Optimizing ML-KEM with OpenMP: A Multilevel Parallelization Study

Vinícius Lagrota[1], **Tertuliano Souza Neto**[1],
Thiago do Rêgo Sousa[1], **Fábio Maia**[2]

[1] Research and Development Center for Communication Security (CEPESC)
Brasília, DF – Brazil.

[2]Integrated Center for Advanced Security Systems (CISSA)
Recife, PE – Brazil.

vinicius.lagrota@gmail.com, tsouzaneto@gmail.com,
thiagodoregosousa@gmail.com, fmww@cesar.org.br

Abstract. *In this work, we present* `libcesarpq`*, a cryptographic library that integrates an optimized implementation of the post-quantum Key Encapsulation Mechanism ML-KEM. We perform a detailed analysis of the algorithm's source code to identify opportunities for parallelization using OpenMP directives. We show that, since most core ML-KEM functions have very short execution time, the overhead introduced can severely limit performance gain. Therefore, we evaluate different parallelization strategies at both the core-layer functions and the API-layer, as well as the batched API-layer. The results are compared with the widely used post-quantum library* `liboqs` *in both single-threaded and multithreaded environments, showing comparable performance on the* `x86-64` *platform and a performance advantage for* `libcesarpq` *on the optimized* `aarch64` *platform.*

1. Introduction

Three decades ago, Peter Shor introduced a quantum algorithm capable of breaking several widely used public-key cryptosystems [Shor 1994]. Shortly thereafter, Lov Grover introduced a quantum search algorithm [Grover 1996] that can potentially reduce the effective security level of symmetric-key cryptosystems by a factor of two. The combined impact of Shor's algorithm on public-key schemes and Grover's algorithm on symmetric primitives motivated the cryptographic community to develop what is now known as post-quantum cryptography (PQC). Consequently, the design of robust cryptographic schemes based on hard mathematical problems that remain intractable for both classical and quantum computers has become an active and rapidly growing research area. Indeed, research and development in PQC have significantly accelerated over the past decade.

In this context, the National Institute for Standards and Technology (NIST) launched a public PQC standardization project [Chen et al. 2016] with the goal of selecting and standardizing public-key cryptographic algorithms for key encapsulation, shared secret establishment, and digital signatures. After four evaluation rounds spanning about six years, NIST selected one algorithm for key encapsulation mechanism (KEM) and three algorithms for digital signatures. The chosen KEM algorithm, originally known

as CRYSTALS-Kyber, was standardized as *Module-Lattice-Based Key-Encapsulation Mechanism Standard* (ML-KEM) in the Federal Information Processing Standards Publication (FIPS)-203 specification [NIST 2024].

The security of ML-KEM algorithm relies on the hardness of the Module-Learning With Errors (MLWE) problem, a generalization of the Learning With Errors (LWE) problem introduced by Regev [Regev 2005] and formally studied in the context of modules by Langlois and Stehlé [Langlois and Stehlé 2015]. ML-KEM belongs to the family of lattice-based cryptosystems and supports three security categories: ML-KEM-512, providing security comparable to AES-128 (NIST security category 1); ML-KEM-768, with security comparable to AES-192 (NIST security category 3); and ML-KEM-1024, offering security comparable to AES-256 (NIST security category 5).

These three security categories are closely related to the parameter k, which determines the module dimension used in the underlying MLWE problem. In particular, the scheme operates over the k-fold cartesian product of the polynomial ring R_q, denoted by R_q^k, where $R_q = \frac{\mathbb{Z}_q[X]}{(X^n+1)}$. In practice, each security category corresponds to a specific value of k. More precisely, ML-KEM-512 uses $k = 2$, ML-KEM-768 uses $k = 3$, and ML-KEM-1024 uses $k = 4$. Increasing the value of k enlarges the dimension of the module lattice and consequently increases the computational hardness of the underlying MLWE problem. As a result, higher values of k provide stronger security guarantees, at the cost of larger key sizes and higher computational complexity.

By combining strong theoretical foundations with practical efficiency [NIST 2024, Avanzi et al. 2019, Bos et al. 2018], ML-KEM represents a cornerstone of the PQC transition. However, despite its suitability for standardization, the computational cost of post-quantum schemes can still pose challenges, particularly in resource-constrained environments [Kim and Seo 2025, Lagrota et al. 2025] or in settings where the algorithm must be executed in batches. This motivates the exploration of optimization techniques, such as parallel programming, to further improve performance.

1.1. Related Works

Improving the performance of ML-KEM and related post-quantum schemes has attracted significant attention due to the high computational cost of lattice-based cryptography. One line of work focuses on improving the software implementation itself. [Paiva et al. 2025] proposed tailored error-correction codes for lattice-based KEMs and implemented compact ML-KEM instantiations with negligible performance impact. [Becker et al. 2022] accelerated lattice-based schemes on Armv8-A through optimized Number Theoretic Transform (NTT) arithmetic and Neon vectorization, while [Wei et al. 2026] presented an optimized ML-KEM implementation on ARMv9-A using SVE2 and SME. These works show that software- and code-level optimizations can substantially improve performance.

Another line of research explores hardware-assisted acceleration. [Gewehr et al. 2024] investigated instruction-set extensions for Kyber/ML-KEM on RISC-V embedded systems, while [Ni et al. 2025] and [Nguyen et al. 2025] proposed efficient hardware architectures for ML-KEM. In addition, [Lagrota et al. 2022] presented a configurable hardware/software co-design combining an ARM processor with FPGA acceleration. Overall, these studies demonstrate the benefits of specialized hardware support, custom instruction sets, and hardware/software co-design.

A third direction investigates parallel execution. [Azevedo et al. 2025] proposed selective parallelization of ML-KEM on a dual-core ESP32, showing that the gains depend strongly on function granularity and parallelization overhead. However, the literature still lacks a systematic study of OpenMP-based parallelization for ML-KEM on general-purpose multicore processors. In particular, there is still limited understanding of how OpenMP behaves across different abstraction levels in PQC, especially in ML-KEM, ranging from fine-grained internal routines to higher-level, more computationally demanding operations. This work addresses that gap.

