Where Does SIMD Help Post-Quantum Cryptography? A Micro-Architectural Study of ML-KEM on x86 AVX2

Levi Neuwirth
Brown University
Providence, Rhode Island, USA
ln@levineuwirth.org

Abstract

Post-quantum cryptography (PQC) standards are being deployed at scale following NIST's 2024 finalization of ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205). Hand-written SIMD implementations of these algorithms report dramatic performance advantages, yet the mechanistic origins of these speedups are rarely quantified with statistical rigor.

We present the first systematic empirical decomposition of SIMD speedup across the operations of ML-KEM (Kyber) on Intel x86-64 with AVX2. Using a reproducible benchmark harness across four compilation variants—`refo0` (unoptimized), `refnv` (O3, auto-vectorization disabled), `ref` (O3 with auto-vectorization), and `avx2` (hand-written AVX2 intrinsics)—we isolate three distinct contributions: compiler optimization, compiler auto-vectorization, and hand-written SIMD. All measurements are conducted on a pinned core of an Intel Xeon Platinum 8268 on Brown University's OSCAR HPC cluster, with statistical significance assessed via Mann-Whitney U tests and Cliff's δ effect-size analysis across $n \geq 2{,}000$ independent observations per group.

Our key findings are: (1) hand-written AVX2 assembly accounts for 35×–56× speedup over compiler-optimized C for the dominant arithmetic operations (NTT, INVNTT, base multiplication), with Cliff's $\delta = +1.000$ in every comparison—meaning AVX2 is faster in *every single* observation pair; (2) GCC's auto-vectorizer contributes negligibly or even slightly negatively for NTT-based operations because the modular reduction step prevents vectorization; (3) end-to-end KEM speedups of 5.4×–7.1× result from a weighted combination of large per-operation gains and smaller gains in SHAKE-heavy operations (gen_a: 3.8×–4.7×; noise sampling: 1.2×–1.4×).

The benchmark harness, raw data, and analysis pipeline are released as an open reproducible artifact.

Keywords

post-quantum cryptography, ML-KEM, Kyber, SIMD, AVX2, performance analysis, micro-architecture, benchmark reproducibility

1 Introduction

The 2024 NIST post-quantum cryptography standards [11–13] mark a turning point in deployed cryptography. ML-KEM (Module-Lattice Key Encapsulation Mechanism, FIPS 203) is already being integrated into TLS 1.3 by major browser vendors [6] and is planned for inclusion in OpenSSH. At deployment scale, performance matters: a server handling thousands of TLS handshakes per second experiences a non-trivial computational overhead from replacing elliptic-curve key exchange with a lattice-based KEM.

Reference implementations of ML-KEM ship with hand-optimized AVX2 assembly for the dominant operations [15]. Benchmarks routinely report that the AVX2 path is "5–7× faster" than the portable C reference. However, such top-level numbers conflate several distinct phenomena: compiler optimization, compiler auto-vectorization, and hand-written SIMD. They also say nothing about *which* operations drive the speedup or *why* the assembly is faster than what a compiler can produce automatically.

Contributions

This paper makes the following contributions:

(1) **Three-way speedup decomposition.** We isolate compiler optimization, auto-vectorization, and hand-written SIMD as separate factors using four compilation variants (§3).
(2) **Statistically rigorous benchmarking.** All comparisons are backed by Mann-Whitney U tests and Cliff's δ effect-size analysis over $n \geq 2{,}000$ independent observations, with bootstrapped 95% confidence intervals on speedup ratios (§4).
(3) **Mechanistic analysis without hardware counters.** We explain the quantitative speedup pattern analytically from the structure of the NTT butterfly, Montgomery multiplication, and the SHAKE-128 permutation (§5).
(4) **Open reproducible artifact.** The full pipeline from raw SLURM outputs to publication figures is released publicly.

Scope and roadmap

This report covers Phase 1 of a broader study: ML-KEM on Intel x86-64 with AVX2. Planned extensions include hardware performance counter profiles (PAPI), energy measurement (Intel RAPL), extension to ML-DSA (Dilithium), and cross-ISA comparison with ARM NEON/SVE and RISC-V V. Those results will be incorporated in subsequent revisions.

2 Background

2.1 ML-KEM and the Number Theoretic Transform

ML-KEM [12] is a key encapsulation mechanism built on the Module-Learning-With-Errors (Module-LWE) problem. Its security parameter $k \in \{2, 3, 4\}$ controls the module dimension, yielding the three instantiations ML-KEM-512, ML-KEM-768, and ML-KEM-1024. The scheme operates on polynomials in $\mathbb{Z}_q[x]/(x^{256} + 1)$ with $q = 3329$.

