Multicore Implementation of ML-KEM on Embedded Devices

Beatriz L. Azevedo[1], **Vinícius Lagrota**[2], **Moisés V. Ribeiro**[1]

[1] Electric Engineering Department
Federal University of Juiz de Fora (UFJF) – Juiz de Fora, MG – Brazil

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

beatriz.azevedo@estudante.ufjf.br, vinicius.lagrota@gmail.com,
moises.ribeiro@ufjf.br

Abstract. *This paper introduces a cost-effective function-selection procedure, anchored in Amdahl's Law, to orchestrate selective parallelization of Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM) scheme on a dual-core ESP32 microcontroller. The procedure quantifies each function's single-core runtime and the overhead of parallel execution to isolate routines whose concurrent execution yields net time savings, while excluding low-latency tasks. The resulting ESP32 implementation eliminates separate single- and dual-core code paths, simplifies development, and achieves substantial reductions in overall runtime. Experimental evaluation on a dual-core ESP32 microcontroller of ML-KEM confirms the procedure's suitability for latency-critical applications.*

1. Introduction

The increasing connectivity among sensors and devices, driven by significant advances in the Internet of Things (IoT) technologies, has also heightened awareness of cyber-attack risks, making data protection a matter of national security[Nagrare et al. 2023, Jurcut et al. 2020]. In this context, cryptographic protocols play a crucial role in securing digital communications by ensuring the confidentiality, integrity, and authenticity of information [Stallings 2017]. Widely used schemes, such as Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC), rely on mathematical problems considered secure against conventional attacks. However, advances in quantum computing pose a threat to the security of these schemes, as sufficiently powerful quantum computers could break them in reduced time using techniques such as Shor's algorithm [Shor 1997]. Consequently, extensive research has been conducted to develop cryptographic schemes that remain secure even against adversaries equipped with such computers.

Recognizing the importance of these advancements, the National Institute for Standards and Technology (NIST) has been leading a standardization process for post-quantum cryptography (PQC) schemes, highlighting ML-KEM, standardized in FIPS-203 [NIST 2024b], as the primary key encapsulation algorithm due to its robustness and foundation on the Module-Learning With Errors (M-LWE) problem [Chen et al. 2016]. ML-KEM was designed to deal with embedded and resource-constrained microcontrollers, such as the ESP32 microcontroller; however, its implementation needs to be carefully designed to deal with latency-sensitive applications. There are a few works (e.g., [Huang et al. 2024, Kim et al. 2024]), in which the Number Theoretic Transform (NTT) is designed to provide additional acceleration on polynomial multiplications used in ML-KEM. Considering low-cost platforms, [Junior and Henriques 2022, Ferro et al. 2021] demonstrated that performance optimizations in memory allocation and the selection of mathematical operations can enable PQC schemes, such as ML-KEM, to reduce computation time while maintaining efficiency and security. However, investigations into how to leverage multi-core microcontrollers, such as the ESP32, have not yet been conducted.

This study proposes the dual-core implementation of ML-KEM scheme, more specifically, ML-KEM-512, on an ESP32 microcontroller by proposing a cost-effective function selection procedure grounded in Amdahl's Law. In this procedure, we measure the single-core execution time of each function and quantify the overhead incurred by parallelization across both cores. By systematically comparing these values, the procedure isolates functions whose parallel execution yields net execution time gains. This procedure eliminates the need for separate single- and dual-core code paths, streamlining development and reducing engineering effort. Moreover, the procedure extends directly to ML-KEM-768 and ML-KEM-1024 (and other PQC schemes), providing a practical framework for optimizing latency-sensitive applications.

The remainder of this paper is organized as follows: Section 2 introduces the fundamental concepts; Section 3 details the function selection procedure; Section 4 discusses implementation details; Section 5 presents a comparative analysis; and Section 6 concludes with final remarks.