1.2. Our Contributions

The main contributions of this work can be summarized as follows:

- A detailed analysis of the ML-KEM algorithm, identifying opportunities for parallelization and investigating OpenMP-based strategies at different abstraction levels of the implementation.
- An extensive experimental evaluation across x86-64 and AArch64 architectures, assessing the impact of parallelization at the core-layer, the Application Programming Interface (API)-layer, and the batched API-layer functions.
- The design and implementation of the post-quantum cryptographic library `libcesarpq`, which incorporates the proposed OpenMP-based parallelization strategies and serves as the experimental platform for the analyses presented in this work.

2. Preliminaries

A dramatic shift in processor design has changed the history of microprocessor performance. Although the last quarter of the 20th century saw an unprecedented growth in processor clock speeds, the current century has seen a marked slowdown in this trend [Pacheco and Malensek 2021]. As a result, contemporary performance gains are driven primarily by parallelism rather than by increases in single-processor speed. In response to this trend, the industry shifted its focus toward the integration of multicore processors onto a single chip. Such processors consist of multiple single-core units and taking advantage of their capabilities requires the development of software that exploits the power of multiple processing elements, commonly referred to as parallel programs. Therefore, parallel programming plays a central role in a wide range of application domains, including climate modeling, drug discovery, energy research, data analysis, and cryptography.

As parallel computing developed, the need to measure the performance of parallel programs emerged, which is accomplished through performance metrics. Beyond simply measuring execution time, these metrics make it possible to identify potential bottlenecks in code execution and to determine whether parallel programming should, in fact, be employed for a particular application, since in some cases performance may even be worse than that of a serial execution. The goal of executing a procedure in parallel is to accelerate the execution of a given algorithm. At least three factors affect processing time: the type of hardware used, the degree of parallelism inherent to the algorithm, and the parallelization model adopted.

The metric that quantifies the benefit of parallelizing an algorithm is known as *speedup*. Let T_s denote the execution time of the algorithm in its serial form, that is,

without parallelism, and let T_p denote the execution time under parallel execution with p threads. Speedup is then defined as

$$S(p) = \frac{T_s}{T_p}. \tag{1}$$

In the ideal scenario, the entire computational workload is evenly distributed among the p threads, with no overhead due to thread creation, synchronization, communication, scheduling, or idleness. In this case, the speedup is given by $S(p) = p$. When this condition is achieved, the parallel program is said to exhibit *linear speedup*.

A useful way to express the trade-off between the execution time of the algorithm in its serial form (i.e., useful work), T_s, and overhead introduced by parallel execution with p threads, $T_{oh}(p)$, is to model the total execution time as

$$T_p(p) = \frac{T_s}{p} + T_{oh}(p), \tag{2}$$

where $T_p(p)$ denotes the total parallel execution time measured when using p threads. In practical implementations, the overhead $T_{oh}(p)$ may include thread-management costs, synchronization, barriers, scheduling, and other costs not associated with useful computation. Equation 2 highlights that increasing the number of threads reduces the time consumed by the ideally parallelizable portion of the computation, represented by T_s/p, but does not eliminate the overhead term. On the contrary, depending on the implementation and the execution environment, $T_{oh}(p)$ may remain significant or even increase with the number of threads. Therefore, although parallel execution may reduce the total execution time, the resulting speedup is typically sublinear.

3. Implementation Strategy for Parallel ML-KEM using OpenMP

Since many cryptographic algorithms and libraries are implemented in the C language and are typically executed on general-purpose processing units (CPUs), OpenMP is a suitable choice for optimizing such functions due to its simplicity and high portability, as it is supported by multiple compilers, architectures, and operating systems. In this context, this section details the API extension and the internal modifications introduced to incorporate OpenMP into ML-KEM. Our analysis considers three abstraction layers in the ML-KEM implementation, as illustrated in Figure 1.

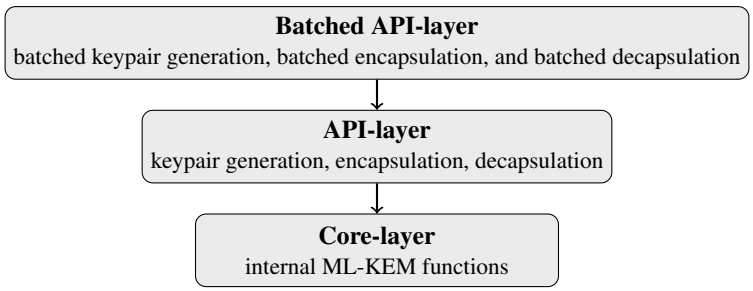

Figure 1. Three abstraction layers in the ML-KEM analysis.

The lowest abstraction layer — the core-layer — is composed of the internal ML-KEM functions, which fundamentally depend on the ML-KEM security level (k,

defined in Section 1). Above it, the API-layer includes the functions provided by the original ML-KEM API — keypair generation, encapsulation, and decapsulation — and relies on the core-layer functions. Finally, the uppermost layer is the batched API-layer. This layer extends the original API by allowing the API-layer functions to be processed in batches — namely, batched keypair generation, batched encapsulation, and batched decapsulation.

To support this analysis while preserving the correctness of the cryptographic operations, we developed a cryptographic library called `libcesarpq`[1], which supports both single-threaded and multithreaded operations depending on compilation and configuration parameters. The library automatically detects the number of available cores, thereby improving usability, but it also provides an API through which users can limit the maximum number of threads. This cryptographic library allows the functions used in the ML-KEM algorithm to be executed with the appropriate OpenMP directives when enabled on different architectures. We also adopted an incremental development strategy: starting from a correct and fully functional baseline implementation, we progressively introduced parallel constructs and verified correctness after each modification.

Sections 3.1 through 3.3 discuss each abstraction layer shown in Figure 1, ranging from the lowest layer (core-layer) to the highest layer (batched API-layer).

3.1. Core-layer

This section discusses how OpenMP was included in the core-layer — i.e., applied to the internal ML-KEM functions — using the `mlkem-native`[2] implementation as a starting point. The overall strategy adopted to optimize the internal ML-KEM functions was to analyze the source code of each internal function involved in the three main operations (keypair generation, encapsulation, and decapsulation), together with their respective subroutines, and identify iterative blocks that account for a significant portion of the total computation time and can be executed independently. These blocks were then selected as candidates for parallel regions executed by independent threads.