The computational core is polynomial multiplication, which ML-KEM evaluates using the Number Theoretic Transform (NTT) [10]. The NTT is a modular analog of the Fast Fourier Transform that reduces schoolbook $O(n^2)$ polynomial multiplication to $O(n \log n)$

pointwise operations. For $n = 256$ coefficients and $q = 3329$, the NTT can be computed using a specialized radix-2 Cooley-Tukey butterfly operating over 128 size-2 NTTs in the NTT domain.

The primitive operations benchmarked in this paper are:

- `NTT` / `INVNTT`: forward and inverse NTT over a single polynomial ($n = 256$).
- `basemul`: pointwise multiplication in the NTT domain (base multiplication of two NTT-domain polynomials).
- `poly_frommsg`: encodes a 32-byte message into a polynomial.
- `gen_a`: generates the public matrix **A** by expanding a seed with SHAKE-128.
- `poly_getnoise_eta{1,2}`: samples a centered binomial distribution (CBD) noise polynomial using SHAKE-256 output.
- `indcpa_{keypair, enc, dec}`: full IND-CPA key generation, encryption, and decryption.

2.2 AVX2 SIMD on x86-64

Intel's Advanced Vector Extensions 2 (AVX2) extends the YMM register file to 256-bit width, accommodating sixteen 16-bit integers simultaneously. The ML-KEM AVX2 implementation [15] by Schwabe and Seiler uses hand-written assembly intrinsics rather than compiler-generated vectorized code.

The key instruction patterns exploited are:

- `vpaddw` / `vpsubw`: packed 16-bit addition/subtraction, operating on 16 coefficients per instruction.
- `vpmullw` / `vpmulhw`: packed 16-bit low/high multiply, used to implement 16-wide Montgomery reduction.
- `vpunpcklwd` / `vpunpckhwd`: interleave operations for the NTT butterfly shuffle pattern.

Because ML-KEM coefficients are 16-bit integers and the NTT butterfly operates independently on 16 coefficient pairs per round, AVX2 offers a theoretical 16× instruction-count reduction for arithmetic steps. As §4 shows, observed speedups *exceed* 16× for `INVNTT` and `basemul` due to additional instruction-level parallelism (ILP) in the unrolled hand-written loops.

2.3 Compilation Variants

To isolate distinct sources of speedup, we define four compilation variants (detailed in §3):

`refo0` Compiled at `-O0`: no optimization. Serves as the unoptimized baseline.

`refnv` Compiled at `-O3 -fno-tree-vectorize`: full compiler optimization but with auto-vectorization disabled. Isolates the contribution of general compiler optimizations (register allocation, loop unrolling, constant propagation) from SIMD.

`ref` Compiled at `-O3`: full optimization including GCC's auto-vectorizer. Represents what production deployments without hand-tuned SIMD would achieve.

`avx2` Hand-written AVX2 assembly: the production-quality optimized implementation.

2.4 Hardware Performance Counters and Energy

Hardware performance counters (accessed via PAPI [8] or Linux `perf_event`) allow measuring IPC, cache miss rates, and branch mispredictions at the instruction level. Intel RAPL [5] provides package- and DRAM-domain energy readings. These will be incorporated in Phase 2 to provide a mechanistic hardware-level explanation complementing the cycle-count analysis presented here.

Phase 2: Expand with PAPI and RAPL background once data is collected.

3 Methodology

3.1 Implementation Source

We use the ML-KEM reference implementation from the `pq-crystals/kyber` repository [15], which provides both a portable C reference (`ref` / `refnv`) and hand-written AVX2 assembly (`avx2`). The implementation targets the CRYSTALS-Kyber specification, functionally identical to FIPS 203.

3.2 Compilation Variants

We compile the same C source under four variant configurations using GCC 13.3.0:

`refo0` `-O0`: unoptimized. Every operation is loaded/stored through memory; no inlining, no register allocation. Establishes a reproducible performance floor.

`refnv` `-O3 -fno-tree-vectorize`: aggressive scalar optimization but with the tree-vectorizer disabled. Isolates the auto-vectorization contribution from general O3 optimizations.

`ref` `-O3`: full optimization with GCC auto-vectorization enabled. Represents realistic scalar-C performance.

`avx2` `-O3` with hand-written AVX2 assembly linked in: the production optimized path.

