 butterfly. `vst3q_s32` restores the interleaved layout.

3.8.2. $len = 3$ (Three-Wide SoA)

When $len = 3$ a single block spans nine coefficients. Within one block the three sub-arrays $Out[start \ldots start + 2]$, $Out[start + 3 \ldots start + 5]$, $Out[start + 6 \ldots start + 8]$ are each contiguous. We load each sub-array into a NEON register via `vld1q_s32`, obtaining three lanes of useful data per register (the fourth lane is unused). The same twiddle factor applies to all three lanes, so a single `vdupq_n_s32` broadcast suffices. After the radix-3 butterfly, we store only the lower three elements of each result register. Because the twiddle is shared within a block, no gather/scatter is needed.

NCC-Sign-3 uses $n = 1536 = 2^9 \times 3$, whose single radix-3 stage operates at $len \geq 4$ and is therefore already handled by the baseline four-lane path; no stride-3 transformation is applied.

Algorithm 6 `radix3_stride3_len1_neon` (`vld3q`, 4-block parallel)

Require: `Out`, `zetas`, `pk`, `vQ`, `vQINV`, `vW`, `vW2`
Ensure: Radix-3 butterfly on four consecutive $len = 1$ triplets
for $start \leftarrow 0$ **to** $N - 1$ **step** 12 **do**
  $data \leftarrow$ `vld3q_s32`(&$Out[start]$) ▷ stride-3 deinterleave
  $a_0 \leftarrow data.val[0]$; $a_1 \leftarrow data.val[1]$; $a_2 \leftarrow data.val[2]$
  $tw \leftarrow$ `vld2q_s32`(&`zetas`[`*pk`]) ▷ load $(\zeta_1, \zeta_2) \times 4$ blocks
  $vZ_1 \leftarrow tw.val[0]$; $vZ_2 \leftarrow tw.val[1]$
  `*pk` $\leftarrow$ `*pk` $+ 8$
  ▷ *Four-lane radix-3 butterfly*
  $t_1 \leftarrow$ `montgomery_mul4`(a_1, vZ_1, `vQ`, `vQINV`)
  $t_2 \leftarrow$ `montgomery_mul4`(a_2, vZ_2, `vQ`, `vQINV`)
  $t_{12} \leftarrow$ `vaddq_s32`(t_1, t_2)
  $u_1 \leftarrow$ `montgomery_mul4`(t_1, `vW`, `vQ`, `vQINV`)
  $u_2 \leftarrow$ `montgomery_mul4`(t_2, `vW2`, `vQ`, `vQINV`)
  $t_{34} \leftarrow$ `vaddq_s32`(u_1, u_2)
  $result.val[0] \leftarrow$ `vaddq_s32`(a_0, t_{12})
  $result.val[1] \leftarrow$ `vaddq_s32`(a_0, t_{34})
  $result.val[2] \leftarrow$ `vsubq_s32`(a_0, `vaddq_s32`(t_{12}, t_{34}))
  `vst3q_s32`(&$Out[start]$, $result$) ▷ stride-3 interleave
end for

3.8.3. Effect

The stride-3 vectorization converts the radix-3 path from fully scalar to fully vectorized execution for NCC-Sign-1 and NCC-Sign-5. The resulting end-to-end impact is most visible in verification, where the inverse NTT is dominated by these small-len radix-3 stages; quantitative results are presented in Section 4.3. We emphasize that the key enabler is the observation that NEON's structure load/store instructions match the stride-3 memory pattern of radix-3 butterflies—an optimization opportunity specific to mixed-radix NTTs that does not arise in the power-of-two transforms common in ML-KEM and ML-DSA.

3.9. Pointwise Multiplication and Parallel Hashing

Two further optimizations target non-NTT components of the signing pipeline.

3.9.1. Pointwise Montgomery Multiplication

In the baseline code, the coefficient-wise multiplication of two NTT-domain polynomials invokes the scalar `montgomery_reduce` function in a simple loop. Because each coefficient pair is independent, the loop maps directly to the existing four-lane Montgomery kernel (Section 3.2.1): four coefficient pairs are loaded with `vld1q_s32`, multiplied via `montgomery_mul4_sqdmulh`, and stored back. For NCC-Sign-1 and NCC-Sign-5 (modes 1/5), whose pointwise operation is a straightforward element-wise product, N is a multiple of four and no scalar tail is needed. For NCC-Sign-3 (mode 3), the pointwise step involves a base multiplication over degree-2 residues with stride-6 memory access; we batch four such blocks into a single NEON pass using explicit gather (`vld1q_lane_s32`) and scatter helpers, performing nine Montgomery multiplications per batch in four-lane vectors. The performance impact of this vectorization is quantified in Section 4.3.

3.9.2. Parallel Hashing (`fips202x2`)

Sampling and hashing operations based on SHAKE128/256 account for over 50% of key-generation cycles (Section 4), motivating the integration of a two-way NEON-parallel Keccak implementation. We adopt the `fips202x2` module from the `neon-ntt` repository, which interleaves two independent SHAKE instances on a single pair of NEON register files by executing the KeccakF1600 permutation on both states simultaneously.

Within NCC-Sign, we apply `fips202x2` to the secret-key sampling phase of key generation, where two independent η-bounded polynomials (s_1 and s_2) are derived from the same seed with different nonces. The two SHAKE256 streams are absorbed and squeezed in parallel, and each stream is independently passed through the rejection-sampling loop (`rej_eta`).