2. Fundamentals

With the advent of quantum computing, the security of classical cryptographic systems has become increasingly at risk. In 1994, Peter Shor developed an algorithm capable of factoring integers and solving the discrete logarithm problem in polynomial time, rendering classical cryptographic systems—widely used today—vulnerable in a quantum era, as they rely on the computational difficulty of mathematical problems [Shor 1994], such as integer factorization. Although large-scale quantum computers are not yet available, the threat has already materialized through the harvest now, decrypt later (HNDL) technique, which consists of an attack where adversaries intercept and store encrypted data transmitted over networks today, with the expectation that quantum computing will allow them to decrypt it in the future [Mosca 2018]. This type of attack demonstrates how information that is currently protected could be compromised in the future, affecting financial transactions, government data, and long-term classified information [Bernstein and Lange 2017].

In response to this reality, the NIST launched a standardization process for PQC algorithms in 2016, aiming to replace vulnerable cryptographic systems with secure alternatives that can withstand attacks from quantum computers [NIST 2016]. This process was divided into two main categories:

- **Key Encapsulation Mechanisms**: Cryptographic schemes that enable the secure exchange of symmetric keys between two parties.
- **Digital Signatures**: Algorithms that ensure the authenticity and integrity of digital documents.

After years of rigorous analysis and testing, the NIST announced in 2022 the selection of four algorithms for standardization, leading to the publication of FIPS-203, FIPS-204, FIPS-205, and FIPS-206, with the latter still in the documentation process [NIST 2024c, NIST 2024a, NIST 2024]. The CRYSTALS-Kyber (standardized as ML-KEM under FIPS-203 [NIST 2024b]) was chosen as the primary standard for secure key exchange, while the CRYSTALS-Dilithium, Falcon, and SPHINCS+ algorithms were selected as standards for digital signatures.

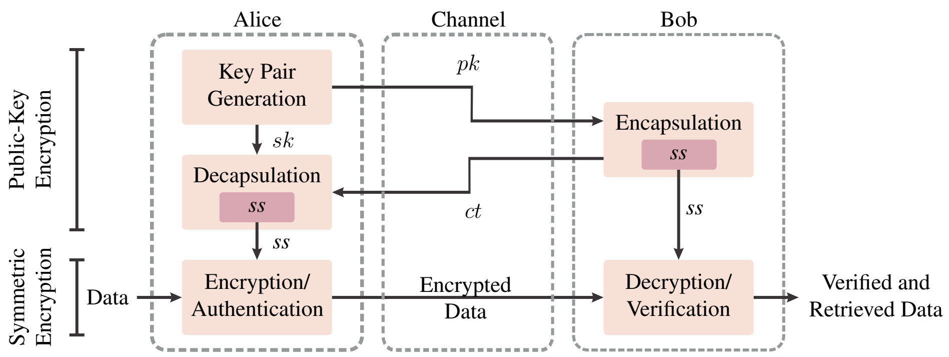
2.1. Key Encapsulation Mechanism

Key Encapsulation Mechanism (KEM) is a widely adopted solution to ensure confidentiality during key exchange. Its efficiency and simplicity easily allow the establishment of shared keys without the need for the explicit exchange of secrets between two parties. KEM is an asymmetric cryptographic scheme commonly used to securely exchange keys between two parties, making it particularly relevant in scenarios that require high security, such as post-quantum scenarios.

An illustration of the KEM process is shown in Figure 1. This process can be described in three main stages [Lagrota et al. 2022], as outlined below:

1. **Key Pair Generation**: This is the initial stage, in which Alice generates a key pair consisting of a public key pk and a private key sk. The public key is securely shared with Bob, while the private key remains securely stored with Alice.
2. **Encapsulation**: Using Alice's public key, Bob generates a random secret key ss and encapsulates it using the public key pk. The result of this process is a ciphertext ct, which is essentially the secret key encrypted with the public key pk.
3. **Decapsulation and Secret Key Retrieval**: Alice uses her private key sk to decapsulate the ciphertext ct, recovering the shared secret key ss. The mathematical primitive used must ensure that only the holder of the private key can access ss, even if ct is intercepted by third parties.