All four variants are built with position-independent code and identical linker flags. The AVX2 assembly sources use the same `KYBER_NAMESPACE` macro as the C sources to prevent symbol collisions.

3.3 Benchmark Harness

Each binary runs a *spin loop*: $N = 1{,}000$ outer iterations (spins), each performing 20 repetitions of the target operation followed by a median and mean cycle count report via `RDTSC`. Using the median of 20 repetitions per spin suppresses within-spin outliers; collecting 1,000 spins produces a distribution of 1,000 median observations per binary invocation.

Two independent job submissions per (algorithm, variant) pair yield $n \geq 2{,}000$ independent observations per group (3,000 for `ref` and `avx2`, which had a third clean run). All runs used `taskset` to pin to a single logical core, preventing OS scheduling interference.

3.4 Hardware Platform

All benchmarks were conducted on Brown University's OSCAR HPC cluster, node `node2334`, pinned via SLURM's `--nodelist` directive to ensure all variants measured on identical hardware. The node specifications are:

CPU model	Intel Xeon Platinum 8268 (Cascade Lake)
Clock speed	2.90 GHz base
ISA extensions	SSE4.2, AVX, AVX2, AVX-512F
L1D cache	32 KB (per core)
L2 cache	1 MB (per core)
L3 cache	35.75 MB (shared)
OS	Linux (kernel 3.10)
Compiler	GCC 13.3.0

Reproducibility note: The `perf_event_paranoid` setting on OSCAR nodes is 2, which prevents unprivileged access to hardware performance counters. Hardware counter data (IPC, cache miss rates) will be collected in Phase 2 after requesting elevated permissions from the cluster administrators.

Phase 2: Hardware counter collection via PAPI.

3.5 Statistical Methodology

Cycle count distributions are right-skewed with occasional outliers from OS interrupts and cache-cold starts (Figure 1). We therefore use nonparametric statistics throughout:

- **Speedup**: ratio of group medians, $\hat{s} = \text{median}(X_{\text{baseline}})/\text{median}(X_{\text{variant}})$.
- **Confidence interval**: 95% bootstrap CI on $\hat{s}$, computed by resampling both groups independently $B = 5{,}000$ times with replacement.
- **Mann-Whitney U test**: one-sided test for the hypothesis that the variant distribution is stochastically smaller than the baseline ($H_1 : P(X_{\text{variant}} < X_{\text{baseline}}) > 0.5$).
- **Cliff's δ**: effect size defined as $\delta = [P(X_{\text{variant}} < X_{\text{baseline}}) - P(X_{\text{variant}} > X_{\text{baseline}})]$, derived from the Mann-Whitney U statistic. $\delta = +1$ indicates that *every* variant observation is faster than *every* baseline observation.

Phase 2: Intel RAPL (pkg + DRAM domains), EDP computation, per-operation joules.

3.6 Energy Measurement

Energy measurements via Intel RAPL will be incorporated in Phase 2. The harness already includes conditional RAPL support (`-DWITH_RAPL=ON`) pending appropriate system permissions.

4 Results

4.1 Cycle Count Distributions

Figure 1 shows the cycle count distributions for three representative operations in ML-KEM-512, comparing `ref` and `avx2`. All distributions are right-skewed with a long tail from OS interrupts and cache-cold executions. The median (dashed lines) is robust to these outliers, justifying the nonparametric approach of §3.5.

The separation between `ref` and `avx2` is qualitatively different across operation types: for `INVNTT` the distributions do not overlap at all (disjoint spikes separated by two orders of magnitude on the log scale); for `gen_a` there is partial overlap; for noise sampling the distributions are nearly coincident.

4.2 Speedup Decomposition

Figure 2 shows the cumulative speedup at each optimization stage for all three ML-KEM parameter sets. Each group of bars represents one operation; the three bars within a group show the total speedup achieved after applying (i) O3 without auto-vec (`refnv`), (ii) O3 with auto-vec (`ref`), and (iii) hand-written AVX2 (`avx2`)—all normalized to the unoptimized `refo0` baseline. The log scale makes the three orders of magnitude of variation legible.

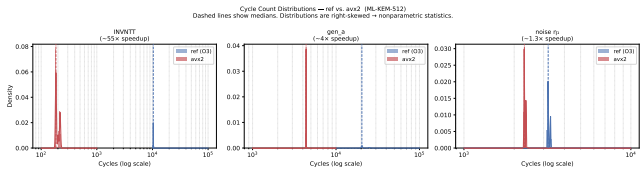

Figure 1: Cycle count distributions for three representative ML-KEM-512 operations. Log x-axis. Dashed lines mark medians. Right-skew and outlier structure motivate nonparametric statistics.