The achievable speedup is inherently limited by the algorithm structure. NCC-Sign uses a single polynomial ring (as opposed to the module structure of ML-DSA), so the dominant hashing cost—the `ExpandA` call that derives the public matrix element from a SHAKE128 stream—is a single sequential invocation that cannot be paired. The parallel path therefore covers only the s_1/s_2 sampling, which represents roughly 10–15% of total sampling cost. The measured end-to-end improvement from `fips202x2` is accordingly modest (1–2% for key generation). We include this optimization for completeness and note that it would yield substantially larger gains in module-lattice schemes where `ExpandA` generates $k \times l$ independent matrix entries.

4. Performance Evaluation

4.1. Benchmarking Environment

We measured the end-to-end cost of three core signature operations—key generation (`crypto_sign_keypair`), signing (`crypto_sign`), and verification (`crypto_sign_open`)—on two architecturally distinct ARMv8 platforms.

4.1.1. Platform 1: Apple M1 Pro

A MacBook Pro with Apple M1 Pro (eight performance + two efficiency cores), 16 GB RAM, macOS Sequoia, compiled with `Apple clang 14.0.3` (target: `arm64-apple-darwin`) using `-O3 -fomit-frame-pointer` and the same warning flags as the baseline. Cycle counting used Apple silicon's PMU core-cycle counter through the private `kperf`/`kpc` interface (`kpc_get_thread_counters()`), following the open-source M1/M2 cycle-counter library by Nguyen [30] and established practice for Apple silicon benchmarking [31–33]. The benchmark thread was pinned to a high-performance core via `QOS_CLASS_USER_INTERACTIVE`.

4.1.2. Platform 2: Arm Cortex-A72 (Raspberry Pi 4)

A Raspberry Pi 4 Model B with a quad-core Cortex-A72 at 1.8 GHz, 8 GB RAM, running 64-bit Raspberry Pi OS (Linux aarch64), compiled with GCC 14 using the same `-O3` flag. Cycle counting used the Linux `perf_event_open` interface configured for `PERF_COUNT_HW_CPU_CYCLES` with kernel-mode exclusion, which does not require root privileges when `perf_event_paranoid` ≤ 2. The Cortex-A72 represents a widely deployed embedded ARMv8 core and serves as the standard benchmarking platform in prior PQC NEON work [11,28].

The two platforms differ substantially in microarchitecture: the M1 Pro features a wide out-of-order pipeline with high instruction-level parallelism, whereas the Cortex-A72 has a narrower pipeline. Evaluating on both allows us to distinguish optimization effects that are robust across microarchitectures from those specific to a particular core design.

4.1.3. Methodology

Each benchmark binary (`KpqC_bench`) executed 10,000 iterations of the target operation and reported the arithmetic mean. We ran each binary 20 times and report the minimum (best) of the 20 runs as the primary metric, as this most closely reflects the true algorithmic cost by minimizing interference from OS scheduling and DVFS transients. The mean and standard deviation of the 20 runs are reported alongside to characterize measurement stability. Because signing involves a rejection loop whose iteration count varies stochastically (expected attempts: ∼1.93 for Sign-1, ∼2.76 for Sign-3, ∼2.32 for Sign-5), the Sign metric inherently exhibits higher variance than the deterministic KeyGen and Verify operations.

4.1.4. Baselines

We adopted the NCC-Sign implementations from *KPQClean* and its ver. 2 branch [34,35], which follow the PQClean philosophy of clean, dependency-free baselines [36]. Both the baseline and optimized variants were compiled with the *identical* toolchain, optimization level (`-O3`), and flags on each platform; the only difference was the activation of ARMv8/NEON kernels in the optimized build.

4.1.5. Parameter Sets

We benchmarked the **1**, **3**, and **5** parameter sets, corresponding to the build targets `NCC-Sign1`, `NCC-Sign3`, and `NCC-Sign5` [20].

4.1.6. Public Release

All artifacts described above—including compile-time switches to enable or disable NEON kernels, cycle-counting adaptations for both platforms, the dudect test harness, and the Montgomery-versus-Barrett microbenchmark—are included in the public repository introduced in Section 1. Functional correctness was verified by the built-in test routines, which confirmed that signatures generated by the optimized path were accepted by the corresponding verification routine.

4.2. End-to-End Results

Table 6 reports cycle counts on Apple M1 Pro for NCC-Sign-1/3/5, measured as the best of 20 runs with 10,000 iterations each.

Table 6. Cycle counts on Apple M1 Pro (best of 20 runs × 10,000 iterations). The primary metric is the best (minimum) of 20 independent runs. Measurement stability (mean ± std) is reported separately in Table 7.

Param	KeyGen		Sign		Verify	
	Baseline	**Optimized**	**Baseline**	**Optimized**	**Baseline**	**Optimized**
NCC-Sign-1	165,429	117,310 (1.41×)	359,174	157,270 (2.28×)	217,437	106,590 (2.04×)
NCC-Sign-3	221,737	161,638 (1.37×)	478,802	228,157 (2.10×)	285,469	152,556 (1.87×)
NCC-Sign-5	335,168	235,254 (1.42×)	746,461	318,339 (2.34×)	445,271	210,566 (2.11×)
Geo-mean speedup	1.40×		2.24×		2.01×	

Table 8 presents the corresponding results on Arm Cortex-A72 (Raspberry Pi 4).