A preliminary analysis of the individual internal ML-KEM functions revealed that the true potential for parallelization is in the functions related to polynomial algebra. The existence of `for` loops in most of the functions make them good initial candidates for the inclusion of parallel regions. Furthermore, it was verified that the `for` loops are independent, because each loop operates in a polynomial ring of the ML-KEM algorithm, which are independent from each other.

Most of the functions have loops of size varying according to the security level $k \in \{2, 3, 4\}$. This is the case for functions such as `mlk_polyvec_{mulcache_compute, tobytes, frombytes, add, compress_du, ntt, invntt_tomont, tomont, assert_bound_2d}`, `mlk_polymat_permute_bitrev_to_custom`, and `mlk_matvec_mul`. Therefore, we have added parallel regions in these loops by inserting the directive `#pragma omp parallel`. The same strategy was used for the functions `mlk_polyvec_assert_bound_2d` and

[1]More specific, in this work, we used `libcesarpq-v1.5.1`, available in `https://gitlabcissa.cesar.org.br/cepesc/libcesarpq/-/releases/1.5.1`.

[2]It is a secure, fast, and portable implementation of ML-KEM. Available at `https://github.com/pq-code-package/mlkem-native` and also deployed in the widely known `liboqs` library.

`mlk_polyvec_basemul_acc_montgomery_cached_c` whose loop size is $N/2 = 128$, where $N = 256$ for all security levels, and the `mlk_gen_matrix` function whose loop size is $\texttt{nblocks} = (k * k)/4 \in \{1, 2, 4\}$.

As an illustrative example, we analyze the internal ML-KEM function `mlk_gen_matrix`, shown in Code 1. To evaluate the speedup achieved with OpenMP directives, parallel constructs were applied where beneficial, and the same strategy was extended to the remaining internal functions.

Code 1. Snippet of `mlk_gen_matrix`.

```
1  void mlk_gen_matrix(mlk_polymat *a, const uint8_t seed, ... )
2  {
3      const unsigned nblocks = (MLKEM_K * MLKEM_K / 4);
4      ...
5      #pragma omp parallel for ... num_threads(omp_get_max_threads
           () < (int)nblocks ? omp_get_max_threads() : (int)nblocks)
6      for (unsigned b = 0; b < nblocks; b++)
7      {
8          // Operations executed in parallel involving polynomial
               and seed
9          ...
10     }
11     //Remaining operations executed serialized
12     ...
13 }
```

The operations performed in the `for` loop of the function `mlk_gen_matrix`, described in Code 1, are processed parallelized, independently for each value of the variable `b`, which is incremented within the loop. This independence is the primary reason for executing this loop in parallel.

Another factor affecting OpenMP speedup is thread-management overhead (see Equation 2). As a first countermeasure, in all parallel sections we bounded the number of threads by the maximum number of physical cores available on the target hardware. Initial benchmarking further showed that increasing the number of threads beyond the number of loop iterations did not yield any performance improvement. In this situation, the additional threads do not decrease the sequential time of useful parallel work ($\frac{T_s}{p}$ in Equation 2), but only add runtime overhead related to thread coordination, iteration scheduling, and synchronization (T_{oh} in Equation 2). To reduce this overhead, we limited the number of threads according to both the loop size and the number of available cores.

3.2. API-layer

The API-layer implements the original ML-KEM interface, including the conventional keypair generation, encapsulation, and decapsulation functions. No OpenMP-based parallelization is implemented directly at this layer. However, it relies on the core-layer, discussed previously in Section 3.1, where OpenMP parallelization is employed. Therefore, the API-layer makes indirect use of OpenMP through its dependence on the core-layer.

3.3. Batched API-layer

Certain high-performance scenarios would be benefited by batched execution of key exchange operations. Two representative examples are high-throughput Transport Layer Security (TLS) servers, which may need to process thousands of handshakes per second, and secure group messaging systems, in which a single sender establishes shared secrets with multiple recipients simultaneously.

To address these scenarios, the `libcesarpq` library implements an extended API for batched ML-KEM operations, namely batched API-layer. This extension provides wrapper functions that iterate over a fixed number of operations — informed by the user — and execute them in parallel using OpenMP. In particular, the extended API exposes the functions batched keypair generation, batched encapsulation, and batched decapsulation. These functions use OpenMP directives to distribute the workload across threads, following a strategy similar to that described in Section 3.1, but applied at the batch level through the API-layer.

In the batched setting, however, each operation invokes the single-threaded version of the underlying ML-KEM implementation; thus, nested parallelism is intentionally avoided. Furthermore, the batched functions use OpenMP's `schedule(static)` directive, since the number of operations is known in advance, informed by the user in the function arguments, which helps reduce scheduling overhead. They also limit the number of threads according to both the number of batch operations and the number of available cores, thereby avoiding unnecessary thread-management overhead (T_{oh} in Equation 2).

4. Experimental Results for Parallel ML-KEM

Building on Section 3, this section presents the experimental results for the parallel ML-KEM implementation using OpenMP. Among the three NIST security levels for ML-KEM, this work focuses on the lowest security level (ML-KEM-512) and highest security level (ML-KEM-1024) allowing to highlight the difference between them. We evaluate two platforms: an Intel(R) Xeon(R) 6710E in a virtualized environment with 12 available cores and no acceleration instruction set enabled (e.g., SSE2, SSE4, AVX2)[3], denoted `x86-64-ref`; and an Apple Silicon M2 Pro operating natively with 10 available cores (6 performance and 4 efficiency) and all accelerations enabled, denoted `aarch64-native`. These contrasting environments were intentionally chosen to assess the impact of OpenMP under different execution conditions.

For the API-layer and batched API-layer functions, we also compare `libcesarpq` with `liboqs`, an open-source C library widely used as a benchmarking baseline for PQC algorithms [Abbasi et al. 2025]. Both libraries were compiled with the most suitable settings for each target architecture. Section 4.1 analyzes the internal ML-KEM functions, Section 4.2 discusses the API-layer, and Section 4.3 presents the batched API-layer results.

4.1. Performance of the core-layer

Figure 2 presents the benchmark results for the core-layer functions of ML-KEM-512 and ML-KEM-1024 on the `x86-64-ref` and `aarch64-native` architectures under

[3]Acceleration instruction set are automatically disabled by virtualization in this specific test environment.

varying thread counts. The reported values correspond to the speedup relative to the single-threaded model. For each combination of (i) architecture; (ii) thread count, and (iii) core-layer function, the experiment was repeated 50,000 times after a warm-up phase, and the median execution time was used to compute the speedup as in Equation 1.