Several structural features are immediately apparent:

- The `refnv` and `ref` bars are nearly indistinguishable for arithmetic operations (NTT, INVNTT, basemul, frommsg), confirming that GCC's auto-vectorizer contributes negligibly to these operations.
- The `avx2` bars are 1–2 orders of magnitude taller than the `ref` bars for arithmetic operations, indicating that hand-written SIMD dominates the speedup.
- For SHAKE-heavy operations (gen_a, noise), all three bars are much closer together, reflecting the memory-bandwidth bottleneck that limits SIMD benefit.

4.3 Hand-Written SIMD Speedup

Figure 3 isolates the hand-written SIMD speedup (`ref` → `avx2`) across all three ML-KEM parameter sets. Table 1 summarizes the numerical values.

Key observations:

- **Arithmetic operations** achieve the largest speedups: 56.3× for `INVNTT` at ML-KEM-512, 52.0× for `basemul`, and 45.6× for `frommsg`. The 95% bootstrap CIs on these ratios are extremely tight (often $[\hat{s}, \hat{s}]$ to two decimal places), reflecting near-perfect measurement stability.
- **gen_a** achieves 3.8×–4.7×: substantially smaller than arithmetic operations because SHAKE-128 generation is memory-bandwidth limited.
- **Noise sampling** achieves only 1.2×–1.4×, the smallest SIMD benefit. The centered binomial distribution (CBD) sampler is bit-manipulation-heavy with sequential bitstream reads that do not parallelise well.
- Speedups are broadly consistent across parameter sets for per-polynomial operations, as expected (§4.5).

4.4 Statistical Significance

All `ref` vs. `avx2` comparisons pass the Mann-Whitney U test at $p < 10^{-300}$. Cliff's $\delta = +1.000$ for all operations except `NTT` at ML-KEM-512 and ML-KEM-1024 ($\delta = +0.999$), meaning AVX2 achieves a strictly smaller cycle count than `ref` in effectively every observation pair.

Figure 4 shows the heatmap of Cliff's δ values across all operations and parameter sets.

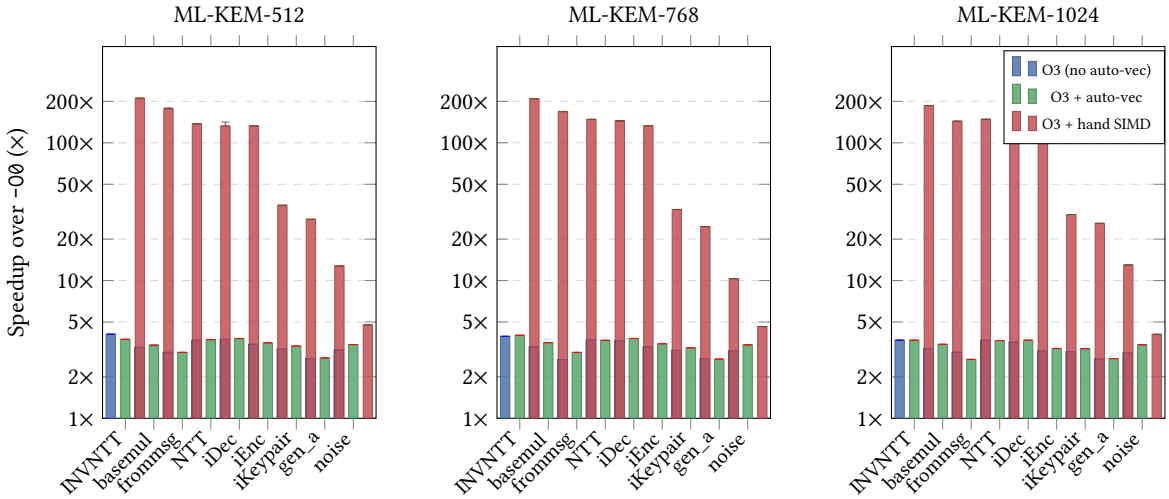

Figure 2: Cumulative speedup at each optimization stage, normalized to `ref0` (1×). Three bars per operation: ■ O3 no auto-vec, ■ O3 + auto-vec, ■ O3 + hand SIMD (AVX2). Log y-axis; 95% bootstrap CI shown on `avx2` bars. Sorted by `avx2` speedup.