Figure 1. Illustration of the KEM process.



The secret key ss generated by a KEM enables the use of a symmetric cryptographic scheme for data exchange, as it is more efficient for this purpose. This is because symmetric cryptographic schemes, such as Advanced Encryption Standard (AES), offer superior performance in encrypting large volumes of data, while KEM ensures secure key distribution without transmitting the secret or key directly [Lagrota et al. 2022]. However, it is important to highlight that KEM, by itself, does not provide direct protection against quantum computers. While it is an efficient mechanism for key encapsulation, its security depends on the underlying cryptographic primitive. Thus, it represents a viable solution for securing key exchange in the quantum era when combined with post-quantum cryptographic primitives, as these are specifically designed to withstand the aforementioned threats.

2.2. ML-KEM

ML-KEM represents an innovative advancement in the field of PQC, as its security foundation relies on the M-LWE problem. This structure employs modular lattices to ensure a high level of security, offering resistance against both classical and quantum computer attacks [NIST 2024d, Bos et al. 2018]. This mathematical choice reinforces the resilience of the cryptographic scheme and positions it as a viable solution in the face of future computational advancements, especially considering that potential vulnerabilities are more likely to emerge in schemes based on Ring-Learning With Errors (R-LWE) than in those based on M-LWE [Bos et al. 2021]. Unlike many post-quantum cryptographic schemes that impose heavy computational burdens, ML-KEM is specifically designed for efficient implementation on embedded and resource-constrained platforms. Additionally, the low cycle count and energy efficiency of ML-KEM make it ideal for battery-powered or energy-harvesting devices. These characteristics position ML-KEM as a strong candidate for deployment in the IoT and satellite communication.

Based on the Module-Learning With Errors (LWE) problem, ML-KEM offers lower computational complexity than other LWE schemes. In addition, to maximize its computational efficiency, ML-KEM integrates polynomial vectors with the NTT, optimizing critical operations such as modular multiplications and reducing complexity from $O(n^2)$ to $O(n \log n)$. ML-KEM offers flexible security levels with compact key and ciphertext sizes, corresponding to ML-KEM-512 (NIST Level 1), ML-KEM-768 (NIST Level 3), and ML-KEM-1024 (NIST Level 5). These levels are configured by setting the parameter `KYBER_K` to 2, 3, and 4, respectively, providing security comparable to AES-128, AES-192, and AES-256. This flexibility, combined with NIST standardization, allows the cryptographic scheme to be adapted to different security requirements without compromising its core structure.

Regarding its structure, ML-KEM follows the same three classic stages of a KEM: generation of a key pair (public and private)—represented by the `keygen` function; encapsulation of a secret key using the public key—represented by the `encap` function; and decapsulation of this secret key by the receiving party using its private key—represented by the `decap` function, as defined in the decapsulation step of ML-KEM specification [NIST 2016]. Additionally, the cryptographic scheme provides implementations that utilize SHAKE or AES-256 for pseudo-random generation of public keys, offering greater flexibility in configuring security parameters [Lagrota et al. 2022].

Another advantage related to ML-KEM is its implementations with constant-time execution and fixed-point arithmetic, which is essential for preventing timing attacks and ensuring compatibility with microcontrollers like the ARM Cortex-M4 [Kannwischer et al. 2019]. ML-KEM also features a compact memory footprint, requiring less than 4 kB of stack on common microcontrollers [Kannwischer et al. 2019, Kannwischer et al. 2020]. In the context of practical implementations, ML-KEM demonstrates its potential in resource-constrained devices, such as ARM Cortex M4 [Abdulrahman et al. 2025] and ESP32 microcontroller [Segatz and Hafiz 2025].