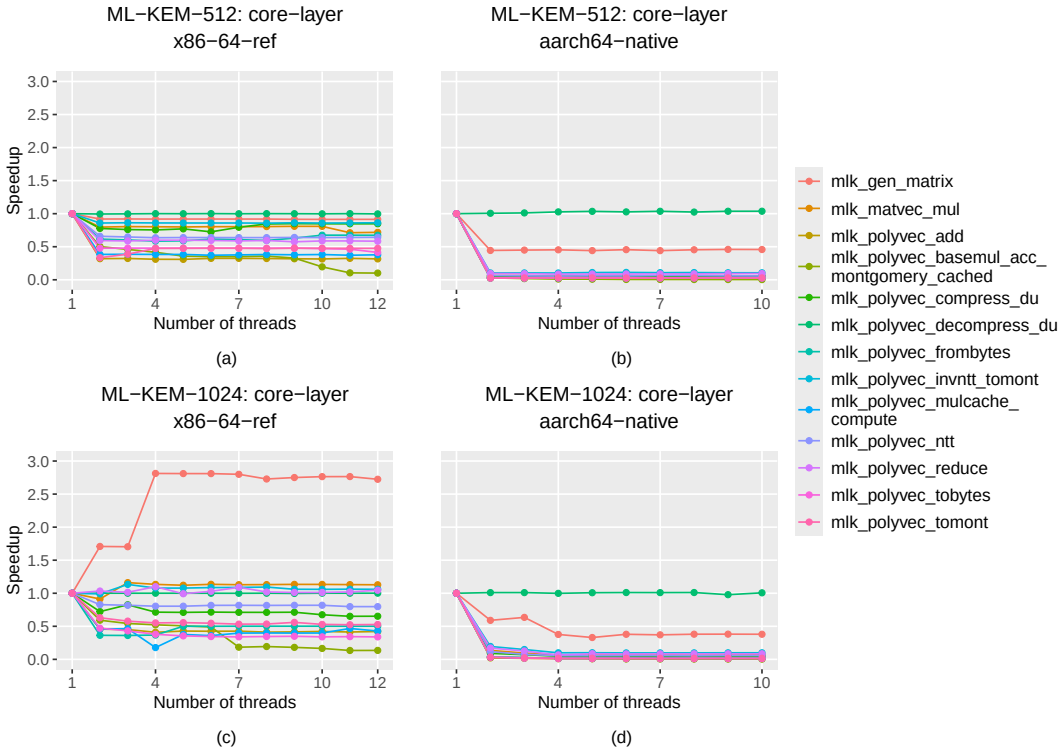

Figure 2. Benchmark results for the core-layer functions of ML-KEM-512 and ML-KEM-1024 on the `x86-64-ref` and `aarch64-native` architectures.

Figure 2 shows similar behavior in plot (a), corresponding to ML-KEM-512 on `x86-64-ref`, and in plots (b) and (d), corresponding to ML-KEM-512 and ML-KEM-1024 on `aarch64-native`, respectively. In all three cases, no function achieved speedup above 1.0, indicating that the overhead introduced by OpenMP was significant relative to the short execution time of the functions. This observation agrees with the literature, which indicates that execution times on the order of tens of microseconds lie near the threshold at which OpenMP can provide meaningful speedup (see [Fredrickson et al. 2003] and Lindberg[4]), and is especially evident for the `aarch64-native` architecture.

In contrast, plot (c) shows that, for ML-KEM-1024 on `x86-64-ref`, the functions `mlk_gen_matrix`, `mlk_matvec_mul`, and `mlk_polyvec_invntt` achieved speedups consistently greater than 1.0. Among them, `mlk_gen_matrix` reached the highest speedup, exceeding 2.5× with four or more threads.

Figure 2 also indicates that the strategy described in Section 3.1 effectively controlled thread-management overhead, since allowing more threads than necessary did not degrade performance.

Since these results were obtained with isolated functions, the speedups may differ

[4]https://software.intel.com/en-us/articles/basic-openmp-threading-overhead

in the complete ML-KEM implementation because of additional parallelization overhead. Therefore, a conservative approach was adopted to avoid degraded performance: in the subsequent analyses, only `mlk_gen_matrix` was parallelized, and only for ML-KEM-1024 on `x86-64-ref`; in all other cases, parallelism remained disabled.

4.2. Performance of the API-layer

Section 4.1 identified the core-layer functions worth parallelizing with OpenMP. As discussed there, only `mlk_gen_matrix` is parallelized, and only for ML-KEM-1024 on `x86-64-ref`, to provide the best performance for `libcesarpq`. In this section, we evaluate the impact of this decision at the API-layer and compare the results with `liboqs`.

Figure 3 presents the API-layer benchmark results for ML-KEM-512 and ML-KEM-1024 on `x86-64-ref` and `aarch64-native` under varying thread counts. The reported values correspond to the speedup relative to the single-threaded model. For each combination of (i) architecture; (ii) thread count; and (iii) API-layer function (i.e., keypair generation, encapsulation, and decapsulation), the experiment was repeated 150,000 times after a warm-up phase, and the median execution time was used to compute the speedup.