Table 7. Measurement stability of the optimized build on Apple M1 Pro: mean and standard deviation over 20 independent runs. KeyGen and Verify are deterministic and exhibit low variance (KeyGen: std $<$ 0.5% of mean; Verify: std $<$ 1.5% of mean). Sign includes rejection sampling, producing inherently high variance; the best (minimum) value reported in Table 6 reflects the per-attempt cost most accurately.

Param	KeyGen (Mean ± Std)	Sign (Mean ± Std)	Verify (Mean ± Std)
NCC-Sign-1	117,532 ± 114	218,582 ± 69,366	107,786 ± 850
NCC-Sign-3	162,023 ± 463	490,279 ± 373,661	153,812 ± 814
NCC-Sign-5	235,770 ± 533	538,260 ± 223,511	214,022 ± 1810

Table 8. Cycle counts on Arm Cortex-A72 (best of 20 runs × 10,000 iterations). The primary metric is the best (minimum) of 20 independent runs. Measurement stability is reported in Table 9.

Param	KeyGen		Sign		Verify	
	Baseline	Optimized	Baseline	Optimized	Baseline	Optimized
NCC-Sign-1	390,795	312,534 (1.25×)	930,619	602,042 (1.55×)	591,989	406,379 (1.46×)
NCC-Sign-3	510,523	416,284 (1.23×)	1,177,012	787,940 (1.49×)	750,000	539,245 (1.39×)
NCC-Sign-5	790,094	625,889 (1.26×)	1,920,755	1,219,615 (1.57×)	1,206,710	823,245 (1.47×)
Geo-mean speedup	1.25×		1.54×		1.44×	

Table 9. Measurement stability of the optimized build on Arm Cortex-A72: mean and standard deviation over 20 independent runs. KeyGen and Verify are deterministic and exhibit negligible variance (KeyGen: std $<$ 0.5% of mean; Verify: std $<$ 1.5% of mean).

Param	KeyGen (Mean ± Std)	Sign (Mean ± Std)	Verify (Mean ± Std)
NCC-Sign-1	313,259 ± 311	838,782 ± 314,395	416,552 ± 5400
NCC-Sign-3	417,043 ± 720	1,436,784 ± 618,473	546,753 ± 4119
NCC-Sign-5	627,231 ± 1248	2,028,078 ± 961,082	831,527 ± 4003

Discussion

Figure 5 visualizes the per-operation improvements on M1 Pro. On Apple M1 Pro, the NEON-optimized build achieves geometric-mean speedups of 1.40× for key generation, 2.24× for signing, and 2.01× for verification across the three parameter sets. The signing and verification paths benefit the most, consistent with these operations spending the majority of their cycles in NTT/INTT and pointwise multiplication (quantified in Section 4.3). Key generation improves more modestly because its dominant cost is Keccak-based sampling (`ExpandA`), which is not targeted by our NEON kernels.

On Cortex-A72 the same optimizations yield lower but consistent speedups: 1.25× for key generation, 1.54× for signing, and 1.44× for verification. The reduced gains reflect the narrower execution pipeline of the Cortex-A72, where the scalar baseline already achieves relatively higher utilization of the available issue width. Nevertheless, the optimizations are beneficial on both microarchitectures, confirming that the NEON kernels are portable across distinct ARMv8-A implementations.

Absolute cycle counts on Cortex-A72 are 2.6–3.9× higher than on M1 Pro for the same operation, with the ratio being larger for NTT-intensive operations (Sign, Verify) than for sampling-dominated ones (KeyGen). This is consistent with the M1 Pro's wider pipeline extracting more instruction-level parallelism from the vectorized NTT loops.

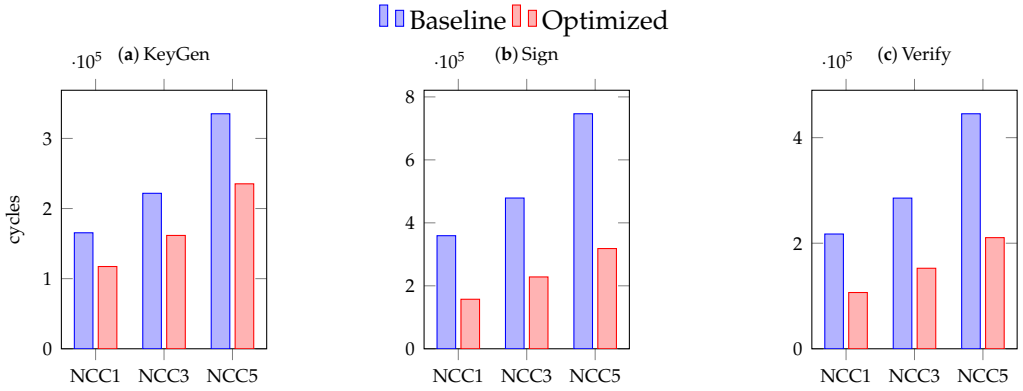

Figure 5. Per-operation cycle counts on Apple M1 Pro: baseline (**left**) vs. optimized (**right**) across NCC-Sign-1/3/5.

4.3. Function-Level Profiling

To understand where the speedups originate, we profile the optimized build on Apple M1 Pro by measuring each arithmetic kernel in isolation (1000 iterations, median cycles). Table 10 reports the per-kernel speedups for the three operations targeted by our NEON optimization.

Table 10. Per-kernel speedups (clean baseline → NEON optimized) on Apple M1 Pro. Median of 1000 independent calls.