Due to the hardware constraints of embedded devices—such as low power consumption, limited memory, and reduced processing capability—and the high computational cost of certain ML-KEM operations, it becomes necessary to identify critical points where parallelization techniques can be applied. In the case of the ESP32 microcontroller, it is worth mentioning that its dual-core architecture and *FreeRTOS* support provide advantages for implementing ML-KEM. However, the memory constraints (520 kB of SRAM) and clock limitations (up to 240 MHz) highlight the opportunity to leverage both cores to reduce the execution time of functions in ML-KEM, as detailed in Section 3.

2.2.1. ML-KEM Functions

For effectively leveraging the dual-core architecture of the ESP32 microcontroller requires a detailed understanding of the internal structure of ML-KEM to guide parallelization efforts strategically. In this context, its modular design of plays a fundamental role.

Figure 2 illustrates the modular organization of ML-KEM scheme, which is structured in layers that reflect both the functional decomposition of the algorithm and its potential for parallelization. At the top of the hierarchy, the KEM block encapsulates the full set of cryptographic operations and ensures indistinguishability under chosen ciphertext attack (IND-CCA) by introducing mechanisms such as ciphertext verification and re-encryption checks. These components are essential in real-world applications to ensure robustness against adaptive adversaries.

Directly beneath it lies the indistinguishability under chosen plain text attack (IND-CPA) block, which implements the core logic of key generation, encapsulation, and decapsulation. This layer is responsible for constructing the main cryptographic functionality of the scheme. It leverages structured polynomial operations to maintain confidentiality and relies heavily on data transformations built upon polynomial vectors.

Supporting the IND-CPA logic are the **Polynomial vectors** and **Polynomials** layers, which correspond to the `polyvec` and `poly` modules, respectively. These layers are tightly connected, as the `polyvec` module operates on vectors of polynomials, each of which is handled individually by the `poly` module. The `polyvec` layer is of particular interest in this work, as its structure naturally supports parallel execution: each polynomial in a vector can be processed independently, enabling effective distribution across cores. Typical operations at this level include applying the NTT, compression, modular multiplication, and vector addition. The independence of these computations means they can be parallelized with minimal synchronization overhead.

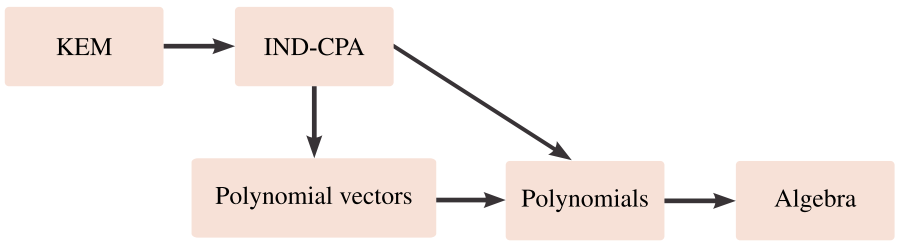
In contrast, although the `poly` module—represented in the **Polynomials** layer—is responsible for low-level operations on individual polynomials, its functions tend to be too fine-grained for parallelization to yield practical performance gains. Parallelizing at this level often results in excessive overhead from task management, negating any speedup. As such, parallelization efforts are best concentrated at the `polyvec` level.

At the bottom of the hierarchy lies the **Algebra** block, which consists of the basic arithmetic operations that act on polynomial coefficients. These foundational routines support the higher layers but are themselves not directly targeted for optimization in this study.

The modular decomposition, shown in Figure 2, guides the selective parallelization strategy adopted in this work. By focusing on layers where computational workloads are naturally independent—particularly within the `polyvec` module — it becomes possible to reduce execution time without incurring the penalties associated with unnecessary synchronization at lower abstraction levels.

Based on this modular decomposition, specific functions were selected for analysis according to their suitability for parallel execution. Functions from the `polyvec` module were prioritized because operations on individual polynomials are naturally independent, allowing effective distribution of tasks across cores with minimal synchronization overhead. Functions from the `IND-CPA` module were selectively included where it was possible to parallelize isolated segments, such as noise generation, without introducing correctness risks. Additionally, functions from the `poly` module were evaluated to demonstrate that parallelization at too fine a granularity, such as the individual polynomial level, often leads to excessive task management overhead and performance degradation. This selection strategy ensures that the study focuses on parallelization opportunities that are both practical and impactful within the dual-core environment.