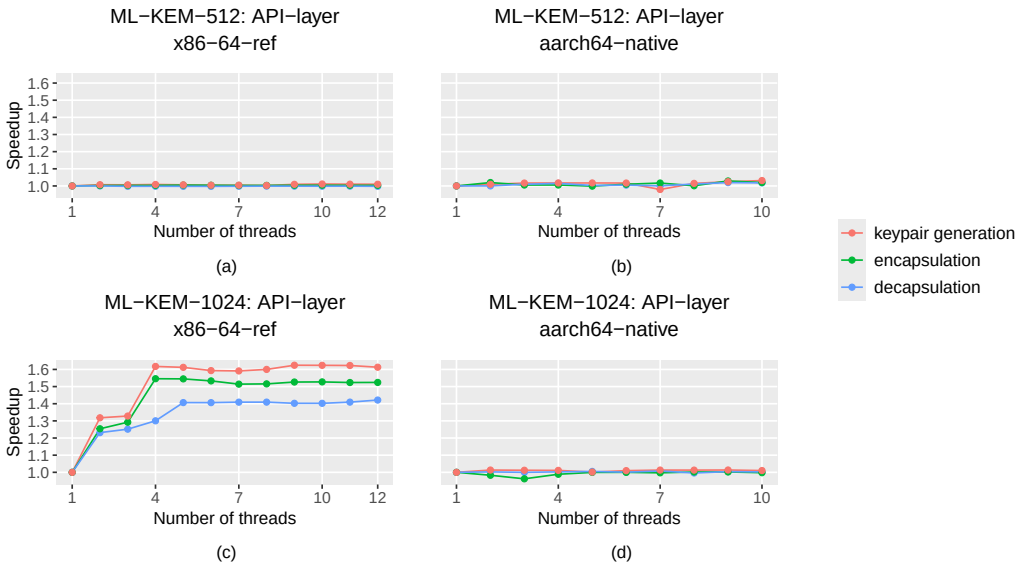

Figure 3. Benchmark results for the API-layer of ML-KEM-512 and ML-KEM-1024 on the `x86-64-ref` and `aarch64-native` architectures.

Consistent with Section 4.1, Figure 3(a), (b), and (d) show no speedup when additional threads are available, since no core-layer function is parallelized for ML-KEM-512 on `x86-64-ref` or for ML-KEM-512 and ML-KEM-1024 on `aarch64-native`. These results indicate that OpenMP introduces no observable overhead when parallelism is disabled.

In contrast, Figure 3(c), corresponding to ML-KEM-1024 on the `x86-64-ref` architecture, shows a speedup ranging from 1.4× to 1.6× for the keypair generation, encapsulation, and decapsulation. These results indicate that the `mlk_gen_matrix` function has a substantial impact on the execution time of the ML-KEM algorithm. When

evaluated in isolation, it achieved a speedup of about 2.5× (see Figure 2(c)); even when combined with the other ML-KEM functions, it still yielded a speedup greater than 1.4× when measured through the API-layer functions.

Table 1 provides another relevant comparison by combining (i) architectures; (ii) single-threaded or multithreaded execution[5]; and (iii) API-layer functions, while also including `liboqs`[6] for comparison. Both libraries were compiled with the most suitable settings for each architecture.

Table 1. Median execution times (μs) and speedup relative to the baseline of each experimental block for API-layer.

Backend	Algorithm	Thread Type	Keypair (μs)	Speedup	Encaps. (μs)	Speedup	Decaps. (μs)	Speedup
Block A: (baseline `libcesarpq-1.5.1-x86-64-ref`, ML-KEM-512, single-thread)								
`libcesarpq-1.5.1-x86-64-ref`	**ML-KEM-512**	**single (baseline)**	**23.213**	**1.000**	**24.966**	**1.000**	**26.997**	**1.000**
`libcesarpq-1.5.1-x86-64-ref`	ML-KEM-512	multi	23.018	1.008	24.928	1.002	27.016	0.999
`liboqs-v0.15.0-x86-64-ref`	ML-KEM-512	native	24.215	0.959	26.631	0.937	26.359	1.024
Block B: (baseline `libcesarpq-1.5.1-aarch64-native`, ML-KEM-512, single-thread)								
`libcesarpq-1.5.1-aarch64-native`	**ML-KEM-512**	**single (baseline)**	**4.792**	**1.000**	**5.625**	**1.000**	**5.875**	**1.000**
`libcesarpq-1.5.1-aarch64-native`	ML-KEM-512	multi	4.750	1.009	5.417	1.038	5.708	1.029
`liboqs-v0.15.0-aarch64-native`	ML-KEM-512	native	7.459	0.642	8.500	0.662	9.875	0.595
Block C: (baseline `libcesarpq-1.5.1-x86-64-ref`, ML-KEM-1024, single-thread)								
`libcesarpq-1.5.1-x86-64-ref`	**ML-KEM-1024**	**single (baseline)**	**59.769**	**1.000**	**63.411**	**1.000**	**64.263**	**1.000**
`libcesarpq-1.5.1-x86-64-ref`	ML-KEM-1024	multi	36.947	1.618	41.459	1.529	45.213	1.421
`liboqs-v0.15.0-x86-64-ref`	ML-KEM-1024	native	59.847	0.999	63.226	1.003	60.978	1.054
Block D: (baseline `libcesarpq-1.5.1-aarch64-native`, ML-KEM-1024, single-thread)								
`libcesarpq-1.5.1-aarch64-native`	**ML-KEM-1024**	**single (baseline)**	**10.333**	**1.000**	**11.417**	**1.000**	**13.208**	**1.000**
`libcesarpq-1.5.1-aarch64-native`	ML-KEM-1024	multi	10.167	1.016	11.416	1.000	13.167	1.003
`liboqs-v0.15.0-aarch64-native`	ML-KEM-1024	native	18.459	0.560	19.750	0.578	23.000	0.574

In Table 1 (Block A), as expected, there is almost no difference between the single-threaded and multithreaded versions of `libcesarpq` for ML-KEM-512 on `x86-64-ref`, consistent with Figure 3(a). Its performance is also very similar to that of `liboqs`, since both use the same backend, namely `mlkem-native`.

For `x86-64-ref` with ML-KEM-1024, shown in Table 1 (Block C), the multithreaded version of `libcesarpq` achieves a speedup of about 1.6× over the single-threaded version, as already discussed in Figure 3(c). This gain is explained by the OpenMP-based parallelization of `mlk_gen_matrix`. As expected, `liboqs` performs similarly to the single-threaded `libcesarpq`, but it is outperformed by its multithreaded version.

For `aarch64-native`, Table 1 (Blocks B and D), the single-threaded and multithreaded versions of `libcesarpq` show similar behavior, in agreement with Figures 3(b) and (d). However, `libcesarpq` is approximately 50% to 80% faster than `liboqs`. This gap is explained by backend specialization and by the build options exposed by each library. In `libcesarpq`, ML-KEM provides a dedicated `aarch64+sha3` path, compiled with `-march=armv8.4-a+sha3` and selected at runtime when SHA3 instruction set architecture (ISA) support is available; in addition, the

[5]In single-thread mode, OpenMP is disabled. In multithreaded mode, `libcesarpq` automatically defines the number of threads to be used.