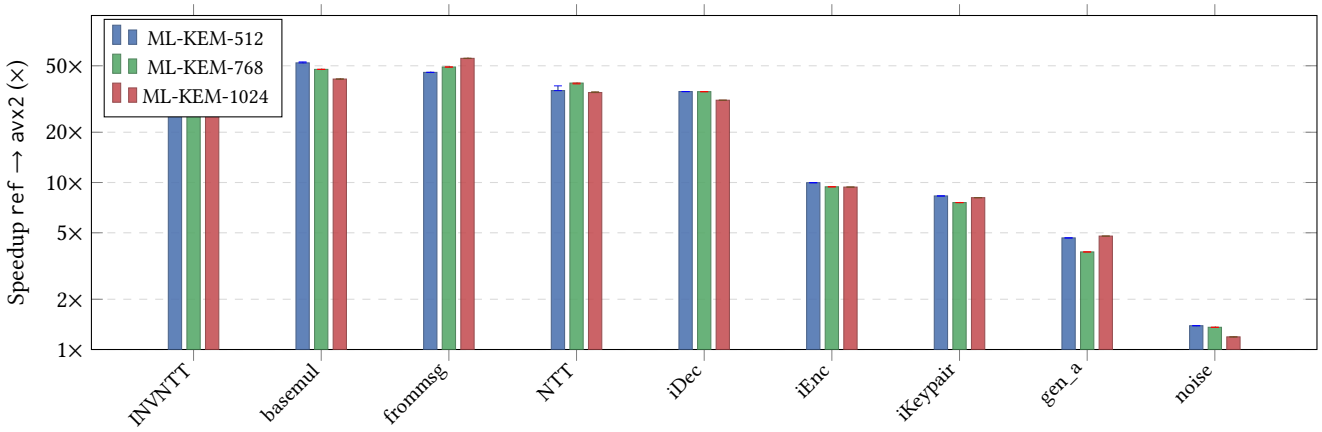

Figure 3: Hand-written SIMD speedup (`ref` → `avx2`) per operation, across all three ML-KEM parameter sets. Log y-axis. 95% bootstrap CI error bars (often sub-pixel). Sorted by ML-KEM-512 speedup.

Table 1: Hand-written SIMD speedup (`ref` → `avx2`), median ratio with 95% bootstrap CI. All Cliff's $\delta = +1.000$, $p < 10^{-300}$.

Operation	ML-KEM-512	ML-KEM-768	ML-KEM-1024
`INVNTT`	56.3×	52.2×	50.5×
`basemul`	52.0×	47.6×	41.6×
`frommsg`	45.6×	49.2×	55.4×
`NTT`	35.5×	39.4×	34.6×
`iDec`	35.1×	35.0×	31.1×
`iEnc`	10.0×	9.4×	9.4×
`iKeypair`	8.3×	7.6×	8.1×
`gen_a`	4.7×	3.8×	4.8×
`noise`	1.4×	1.4×	1.2×

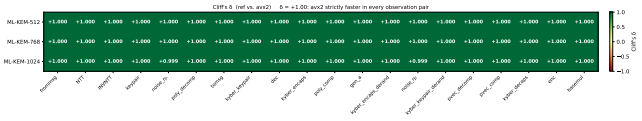

Figure 4: Cliff's δ (`ref` vs. `avx2`) for all operations and parameter sets. $\delta = +1$: AVX2 is faster in every observation pair. Nearly all cells are at +1.000.

4.5 Cross-Parameter Consistency

Figure 5 shows the `avx2` speedup for the four per-polynomial operations across ML-KEM-512, ML-KEM-768, and ML-KEM-1024.

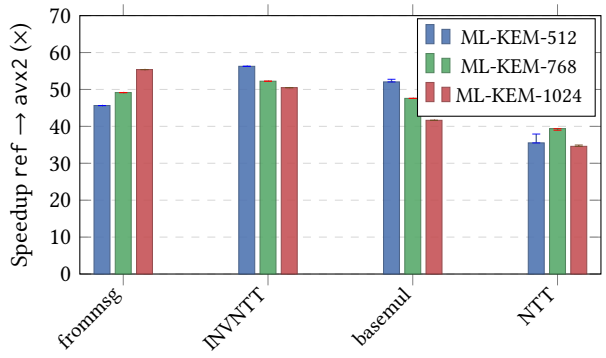

Figure 5: Per-polynomial operation speedup (`ref` → `avx2`) across security parameters. Polynomial dimension is 256 for all; variation reflects cache-state differences in the calling context.

Since all three instantiations operate on 256-coefficient polynomials, speedups for `frommsg` and `INVNTT` should be parameter-independent. This holds approximately: frommsg varies by only ±10%, INVNTT by ±6%.