Kernel	Sign-1	Sign-3	Sign-5
NTT	5.11×	3.59×	5.21×
INTT	6.02×	4.70×	6.05×
BaseMul	6.84×	4.19×	7.52×

The NTT and INTT kernels achieve 3.6–6.1× speedups, reflecting the combined effect of four-lane Montgomery butterflies, multi-stage merging, and stride-3 radix-3 vectorization. The pointwise multiplication (BaseMul) sees even larger gains (4.2–7.5×) because the baseline scalar code performs one Montgomery reduction per coefficient, whereas the NEON path processes four in parallel with no additional overhead.

Sign-3 exhibits somewhat lower kernel-level speedups than Sign-1 and Sign-5. This is because NCC-Sign-3 uses $n = 1536 = 2^9 \times 3$ with a single radix-3 stage at $len \geq 4$, so the stride-3 vectorization of Section 3.8 does not apply; moreover, its base multiplication uses a degree-2 residue structure requiring gather/scatter access patterns that reduce NEON throughput.

Table 11 shows the cycle breakdown of the optimized build for key generation on Sign-1, illustrating the relative weight of each component.

Amdahl's Law Interpretation

On M1 Pro, the combined NTT + INTT cost in the clean baseline accounts for approximately 63% of a single signing attempt for NCC-Sign-1. With per-kernel speedups of 5–6×, the theoretical end-to-end upper bound under Amdahl's law is approximately $1/(0.37 + 0.63/5.5) \approx 2.07\times$. The measured end-to-end signing speedup of 2.28× (best of 20 runs) exceeds this single-kernel bound because the pointwise multiplication and the radix-3 vectorization contribute additional, independent gains.

For key generation, the NTT/INTT/BaseMul kernels occupy only 8.5% of the total cost (Table 11); even infinite speedup on these kernels would yield at most a 1.09× end-to-end improvement. The measured 1.41× reflects savings in other vectorized components (polynomial arithmetic helpers such as `poly_reduce` and `poly_caddq`), but further improvement

requires accelerating the Keccak-based sampling path, which we identify as future work (Section 5).

Table 11. Cycle breakdown of NCC-Sign-1 key generation (optimized build, Apple M1 Pro). The NTT/INTT/BaseMul kernels account for only 8.5% of the total, with Keccak-based sampling dominating at 30.7%.

Category	Cycles	Share
`poly_uniform` (ExpandA)	35,978	30.7%
NTT	4649	4.0%
INTT	4786	4.1%
BaseMul	542	0.5%
`poly_uniform_eta`	4902	4.2%
`poly_power2round`	4095	3.5%
Packing (`pack_sk/pk`)	4278	3.6%
Other †	58,080	49.5%
Total	**117,310**	**100%**

† Includes `poly_reduce`, `poly_caddq`, `poly_sub`, `poly_add`, and other small polynomial arithmetic helpers.

4.4. Cross-Scheme Comparison

To assess the competitiveness of our optimized NCC-Sign against a state-of-the-art ARMv8 implementation of a standardized scheme, we benchmarked the NEON-optimized ML-DSA (Dilithium) of Becker et al. [11] on the same two platforms under the same measurement methodology (best of 20 runs, 10,000 iterations). The ML-DSA code was compiled from the `neon-ntt` repository with `-O3`; on Apple M1 Pro it used the same `kperf/kpc` cycle counter, and on Cortex-A72 it used the PMU kernel module provided in the repository. Both implementations used the same `fips202/fips202x2` modules for Keccak hashing and were compiled with identical optimization flags (`-O3 -fomit-frame-pointer` on M1 Pro, `-O3` on Cortex-A72) to ensure a fair comparison.

Table 12 compares NCC-Sign and ML-DSA (Dilithium) on Apple M1 Pro at matching security levels.

Table 12. Cycle counts of optimized NCC-Sign and NEON-optimized Dilithium on Apple M1 Pro (best of 20 runs). Bold entries indicate the faster scheme at each security level.

Scheme	Level	KeyGen	Sign	Verify
Dilithium-2	1	**71,930**	224,737	**71,803**
NCC-Sign-1	1	117,310	**157,270**	106,590
Dilithium-3	3	**155,325**	347,067	**108,579**
NCC-Sign-3	3	161,638	**228,157**	152,556
Dilithium-5	5	**184,386**	415,347	**174,835**
NCC-Sign-5	5	235,254	**318,339**	210,566

Table 13 presents the same comparison on Cortex-A72.

Table 13. Cycle counts of optimized NCC-Sign and NEON-optimized Dilithium on Arm Cortex-A72 (best of 20 runs).

Scheme	Level	KeyGen	Sign	Verify
Dilithium-2	1	**303,396**	663,828	**297,784**
NCC-Sign-1	1	312,534	**602,042**	406,379

Table 13. *Cont.*

Scheme	Level	KeyGen	Sign	Verify
Dilithium-3	3	568,489	816,447	**492,878**
NCC-Sign-3	3	**416,284**	**787,940**	539,245
Dilithium-5	5	873,268	**1,122,437**	842,065
NCC-Sign-5	5	**625,889**	1,219,615	**823,245**

Discussion

On Apple M1 Pro, NCC-Sign produces signatures faster than ML-DSA at all three security levels, by 23–34%. On Cortex-A72, NCC-Sign is faster at Levels 1 and 3 (by 3–9%), but at Level 5, ML-DSA's signing is 8.7% faster (1122k vs. 1220k cycles), likely because ML-DSA's power-of-two NTT benefits more from the Cortex-A72's narrower pipeline at larger transform sizes. ML-DSA's advantage in key generation and verification on M1 Pro reflects its module-lattice structure, where the $k \times l$ matrix $\mathbf{A}$ enables parallel NTT calls that amortize sampling and hashing costs more effectively than NCC-Sign's single-polynomial ring. Conversely, NCC-Sign's signing path benefits from a lower expected rejection count and a more compact NTT pipeline.