Figure 2. Illustration of ML-KEM.



Among the functions in the `polyvec` module, the following are noteworthy:

- `polyvec_ntt` and `polyvec_invntt_tomont`: apply the NTT and its inverse, respectively, to all polynomials in the vector.
- `polyvec_basemul_acc_montgomery`: multiplies and accumulates the coefficients of the polynomials stored in two `polyvec` vectors. This function is essential for key generation and encapsulation.
- `polyvec_compress` and `polyvec_decompress`: handle the compression and decompression of polynomial vectors.
- `polyvec_reduce`: performs modular reduction of polynomial coefficients to ensure they remain within the valid range for subsequent operations.
- `polyvec_add`: computes the element-wise addition of two polynomial vectors.
- `polyvec_tobytes` and `polyvec_frombytes`: convert polynomial vectors to and from their binary representations, ensuring compatibility with storage and transmission formats.

In addition to the functions in the `polyvec` module, functions from the `IND-CPA` and `poly` modules were also analyzed. In the `IND-CPA` module, `gen_matrix`, `indcpa_keypair_derand`, and `indcpa_enc` were selected, with only parallelizable segments modified in the last two. This is because other functions in `IND-CPA` exhibit strong sequential dependencies or frequent shared data access, making them unsuitable for parallel processing. Conversely, the analysis of the `poly` module was conducted to show that parallelizing at lower levels introduces significant overhead, ultimately degrading performance rather than improving it. As such, the following functions from `IND-CPA` were selected:

- `gen_matrix`: generates the public matrix using pseudorandom coefficients, with independent processing for each entry.
- `indcpa_enc`: performs key encapsulation. To avoid dependency issues, only noise generation (`indcpa_noise_eta`) was tested.
- `indcpa_keypair_derand`: optimizes noise generation in key pair creation. Similar to `indcpa_enc`, only its noise generation function (`indcpa_noise_keypair`) was parallelized.

In the `poly` module, the `poly_tomont` function was selected for analysis due to its role in converting polynomial coefficients to the Montgomery representation, which is essential for modular arithmetic operations and for reducing the protocol's execution time.

3. Function Selection Procedure

In embedded devices, it is crucial to adopt strategies that reduce computational cost without compromising the effectiveness of the cryptographic scheme. In the case of ML-KEM, such strategies are built upon three main pillars: (i) restructuring the code to eliminate unnecessary operations (e.g., having a different NTT that demands less computational complexity, which can be efficiently performed on an ESP32 microcontroller; (ii) leveraging specialized processor instructions, commonly available in architectures like x86 (AVX) and ARM (Neon), that support single instruction, multiple data (SIMD) operations, which are not available on the ESP32 microcontroller; and (iii) implementing a *multicore* execution model that distributes the workload across available processing cores, which is feasible given the ESP32's dual-core architecture.