[6]For `liboqs`, version `0.15.0` was used, as it was the latest stable release at the time this paper was written (https://github.com/open-quantum-safe/liboqs/releases/tag/0.15.0).

release build applies native tuning and interprocedural optimization (IPO)/link-time optimization (LTO). For `liboqs`, we used the best configuration available without source-code changes, enabling its `aarch64-native` ML-KEM path. However, `liboqs` does not currently expose an equivalent `aarch64+sha3` backend selectable only through compilation directives, which explains the observed performance gap.

Finally, it is important to highlight the significant performance improvement observed when architecture-specific optimized implementations are available. In this particular case, the optimized implementation used on `aarch64-native` is 4× to 5× faster than the reference-oriented implementation used on `x86-64-ref`. This result should be interpreted with caution, since it reflects a specific experimental setting and may vary according to factors such as processor architecture, microarchitectural characteristics, compiler optimizations, available instruction-set extensions, and processor model. Even so, it clearly indicates that optimized, architecture-aware implementations can substantially outperform generic reference implementations.

4.3. Performance of the batched API-layer

This section analyzes the batched API-layer. As discussed in Section 3.3, nested parallelism is intentionally disabled, so the batched API-layer invokes the API-layer in single-threaded mode and applies OpenMP only at the upper layer.

Figure 4 presents the batched API-layer benchmark results for ML-KEM-512 and ML-KEM-1024 on `x86-64-ref` and `aarch64-native` under different thread counts. The reported values correspond to the speedup relative to the single-threaded baseline. For each combination of (i) architecture; (ii) thread count; and (iii) batched API-layer function (i.e., batched keypair generation, batched encapsulation, and batched decapsulation), the experiment was repeated 5,000 times after a warm-up phase, and the median execution time was used to compute the speedup. Each benchmark repetition processes a batch of 256 calls to the underlying API-layer; that is, each batch performs 256 executions of keypair generation, encapsulation, or decapsulation, depending on the function being evaluated.

Figure 4 shows similar behavior within the same architecture. In particular, Figure 4(a) and (c), corresponding to ML-KEM-512 and ML-KEM-1024 on `x86-64-ref`, show near-linear speedup, reaching about 12× with 12 threads. According to Equation 2, this indicates that, within the evaluated range, the reduction of the useful-work term, $\frac{T_s}{p}$, remains dominant over the overhead term $T_{oh}(p)$.

In contrast, Figure 4(b) and (d), corresponding to `aarch64-native`, initially show an approximately linear trend, but the speedup soon stabilizes. With 10 threads, the speedup ranges from about 2× to 4× for ML-KEM-512 and from about 4× to 5× for ML-KEM-1024. In terms of Equation 2, this saturation indicates that, as p increases, the reduction in $\frac{T_s}{p}$ is progressively offset by the growth of $T_{oh}(p)$.

This effect is associated with the shorter execution time of the batched operations on `aarch64-native`, which implies a smaller T_s. As a result, the overhead term $T_{oh}(p)$ becomes comparatively more significant as the number of threads p increases, thereby limiting the achievable speedup. This phenomenon is more pronounced for ML-KEM-512 than for ML-KEM-1024 on the same architecture, since the former involves less useful work and is therefore more strongly affected by the relative weight of $T_{oh}(p)$. A

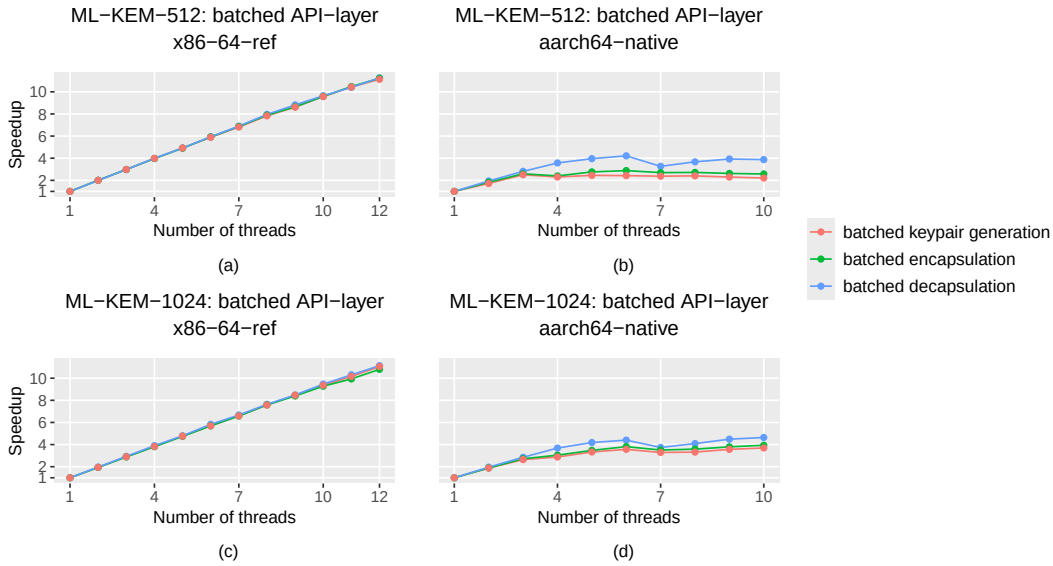

Figure 4. Benchmark results for the batched API-layer of ML-KEM-512 and ML-KEM-1024 on the `x86-64-ref` and `aarch64-native` architectures.

similar effect would also be expected on the `x86-64-ref` architecture; however, within the evaluated range, the larger absolute execution time of the functions implies a larger T_s, so the term $\frac{T_s}{p}$ still dominates over $T_{oh}(p)$, and the saturation effect is not observed.

As discussed in Section 4.2, Table 2 provides another relevant comparison by combining (i) architectures; (ii) single-threaded and multithreaded execution modes; and (iii) batched API-layer functions, while also including `liboqs`. Both libraries were compiled with the most suitable configuration for each target architecture.

To enable a clearer comparison of the batched API-layer results, we adopted as baseline the single-threaded version of `libcesarpq`, with all OpenMP-based parallelization disabled, including at the batched API-layer. We then compared it with three alternatives: (i) a single-threaded version of `libcesarpq` with externally implemented OpenMP-based batching[7]; (ii) a multithreaded version of `libcesarpq` using the internally implemented extended batched API; and (iii) `liboqs` with externally implemented OpenMP-based batching, so as to ensure a fair comparison between the two libraries.