On Cortex-A72, an interesting reversal occurs at Levels 3 and 5: NCC-Sign's key generation becomes faster than ML-DSA's (416k vs. 568k at Level 3, 626k vs. 873k at Level 5). This suggests that ML-DSA's matrix-expansion overhead scales less favorably on the narrower Cortex-A72 pipeline, whereas NCC-Sign's single-polynomial `ExpandA` remains lightweight.

For completeness, Table 14 reproduces the NEON-optimized HAETAE results reported by Sim et al. [13] on the same Apple M1 Pro platform.

Table 14. HAETAE NEON-optimized performance on Apple M1 Pro reported by Sim et al. [13]. Unit: µs per operation (derived from 10,000-iteration wall-clock totals). These figures use wall-clock timing rather than PMU cycle counts and are included for qualitative reference only.

Scheme	Level	KeyGen (µs)	Sign (µs)	Verify (µs)
HAETAE-120	2	111.4	681.8	31.6
HAETAE-180	3	129.5	372.1	54.5
HAETAE-260	5	217.7	6686.4	69.2

Because Sim et al. employed wall-clock timing rather than PMU cycle counters, the measurements include OS scheduling and DVFS overhead that our methodology excludes; a direct numerical comparison with Tables 6 and 12 is therefore not appropriate. Nevertheless, two qualitative observations are possible. First, HAETAE-260 (Level 5) exhibits a signing cost roughly two orders of magnitude higher than Levels 2–3, reflecting its larger module dimensions and substantially higher rejection rate. Second, at Levels 2 and 3, HAETAE's verification is notably fast (31.6–54.5 µs), while its signing cost (372–682 µs) places it in a broadly similar range to ML-DSA on the same platform, and both are slower than our optimized NCC-Sign in the signing operation.

4.5. Constant-Time Analysis

A cryptographic implementation intended for deployment must not leak secret information through execution-time variations. Our NEON kernels are designed to be branch-free and data-independent with respect to secret values: all NEON instructions used (`vqdmulhq_s32`, `vmulq_s32`, `vhsubq_s32`, `vaddq_s32`, `vsubq_s32`, `vld1q_s32`, `vst1q_s32`, `vld3q_s32`, `vst3q_s32`) are specified as fixed-latency operations in the Arm architecture,

and no secret-dependent branch or table lookup appears in any kernel. To substantiate this claim empirically, we apply the dudect methodology of Reparaz et al. [37].

4.5.1. Methodology

dudect performs a Welch t-test on execution-time distributions collected under two input classes. Following recommended practice, we used a fixed–random versus random–random design: class 0 supplied a fixed (but randomly chosen) input polynomial, and class 1 supplied a fresh random polynomial for each measurement. This avoids false positives that can arise from degenerate inputs such as all-zero polynomials, whose cache-access patterns may differ systematically from typical inputs without implying a secret-dependent timing channel. Each test accumulated 30 million measurements on Apple M1 Pro. A maximum $|t|$-value below five was interpreted as no detected leakage.

Table 15 summarizes the results.

All NEON-optimized kernels—NTT, INTT, and pointwise multiplication—passed the dudect threshold on all three parameter sets, with a comfortable margin (maximum $|t| = 4.10$). The clean scalar pointwise multiplication also passed, confirming that the NEON vectorization did not introduce timing leakage absent from the reference code.

Two functions reported FAIL, both for design-level reasons unrelated to our optimization. First, `poly_chknorm` uses an early-return structure that terminates as soon as any coefficient exceeds a public bound; this is a norm-checking utility whose output determines only whether the signing attempt is rejected, not any secret value. Second, `crypto_sign` as a whole exhibits large timing variation ($|t| > 100$) because the Fiat–Shamir-with-Aborts paradigm entails a rejection loop whose iteration count depends on the sampled intermediate values. This variable-time behavior is inherent to the algorithm design and is shared by all Fiat–Shamir-with-Aborts schemes, including the NIST-standardized ML-DSA. It does not constitute a side-channel vulnerability, as the number of rejection iterations does not reveal information about the long-term signing key.

Table 15. Dudect leakage-detection results on Apple M1 Pro (fixed–random vs. random–random, 30 M measurements). PASS: max $|t| < 5$; FAIL entries are design-level timing variations, not implementation defects (see text).

Function	Sign-1	Sign-3	Sign-5
NTT (optimized)	PASS (2.09)	PASS (1.84)	PASS (2.13)
INTT (optimized)	PASS (1.32)	PASS (2.10)	PASS (2.36)
Pointwise (optimized)	PASS (4.10)	PASS (3.53)	PASS (1.27)
Pointwise (clean)	PASS (1.96)	PASS (2.10)	PASS (1.80)
poly_chknorm	FAIL * *(10.19)*	FAIL * *(10.34)*	FAIL * *(10.23)*
crypto_sign	FAIL * *(163)*	FAIL * *(567)*	FAIL * *(536)*

* Design-level timing variation (early-return or rejection loop), not an implementation defect. See text for discussion.