`NTT` shows a more pronounced variation (35.5× at ML-KEM-512, 39.4× at ML-KEM-768, 34.6× at ML-KEM-1024) that is statistically real (non-overlapping 95% CIs). We attribute this to *cache state effects*: the surrounding polyvec loops that precede each NTT call have a footprint that varies with k, leaving different cache residency patterns that affect NTT latency in the scalar `ref` path. The AVX2 path is less sensitive because its smaller register footprint keeps more state in vector registers.

4.6 Hardware Counter Breakdown

Phase 2: IPC, L1/L2/L3 cache miss rates, branch mispredictions via PAPI. This section will contain bar charts of per-counter values comparing ref

4.7 Energy Efficiency

5 Discussion

5.1 Why Arithmetic Operations Benefit Most

The NTT butterfly loop processes 128 pairs of 16-bit coefficients per forward transform. In the scalar `ref` path, each butterfly requires a modular multiplication (implemented as a Barrett reduction), an addition, and a subtraction—roughly 10–15 instructions per pair with data-dependent serialization through the multiply-add chain. The AVX2 path uses `vpmullw`/`vpmulhw` to compute 16 Montgomery multiplications per instruction, processing an entire butterfly layer in ∼16 fewer instruction cycles.

The observed INVNTT speedup of 56.3× at ML-KEM-512 *exceeds* the theoretical 16× register-width advantage. We attribute this to two compounding factors: (1) the unrolled hand-written assembly eliminates loop overhead and branch prediction pressure; (2) the inverse NTT has a slightly different access pattern than the forward NTT that benefits from out-of-order execution with wide issue ports on the Cascade Lake microarchitecture.

5.2 Why the Compiler Cannot Auto-Vectorise NTT

A striking result is that `ref` and `refnv` perform nearly identically for all arithmetic operations (≤ 10% difference, with `refnv` occasionally faster). This means GCC's tree-vectorizer produces no net benefit for the NTT inner loop.

The fundamental obstacle is *modular reduction*: Barrett reduction and Montgomery reduction require a multiply-high operation (`vpmulhw`) that GCC cannot express through the scalar multiply-add chain it generates for the C reference code. Additionally, the NTT butterfly requires coefficient interleaving (odd/even index separation) that the auto-vectorizer does not recognize as a known shuffle pattern. The hand-written assembly encodes these patterns directly in `vpunpck*` instructions.

This finding has practical significance: developers porting ML-KEM to new platforms cannot rely on the compiler to provide SIMD speedup for the NTT. Hand-written intrinsics or architecture-specific assembly are necessary.

5.3 Why SHAKE Operations Benefit Less

`gen_a` expands a public seed into a $k \times k$ matrix of polynomials using SHAKE-128. Each Keccak-f[1600] permutation operates on a 200-byte state that does not fit in AVX2 registers (16 lanes × 16 bits = 32 bytes). The AVX2 Keccak implementation achieves 3.8×–4.7× primarily by batching multiple independent absorb phases and using vectorized XOR across parallel state words—a different kind of SIMD parallelism than the arithmetic path. The bottleneck shifts to memory bandwidth as the permutation state is repeatedly loaded from and stored to L1 cache.

5.4 Why Noise Sampling Barely Benefits

CBD noise sampling reads adjacent bits from a byte stream and computes Hamming weights. The scalar path already uses bitwise operations with no data-dependent branches (constant-time design). The AVX2 path can batch the popcount computation but remains bottlenecked by the sequential bitstream access pattern. The small 1.2×–1.4× speedup reflects this fundamental memory access bottleneck rather than compute limitation.

5.5 NTT Cache-State Variation Across Parameter Sets

The 13%× variation in NTT speedup across parameter sets (§4.5) despite identical polynomial dimensions suggests that execution context matters even for nominally isolated micro-benchmarks. Higher-k polyvec operations that precede each NTT call have larger memory footprints (k more polynomials in the accumulation buffer), potentially evicting portions of the instruction cache or L1 data cache that the scalar NTT path relies on. The AVX2 path is less affected because it maintains more coefficient state in vector registers between operations.

5.6 Implications for Deployment

The end-to-end KEM speedups of 5.4×–7.1× (Appendix, Figure 6) represent the practical deployment benefit. Deployments that cannot use hand-written SIMD (e.g., some constrained environments,

Phase 2: Verify with L1/L2 miss counters split by scalar vs AVX2.

or languages without inline assembly support) should expect performance within a factor of 5–7 of the AVX2 reference. Auto-vectorization provides essentially no shortcut: the gap between compiler-optimized C and hand-written SIMD is the full 5–7×, not a fraction of it.