Table 2 can likewise be analyzed by architecture. For `x86-64-ref`, `libcesarpq` and `liboqs` exhibit similar performance, as expected, since both rely on the same `mlkem-native` backend. Moreover, consistently with Figure 4, an approximate 12× speedup is achieved, in line with the near-linear scaling observed up to the 12 available threads.

For `aarch64-native`, the results differ between security levels. For ML-KEM-512, the observed speedup ranges from about 2× to 4×. Although ML-KEM-512 on `aarch64-native` is faster in `libcesarpq` than in `liboqs` in the non-batched baseline comparison (Table 1), the OpenMP overhead in the batched scenario offsets much of this advantage. According to Equation 2, this occurs because the useful-work

[7]In this scenario, we implemented the batching manually in the test code using OpenMP using the single-threaded version of `libcesarpq`.

Table 2. Median execution times per execution (μs) and speedup relative to the baseline of each experimental block for batched API-layer.

Backend	Algorithm	Thread Type	Keypair (μs)	Speedup	Encaps. (μs)	Speedup	Decaps. (μs)	Speedup
Block A: (baseline `libcesarpq-1.5.1-x86-64-ref`, ML-KEM-512, single-thread)								
`libcesarpq-1.5.1-x86-64-ref`	**ML-KEM-512**	**single (baseline)**	**23.349**	**1.000**	**26.023**	**1.000**	**29.382**	**1.000**
`libcesarpq-1.5.1-x86-64-ref`	ML-KEM-512	single (OpenMP external)	2.069	11.284	2.328	11.176	2.591	11.339
`libcesarpq-1.5.1-x86-64-ref`	ML-KEM-512	multi	2.085	11.198	2.290	11.364	2.586	11.360
`liboqs-v0.15.0-x86-64-ref`	ML-KEM-512	native (OpenMP external)	3.316	7.040	3.766	6.909	2.537	11.580
Block B: (baseline `libcesarpq-1.5.1-aarch64-native`, ML-KEM-512, single-thread)								
`libcesarpq-1.5.1-aarch64-native`	**ML-KEM-512**	**single (baseline)**	**4.659**	**1.000**	**5.607**	**1.000**	**5.874**	**1.000**
`libcesarpq-1.5.1-aarch64-native`	ML-KEM-512	single (OpenMP external)	1.992	2.339	2.085	2.689	1.589	3.697
`libcesarpq-1.5.1-aarch64-native`	ML-KEM-512	multi	2.137	2.180	2.182	2.570	1.467	4.003
`liboqs-v0.15.0-aarch64-native`	ML-KEM-512	native (OpenMP external)	1.986	2.346	2.309	2.428	2.531	2.321
Block C: (baseline `libcesarpq-1.5.1-x86-64-ref`, ML-KEM-1024, single-thread)								
`libcesarpq-1.5.1-x86-64-ref`	**ML-KEM-1024**	**single (baseline)**	**60.573**	**1.000**	**65.270**	**1.000**	**71.791**	**1.000**
`libcesarpq-1.5.1-x86-64-ref`	ML-KEM-1024	single (OpenMP external)	5.285	11.462	5.733	11.385	6.279	11.434
`libcesarpq-1.5.1-x86-64-ref`	ML-KEM-1024	multi	5.467	11.079	6.013	10.855	6.414	11.193
`liboqs-v0.15.0-x86-64-ref`	ML-KEM-1024	native (OpenMP external)	5.772	10.493	6.161	10.592	6.011	11.943
Block D: (baseline `libcesarpq-1.5.1-aarch64-native`, ML-KEM-1024, single-thread)								
`libcesarpq-1.5.1-aarch64-native`	**ML-KEM-1024**	**single (baseline)**	**10.169**	**1.000**	**11.462**	**1.000**	**13.273**	**1.000**
`libcesarpq-1.5.1-aarch64-native`	ML-KEM-1024	single (OpenMP external)	2.820	3.607	2.979	3.847	3.178	4.176
`libcesarpq-1.5.1-aarch64-native`	ML-KEM-1024	multi	2.721	3.737	2.916	3.931	2.759	4.811
`liboqs-v0.15.0-aarch64-native`	ML-KEM-1024	native (OpenMP external)	4.189	2.427	4.694	2.442	5.274	2.517

term, $\frac{T_s}{p}$, becomes relatively small, while $T_{oh}(p)$ becomes comparatively more significant. By contrast, for ML-KEM-1024, the larger execution time keeps $\frac{T_s}{p}$ significant relative to $T_{oh}(p)$, so parallel execution still provides a clear benefit. As a result, `libcesarpq` remains slightly faster, achieving speedup from 3.6× to 4.8×, whereas `liboqs` achieves around 2.5×.

The main takeaway in this scenario concerns product-engineering advantages. With `libcesarpq` in multithreaded mode, the user does not need to implement, maintain, and validate a custom parallelization layer: the library already provides built-in parallelism with a consistent API and well-defined behavior. This reduces development effort, lowers the risk of implementation errors, and decreases maintenance costs while preserving competitive performance.

5. Security considerations

In the context of this work, the use of OpenMP only modifies the execution strategy of the implementation. As such, it does not, by construction, introduce additional side effects beyond those inherent to the original sequential algorithm. The implications of this design choice are discussed in the following.

In the core-layer, parallelization is applied to loops whose iterations operate on disjoint memory regions, preserving polynomial computation independence and avoiding concurrent writes to the same destination. In particular, matrix generation and polynomial/vector NTT-domain operations are index-partitioned so each thread processes independent blocks, while seeds, parameters, and lookup tables are read-only. Therefore, assuming no aliasing among output buffers, parallel execution preserves the same semantics as serial execution.

In the batched API layer, parallelism is applied across full ML-KEM instances (keypair generation, encapsulation, and decapsulation functions) that are independent by construction. Each loop iteration computes its own publick key, private key, ciphertext,

and shared secret offsets and calls the scalar API on non-overlapping slices; thus, no cryptographic state is shared across batch instances. The only shared aggregate in the loop is the return code, handled with OpenMP reduction, which prevents a data race on that variable. In addition, the backend dispatcher is initialized once in a constructor and then only read during parallel calls.