4.5.2. Limitations

dudect is an empirical, black-box test; passing it does not constitute a formal proof of constant-time execution. Subtle microarchitectural effects (e.g., operand-dependent forwarding latencies, cache-bank conflicts) may escape statistical detection at the measurement granularity used. Formal verification of constant-time properties, for instance via tools such as ct-verif or Binsec, is left as future work.

4.6. Montgomery vs. Barrett Microbenchmark

Section 3 justified the choice of Montgomery multiplication over Barrett reduction on theoretical grounds. Here, we validate that choice empirically with a microbenchmark on Apple M1 Pro, measuring the cost of a single radix-2 NTT butterfly implemented with each reduction strategy.

For Barrett reduction, we implemented a standard multiply-by-reciprocal sequence adapted to NCC-Sign's modulus $Q = 8{,}401{,}537$: a high-multiply to approximate the quotient, a shift, a correction multiply by Q, and a conditional subtraction. The NEON Barrett path required approximately nine vector instructions per butterfly, compared to five for the Montgomery path (Section 3.2.1). Both implementations were verified against the same set of 100,000 random inputs.

Table 16 reports the per-butterfly cost.

Table 16. Radix-2 NTT butterfly cost (cycles per butterfly) on Apple M1 Pro. Montgomery reduction requires fewer NEON instructions and achieves a 23% advantage in the vectorized path.

Reduction	Scalar	NEON 4-Lane
Montgomery	2.19	**0.74**
Barrett	2.71	0.95

Montgomery's advantage stems from two factors. First, its five-instruction critical path (`vqdmulhq`, `vmul`, `vmul`, `vqdmulhq`, `vhsub`) is shorter than Barrett's nine-instruction sequence, directly reducing arithmetic latency. Second, Montgomery operates entirely within 32-bit lanes using the doubling high-multiply primitive, whereas a faithful Barrett implementation on NEON requires careful management of intermediate precision, since ARMv8 NEON lacks a native 64-bit integer multiply. Together these factors yield a consistent 23% speed advantage per butterfly in the NEON path and 19% in the scalar path, confirming that Montgomery is the better fit for NCC-Sign's mixed-radix NTT on ARMv8.

5. Discussion

5.1. Transferability and Broader Relevance

NCC-Sign was not selected in the final KpqC standardization; AIMer and HAETAE were chosen for signatures. Nevertheless, the optimization techniques developed in this work have relevance beyond the specific scheme. The baseline NEON kernels (four-lane Montgomery multiplication, centered reduction, multi-stage radix-2 merging) are directly applicable to any lattice-based scheme whose arithmetic reduces to modular polynomial multiplication via NTT, including ML-DSA, ML-KEM, and HAETAE. The stride-3 vectorization technique of Section 3.8, which exploits NEON structure load/store instructions to handle radix-3 stages below the SIMD lane width, is specific to mixed-radix NTTs of the form $n = 2^a \cdot 3^b$. Such transforms arise in NCC-Sign and could appear in future schemes that adopt non-power-of-two polynomial rings for algebraic or security reasons. We note, however, that the current NIST and KpqC standards exclusively use power-of-two NTTs, so the immediate applicability of the radix-3 technique is limited to non-standard settings.

More broadly, the function-level profiling methodology and Amdahl's law's analysis framework presented in Section 4.3 are scheme-agnostic and can guide optimization prioritization for any NTT-based cryptographic workload. The dudect-based constant-time verification workflow (Section 4.5) is likewise reusable across implementations.

5.2. Energy Considerations

Our NEON vectorization reduces the dynamic instruction count for NTT-intensive operations by processing four coefficients per instruction rather than one. On both Apple M1 Pro and Cortex-A72, this translates to fewer executed instructions for the same workload, which generally correlates with lower dynamic energy consumption. However, we do not provide quantitative energy measurements in this work, as cycle-accurate energy counters are not publicly exposed on either platform. Obtaining per-operation energy data via external power monitors or platform-specific profiling tools (e.g., Apple's `powermetrics`) remains future work.

5.3. Limitations

Our evaluation covers two ARMv8-A microarchitectures (Apple M1 Pro and Arm Cortex-A72). While these represent distinct design points—a high-performance wide-issue core and a widely deployed embedded core—additional platforms (e.g., Cortex-A55, Cortex-A76, Cortex-X series) would further characterize portability. All kernels use 128-bit NEON only; exploring Arm SVE/SVE2 with scalable vectors is a natural extension. The Keccak-based sampling (`ExpandA`) dominates key-generation cost (Section 4.3) and is not accelerated by our current NEON kernels; leveraging ARMv8.2-SHA3 instructions for single-stream Keccak acceleration is a promising direction. Finally, the dudect assessment is empirical and does not replace formal constant-time verification; tools such as ct-verif or Binsec could provide stronger guarantees.

6. Conclusions

This paper presented an ARMv8/NEON-optimized implementation of NCC-Sign for the mixed-radix NTT trinomial parameter sets (NCC-Sign-1/3/5). Beyond the baseline lane-local SIMD kernels, we introduced radix-2 multi-stage butterfly merging, a stride-3 vectorization technique using NEON structure load/store instructions for small-len radix-3 stages, and NEON-parallel pointwise Montgomery multiplication.