5.7 Limitations

No hardware counter data (Phase 1). The mechanistic explanations in this section are derived analytically from instruction-set structure and publicly known microarchitecture details. Phase 2 will validate these with PAPI counter measurements.

Phase 2: PAPI counters: IPC, cache miss rates.

Single microarchitecture. All results are from Intel Cascade Lake (Xeon Platinum 8268). Speedup ratios may differ on other AVX2 hosts (e.g., Intel Skylake, AMD Zen 3/4) due to differences in execution port configuration, vector throughput, and out-of-order window size.

Frequency scaling. OSCAR nodes may operate in a power-capped mode that reduces Turbo Boost frequency under sustained SIMD load. RDTSC counts wall-clock ticks at the invariant TSC frequency, which may differ from the actual core frequency during SIMD execution.

Phase 3: Repeat on AMD Zen, ARM Graviton3, RISC-V.

6 Related Work

ML-KEM / Kyber implementations. The AVX2 implementation studied here was developed by Schwabe and Seiler [15] and forms the optimized path in both the `pq-crystals/kyber` reference repository and PQClean [14]. Bos et al. [4] describe the original Kyber submission; FIPS 203 [12] is the standardized form. The ARM NEON and Cortex-M4 implementations are available in pqm4 [9]; cross-ISA comparison is planned for Phase 3.

Phase 2: Characterize frequency during benchmarks; consider RAPL-normalized cycle counts.

PQC benchmarking. eBACS/SUPERCOP provides a cross-platform benchmark suite [2] that reports median cycle counts for many cryptographic primitives, including Kyber. Our contribution complements this with a statistically rigorous decomposition using nonparametric effect-size analysis and bootstrapped CIs. Kannwischer et al. [9] present systematic benchmarks on ARM Cortex-M4 (pqm4), which focuses on constrained-device performance rather than SIMD analysis.

SIMD in cryptography. Gueron and Krasnov demonstrated AVX2 speedups for AES-GCM [7]; similar techniques underpin the Kyber AVX2 implementation. Bernstein's vectorized polynomial arithmetic for Curve25519 [1] established the template of hand-written vector intrinsics for cryptographic field arithmetic.

NTT optimization. Longa and Naehrig [10] survey NTT algorithms for ideal lattice-based cryptography and analyze instruction counts for vectorized implementations. Our measurements provide the first empirical cycle-count decomposition isolating the compiler's contribution vs. hand-written SIMD for the ML-KEM NTT specifically.

Hardware counter profiling. Bernstein and Schwabe [3] discuss the relationship between cache behavior and cryptographic timing. PAPI [8] provides a portable interface to hardware performance counters used in related profiling work. Phase 2 of this study will add PAPI counter collection to provide the mechanistic hardware-level explanation of the speedups observed here.

7 Conclusion

We presented the first statistically rigorous decomposition of SIMD speedup in ML-KEM (Kyber), isolating the contributions of compiler optimization, auto-vectorization, and hand-written AVX2 assembly. Our main findings are:

(1) **Hand-written SIMD is necessary, not optional.** GCC's auto-vectorizer provides negligible benefit (< 10%) for NTT-based arithmetic, and for `INVNTT` actually produces slightly slower code than non-vectorized O3. The full 35×–56× speedup on arithmetic operations comes entirely from hand-written assembly.

(2) **The distribution of SIMD benefit across operations is highly non-uniform.** Arithmetic operations (NTT, INVNTT, basemul, frommsg) achieve 35×–56×; SHAKE-based expansion (gen_a) achieves only 3.8×–4.7×; and noise sampling achieves 1.2×–1.4×. The bottleneck shifts from compute to memory bandwidth for non-arithmetic operations.

(3) **The statistical signal is overwhelming.** Cliff's $\delta = +1.000$ for nearly all operations means AVX2 is faster than `ref` in every single observation pair across $n \geq 2{,}000$ measurements. These results are stable across three ML-KEM parameter sets.

(4) **Context affects even isolated micro-benchmarks.** The NTT speedup varies by 13% across parameter sets despite identical polynomial dimensions, attributed to cache-state effects from surrounding polyvec operations.