The security justification for OpenMP is therefore conditional on explicit assumptions, consistent with the `libcesarpq` design: i) buffers passed to parallel calls must be disjoint per instance/iteration; ii) shared inputs (e.g., reference public keys or input seeds) must remain read-only during parallel regions; iii) custom extensions (e.g., backend hooks, custom RNG, or external callbacks) must preserve thread safety; and iv) parallel computations at the iteration level operate only on their own inputs and produce outputs without reading or modifying the state of other iterations.

Regarding the risk of running two threads on the same physical core, `libcesarpq` itself enforces thread-placement policy in its thread initializer and, in automatic mode, caps thread count to detected physical cores. Under this operational model, two worker threads are not intended to be scheduled on the same physical core, reducing intra-core contention and constraining side-channel surface related to shared pipeline/L1 resources under Simultaneous Multithreading (SMT). In summary, under the above conditions, OpenMP parallelization is compatible with the implementation's security requirements.

6. Conclusion

In this work, we presented `libcesarpq`, an OpenMP-based cryptographic library for ML-KEM, and evaluated different parallelization strategies across the core-layer, API-layer, and batched API-layer. Our results showed that the impact of parallelism depends strongly on the granularity of the computation: while most internal functions are too short to benefit significantly from fine-grained OpenMP parallelization, the extended batched API-layer enables the library to exploit parallelism much more effectively, yielding the most consistent speedups. When compared with `liboqs`, `libcesarpq` achieved competitive performance overall and outperformed it in some scenarios, especially on optimized platforms. These results indicate that the proposed extended API is particularly advantageous for high-throughput use cases, as it combines usability with efficient batch execution while avoiding nested parallelism overhead. Finally, the paper analyzes the security implications of the proposed OpenMP-based modifications, highlighting that the proposed optimizations preserve the algorithm's role as a practical and secure building block for post-quantum cryptography.

Acknowledgments

This work has been funded by the project PQC Optimization for Critical Infrastructures supported by Centro Integrado de Segurança em Sistemas Avançados (CISSA), with financial resources from the PPI IoT/Manufatura 4.0 of the MCTI grant number 08/2-24, signed with EMBRAPII.

References

[Abbasi et al. 2025] Abbasi, M., Cardoso, F., Váz, P., Silva, J., and Martins, P. (2025). A practical performance benchmark of post-quantum cryptography across heterogeneous computing environments. *Cryptography*, 9(2):32. pages 7

[Avanzi et al. 2019] Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., Stehlé, D., et al. (2019). Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 2(4):1–43. pages 2

[Azevedo et al. 2025] Azevedo, B. L., Lagrota, V., and Ribeiro, M. V. (2025). Multicore implementation of ml-kem on embedded devices. In *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*, pages 675–691. SBC. pages 3

[Becker et al. 2022] Becker, H., Hwang, V., Kannwischer, M. J., Yang, B.-Y., and Yang, S.-Y. (2022). Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 221–244. pages 2

[Bos et al. 2018] Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehlé, D. (2018). Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European symposium on security and privacy (EuroS&P)*, pages 353–367. IEEE. pages 2

[Chen et al. 2016] Chen, L. et al. (2016). Report on post-quantum cryptography. Technical Report 8105, NIST, Gaithersburg, MD. pages 1

[Fredrickson et al. 2003] Fredrickson, N. R., Afsahi, A., and Qian, Y. (2003). Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 140–149. pages 8

[Gewehr et al. 2024] Gewehr, C., Luza, L., and Moraes, F. G. (2024). Hardware acceleration of crystals-kyber in low-complexity embedded systems with risc-v instruction set extensions. *IEEE Access*, 12:94477–94495. pages 2

[Grover 1996] Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proc. 28th annual ACM symposium on Theory of computing*, pages 212–219. pages 1

[Kim and Seo 2025] Kim, Y. and Seo, S. C. (2025). An optimized instantiation of post-quantum mqtt protocol on 8-bit avr sensor nodes. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, pages 248–266. pages 2

[Lagrota et al. 2022] Lagrota, V., Camponogara, Â., López, J., and Ribeiro, M. V. (2022). The feasibility of the crystals-kyber scheme for smart metering systems. *IEEE Access*, 10:131303–131317. pages 2

[Lagrota et al. 2025] Lagrota, V., Filomeno, M. d. L., de MBA Dib, L., López, J., and Ribeiro, M. V. (2025). A quantum-resistant advanced metering infrastructure. In *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*, pages 32–48. SBC. pages 2

[Langlois and Stehlé 2015] Langlois, A. and Stehlé, D. (2015). Hardness of the learning with errors problem over rings and modules. In *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 464–494. Springer. pages 2

[Nguyen et al. 2025] Nguyen, T.-H., Dang, T.-K., Dam, D.-T., Nguyen, K.-D., Duong, P.-P., Pham, C.-K., and Hoang, T.-T. (2025). An area-time efficient hardware architecture for ml-kem post-quantum cryptography standard. *IEEE Access*. pages 2

[Ni et al. 2025] Ni, Z., Khalid, A., Liu, W., and O'neill, M. (2025). A highly hardware efficient ml-kem accelerator with optimised architectural layers. *ACM Transactions on Embedded Computing Systems*, 24(2):1–24. pages 2

[NIST 2024] NIST (2024). Module-lattice-based key-encapsulation mechanism standard (fips 203). Federal Information Processing Standards Publication FIPS 203, National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA. Published August 13, 2024. pages 2

[Pacheco and Malensek 2021] Pacheco, P. and Malensek, M. (2021). *An introduction to parallel programming*. Morgan Kaufmann. pages 3

[Paiva et al. 2025] Paiva, T. B., Simplicio Jr, M. A., Hafiz, S. M., Yildiz, B., Cominetti, E. L., and Ogawa, H. S. (2025). Tailorable codes for lattice-based kems with applications to compact ml-kem instantiations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(3):139–163. pages 2

[Regev 2005] Regev, O. (2005). On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 84–93. ACM. pages 2

[Shor 1994] Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. 35th Annu. Symp. Foundations of Computer Science*, pages 124–134. pages 1

[Wei et al. 2026] Wei, H., Li, W., Shen, S., Yang, H., and Zhao, Y. (2026). Optimized implementation of ml-kem on armv9-a with sve2 and sme. *Cryptology ePrint Archive*. pages 2