On Apple M1 Pro, the optimized build achieved geometric-mean speedups of **1.40**× for key generation, **2.24**× for signing, and **2.01**× for verification, with individual kernel gains of up to 5–6× (NTT/INTT) and 7.5× (pointwise multiplication). On Arm Cortex-A72, consistent speedups of 1.25× (KeyGen), 1.54× (Sign), and 1.44× (Verify) confirmed cross-platform portability. A direct comparison with the NEON-optimized ML-DSA (Dilithium) showed that NCC-Sign produced signatures 23–34% faster on M1 Pro at all security levels, while ML-DSA retained an advantage in key generation and verification due to its module-lattice structure.

Empirical constant-time testing via dudect confirmed that all NEON kernels passed the leakage-detection threshold under 30 million measurements, and a Montgomery-versus-Barrett microbenchmark validated the modular-reduction design choice with a measured 23% speed advantage.

Author Contributions: Conceptualization, M.L. and H.S.; methodology, M.L.; software, M.L.; validation, M.L., M.S. and S.E.; formal analysis, M.L.; investigation, M.L.; resources, H.S.; data curation, M.L.; writing—original draft preparation, M.L.; writing—review and editing, M.S., S.E. and H.S.; visualization, M.L.; supervision, H.S.; project administration, H.S.; funding acquisition, H.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was financially supported by Hansung University.

Data Availability Statement: The optimized implementation and benchmarking/validation scripts are publicly available at https://github.com/minunejip/PQC-ARMv8 (accessed on 27 March 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Shor, P.W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* **1997**, *26*, 1484–1509. [CrossRef]
2. Grover, L.K. Quantum Mechanics Helps in Searching for a Needle in a Haystack. *Phys. Rev. Lett.* **1997**, *79*, 325–328. [CrossRef]
3. CISA; NSA; NIST. *Quantum-Readiness: Migration to Post-Quantum Cryptography*; Technical Report; U.S. Cybersecurity and Infrastructure Security Agency: Washington, DC, USA, 2023. Available online: https://www.cisa.gov/sites/default/files/2023-08/Quantum%20Readiness_Final_CLEAR_508c%20(3).pdf (accessed on 19 February 2026).
4. NIST PQC Project. Post-Quantum Cryptography Project. 2025. Available online: https://csrc.nist.gov/Projects/post-quantum-cryptography (accessed on 19 February 2026).
5. NIST. *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)*; Technical Report; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2024. Available online: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.203.pdf (accessed on 19 February 2026).
6. NIST. *FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA)*; Technical Report; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2024. Available online: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.204.pdf (accessed on 19 February 2026).
7. NIST. *FIPS 205: Stateless Hash-Based Digital Signature Standard (SLH-DSA)*; Technical Report; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2024. Available online: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.205.pdf (accessed on 19 February 2026).
8. KpqC Team. Selected Algorithms from the KpqC Competition Round 1. 2023. Available online: https://www.kpqc.or.kr/competition.html (accessed on 19 February 2026).
9. KpqC Team. Selected Algorithms from the KpqC Competition Round 2 (Final Selection). 2025. Available online: https://kpqc.or.kr/competition_02.html (accessed on 19 February 2026).
10. Arm Limited. Armv8-A Instruction Set Architecture: An Introduction to the Armv8 Instruction Sets. 2025. Available online: https://developer.arm.com/documentation/den0024/a/An-Introduction-to-the-ARMv8-Instruction-Sets (accessed on 19 February 2026).
11. Becker, H.; Hwang, V.; Kannwischer, M.J.; Yang, B.Y.; Yang, S.Y. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2022**, *2022*, 221–244. [CrossRef]
12. Nguyen, D.T.; Gaj, K. Fast Falcon Signature Generation and Verification Using ARMv8 NEON Instructions. In Proceedings of the NIST 4th PQC Standardization Conference, Virtual, 29 November–1 December 2022. Available online: https://csrc.nist.gov/csrc/media/Events/2022/fourth-pqc-standardization-conference/documents/papers/fast-falcon-signature-generation-and-verification-pqc2022.pdf (accessed on 19 February 2026).
13. Sim, M.; Lee, M.; Seo, H. HAETAE on ARMv8. *Electronics* **2024**, *13*, 3863. [CrossRef]
14. Choi, Y.; Kim, M.; Kim, Y.; Song, J.; Jin, J.; Kim, H.; Seo, S.C. KpqBench: Performance and Implementation Security Analysis of KpqC Competition Round 1 Candidates. *IEEE Access* **2024**, *12*, 18606–18626. [CrossRef]
15. Sim, M.; Song, G.; Eum, S.; Lee, M.; Yoon, S.; Baksi, A.; Seo, H. Integrating and Benchmarking KpqC in TLS/X.509. *Electronics* **2025**, *14*, 3717. [CrossRef]
16. Kim, S.; Cho, J.; Cho, M.; Ha, J.; Kwon, J.; Lee, B.; Lee, J.; Lee, J.; Lee, S.; Moon, D.; et al. *The AIMer Signature Scheme: Algorithm Specifications*; Technical Report; NIST PQC (Digital Signatures Additional Round): Gaithersburg, MD, USA, 2023. Available online: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/AIMer-spec-web.pdf (accessed on 19 February 2026).
17. Cheon, J.H.; Choe, H.; Devevey, J.; Güneysu, T.; Hong, D.; Krausz, M.; Land, G.; Shin, J.; Stehlé, D.; Yi, M. *HAETAE: Algorithm Specifications and Supporting Documentation*; Technical Report; KpqC/NIST PQC (Digital Signatures Additional Round): Gaithersburg, MD, USA, 2023. Available online: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/haetae-spec-web.pdf (accessed on 19 February 2026).
18. Kim, J.; Park, J.H. *NTRU+: Compact Construction of NTRU Using Simple Encoding Method*; Technical report; KpqC Submission: Daejeon, Republic of Korea, 2022. Available online: https://www.kpqc.or.kr/images/pdf/NTRU%2B.pdf (accessed on 19 February 2026).
19. Cheon, J.H.; Choe, H.; Choi, J.; Hong, D.; Hong, J.; Jung, C.G.; Kang, H.; Lee, J.; Lim, S.; Park, A.; et al. *SMAUG-T: The Key Exchange Algorithm Based on Module-LWE and Module-LWR*; Technical Report; KpqC Submission: Daejeon, Republic of Korea, 2024. Available online: https://kpqc.or.kr/images/pdf/SMAUG-T_Document.pdf (accessed on 19 February 2026).
20. Shim, K.A.; Kim, J.; An, Y. *NCC-Sign: A New Lattice-Based Signature Scheme Using Non-Cyclotomic Polynomials*; Technical Report; National Institute for Mathematical Sciences: Daejeon, Republic of Korea, 2023. Available online: https://www.kpqc.or.kr/images/pdf/NCC-Sign.pdf (accessed on 19 February 2026).
21. Lyubashevsky, V. Fiat–Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In *Proceedings of the ASIACRYPT*; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5912, pp. 598–616.