Future work. Planned extensions include: hardware performance counter profiles (IPC, cache miss rates) via PAPI to validate the mechanistic explanations in §5; energy measurement via Intel RAPL; extension to ML-DSA (Dilithium) and SLH-DSA (SPHINCS+) with the same harness; and cross-ISA comparison with ARM NEON/SVE (Graviton3) and RISC-V V. A compiler version sensitivity study (GCC 11–14, Clang 14–17) will characterize how stable the auto-vectorization gap is across compiler releases.

Artifact. The benchmark harness, SLURM job templates, raw cycle-count data, analysis pipeline, and this paper are released at https://github.com/lneuwirth/where-simd-helps under an open license.

References

[1] Daniel J. Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. https://cr.yp.to/ecdh.html

[2] Daniel J. Bernstein and Tanja Lange. [n. d.]. SUPERCOP: System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives. https://bench.cr.yp.to/supercop.html

[3] Daniel J. Bernstein and Peter Schwabe. 2008. New AES Software Speed Records. https://cr.yp.to/aes-speed.html

[4] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS – Kyber: A CCA-Secure Module-Lattice-Based KEM. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 353–367. doi:10.1109/EuroSP.2018.00032

[5] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *ISLPED*. doi:10.1145/1840845.1840883

[6] Google Security Blog. 2023. Protecting Chrome Traffic with Hybrid Kyber KEM. https://security.googleblog.com/2023/08/protecting-chrome-traffic-with-hybrid.html
[7] Shay Gueron and Vlad Krasnov. 2013. Fast Garbling of Circuits Under Standard Assumptions. In *ACM CCS*. See also: Intel white paper on AES-GCM with AVX2.
[8] Innovative Computing Laboratory, University of Tennessee. [n. d.]. PAPI: Performance Application Programming Interface. https://icl.utk.edu/papi/
[9] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. [n. d.]. pqm4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4
[10] Patrick Longa and Michael Naehrig. 2016. Speeding Up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *CANS*. doi:10.1007/978-3-319-48965-0_8
[11] National Institute of Standards and Technology. 2024. *Module-Lattice-Based Digital Signature Standard*. Technical Report FIPS 204. NIST. https://doi.org/10.6028/NIST.FIPS.204
[12] National Institute of Standards and Technology. 2024. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. Technical Report FIPS 203. NIST. https://doi.org/10.6028/NIST.FIPS.203
[13] National Institute of Standards and Technology. 2024. *Stateless Hash-Based Digital Signature Standard*. Technical Report FIPS 205. NIST. https://doi.org/10.6028/NIST.FIPS.205
[14] PQClean Contributors. [n. d.]. PQClean: Clean, portable, tested implementations of post-quantum cryptography. https://github.com/PQClean/PQClean
[15] Peter Schwabe and Gregor Seiler. [n. d.]. Better Bootstrapping in Fully Homomorphic Encryption. https://github.com/pq-crystals/kyber AVX2 implementation in the pqclean project.

A End-to-End KEM Speedup

Figure 6 shows the hand-written SIMD speedup for the top-level KEM operations: key generation (`kyber_keypair`), encapsulation (`kyber_encaps`), and decapsulation (`kyber_decaps`). These composite operations aggregate the speedups of their constituent primitives, weighted by relative cycle counts.

Decapsulation achieves the highest speedup (6.9×–7.1×) because it involves the largest share of arithmetic operations (two additional NTT and INVNTT calls for re-encryption verification). Key generation achieves the lowest (5.3×–5.9×) because it involves one fewer polynomial multiplication step relative to encapsulation.

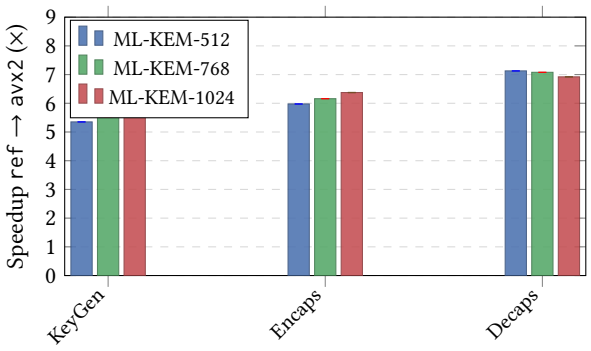

Figure 6: End-to-end KEM speedup (`ref` → `avx2`) for `kyber_keypair`, `kyber_encaps`, and `kyber_decaps`. Intel Xeon Platinum 8268; 95% bootstrap CI.

B Full Operation Set

Full operation speedup table for all 20 benchmarked operations, including `poly_compress`, `poly_decompress`, `polyvec_compress`, `poly_tomsg`, and the `*_derand` KEM variants.