22. Bai, S.; Galbraith, S.D. Lattice Signatures from Rejection Sampling. In *Proceedings of the ASIACRYPT*; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8874, pp. 117–137. [CrossRef]
23. Fouque, P.A.; Hoffstein, J.; Kirchner, P.; Lyubashevsky, V.; Pornin, T.; Prest, T.; Ricosset, T.; Seiler, G.; Whyte, W.; Zhang, Z. *Falcon: Fast-Fourier Lattice-Based Compact Signatures over NTRU. Specification v1.2*; Technical Report; Falcon Team: Gennevilliers, France, 2020. Available online: https://falcon-sign.info/falcon.pdf (accessed on 19 February 2026).
24. Bernstein, D.J.; Hülsing, A.; Kölbl, S.; Niederhagen, R.; Papachristodoulou, T.; Rijneveld, J.; Schwabe, P. The SPHINCS+ Signature Framework. In Proceedings of the ACM CCS, London, UK, 11–15 November 2019; pp. 2129–2146. [CrossRef]
25. Arm Limited. Procedure Call Standard for the Arm 64-Bit Architecture (AAPCS64). 2025. Available online: https://github.com/ARM-software/abi-aa/releases (accessed on 19 February 2026).
26. Arm Limited. Arm C Language Extensions (ACLE). 2025. Available online: https://arm-software.github.io/acle/main/acle.html (accessed on 19 February 2026).
27. Arm Limited. Arm NEON Intrinsics Reference (Advanced SIMD). 2025. Available online: https://arm-software.github.io/acle/neon_intrinsics/advsimd.html (accessed on 19 February 2026).
28. Nguyen, D.T.; Gaj, K. Fast NEON-Based Multiplication for Lattice-Based NIST Post-Quantum Cryptography Finalists. In *Proceedings of the Post-Quantum Cryptography (PQCrypto 2021)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2021; Volume 12710, pp. 234–254. [CrossRef]
29. Bernstein, D.J.; Schwabe, P. NEON Crypto. In *Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2012*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7428, pp. 320–339. [CrossRef]
30. Nguyen, D.T. Apple M1/M2 Cycle Counter Library (kperf/kpc). 2022. Available online: https://gist.github.com/dougallj/5bafb113492047c865c0c8cfbc930155 (accessed on 19 February 2026).
31. Lemire, D. Counting Cycles and Instructions on ARM-Based Apple Systems. 2023. Available online: https://lemire.me/blog/2023/03/21/counting-cycles-and-instructions-on-arm-based-apple-systems/ (accessed on 19 February 2026).
32. ibireme. A Demo Showing How to Read Intel or Apple M1 CPU Performance Counters on macOS (kperf/kpc). 2022. Available online: https://gist.github.com/ibireme/173517c208c7dc333ba962c1f0d67d12 (accessed on 19 February 2026).
33. Apple Inc. Tuning Your Code's Performance for Apple Silicon. 2024. Available online: https://developer.apple.com/documentation/apple-silicon/tuning-your-code-s-performance-for-apple-silicon (accessed on 19 February 2026).
34. KPQC-CryptoCraft. KPQClean: Benchmark on Korean Post Quantum Cryptography. 2024. Available online: https://github.com/kpqc-cryptocraft/KPQClean (accessed on 19 February 2026).
35. KPQC-CryptoCraft. KpqClean_ver2: Benchmark on KPQC (Ver. 2). 2024. Available online: https://github.com/kpqc-cryptocraft/KpqClean_ver2 (accessed on 19 February 2026).
36. Kannwischer, M.J.; Schwabe, P.; Stebila, D.; Wiggers, T. Improving Software Quality in Cryptography Standardization Projects. In Proceedings of the Security Standardization Research (SSR 2022), Genoa, Italy, 6 June 2022. Available online: https://kannwischer.eu/papers/2022_pqclean.pdf (accessed on 19 February 2026).
37. Reparaz, O.; Balasch, J.; Verbauwhede, I. Dude, is my code constant time? In Proceedings of the Design, Automation & Test in Europe Conference (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 1697–1702. Available online: https://eprint.iacr.org/2016/1123 (accessed on 24 March 2026). [CrossRef]