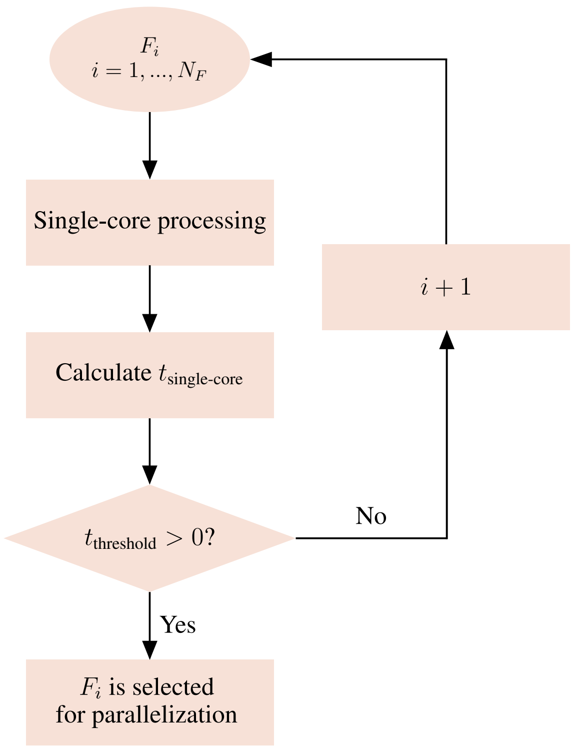
Considering the third pillar, this section details a selection procedure for the efficient parallelization of functions within ML-KEM on a dual-core ESP32 microcontroller. We also introduce a criterion that allow us select the functions offering advantages when they are parallelized. In Subsection 3.1, we detail the flow-chart of the function selection procedure to show the benefits of parallelizing functions of ML-KEM for reducing their execution time. In the sequel, Subsection 3.2 shows how to obtain a threshold time that for performing a decision-making in favor or not of parallelization of functions in a dual-core ESP32 microcontroller.

3.1. Flow Chart

To effectively leverage the dual-core architecture of the ESP32 microcontroller, we propose a procedure based on a simple yet effective criterion to identify which ML-KEM functions can attain less execution time when its processing is parallelized.

Figure 3 shows the flow-chart for this procedure. In this model, we assume that there are F_i, $i = 1, \cdots, N_F$ functions in ML-KEM. According to this flow-chart, the ith function is solely executed in a single-core of the ESP32 microcontroller and its execution time ($t_{\text{single-core}}$) is obtained. In the following, the execution time $t_{\text{single-core}}$ is compared to a quantity to generate the so-called threshold time $t_{\text{threshold}}$. This threshold time represents the minimum single-core execution time required for a function to benefit from parallelization. It is derived by accounting for the overhead introduced by task management and synchronization, and is further discussed in Section 3.2. If $t_{\text{threshold}} > 0$, then the ith function is marked as suitable for parallelization, and its execution across both cores is advantageous for reducing the total execution time. Otherwise, the ith function remains assigned to a single-core execution, thereby avoiding the performance degradation by unnecessary parallelizing functions. Overall, this procedure serves not only as a practical guideline for implementing efficient parallelism but also as a safeguard against applying parallelization strategies that, despite being theoretically promising, fail to deliver benefits in resource-constrained or real-time execution scenarios.

Figure 3. The procedure flow-chart


F_i
$i = 1, ..., N_F$
Single-core processing
Calculate $t_{\text{single-core}}$
$t_{\text{threshold}} > 0$?
No
$i + 1$
Yes
F_i is selected for parallelization

3.2. Threshold Time ($t_{\text{threshold}}$)

Evaluating the benefits of parallelizing a task requires defining a threshold execution time that justifies the overhead incurred when managing *tasks* in a dual-core implementation. In this context, the threshold time ($t_{\text{threshold}}$) at which parallelization becomes advantageous depends on two key parameters: (i) the execution time of a task on a single core, denoted by $t_{\text{single-core}}$, and (ii) the overhead time required for creating, synchronizing, and terminating *tasks* in a dual-core setup, denoted by t_{overhead}.

While Amdahl's Law informs the ideal speedup of a *multicore* implementation under the assumption of zero overhead time [Hill and Marty 2008], we adapt it to include the overhead time as follows:

$$S \triangleq \frac{1}{(1-P)+\frac{P}{N}+\epsilon}, \tag{1}$$

in which S is the speedup, $P \in \mathbb{R}|\, 0 \leq P \leq 1$ is the fraction of the task that can be parallelized, $N \in \mathbb{N}_+$ is the number of cores, and $\epsilon \in \mathbb{R}_+$ denotes the fraction of execution time consumed by the overhead associated with using N cores. Assuming $N = 2$, $P = 1$, and $\epsilon = \frac{1}{2}$, the resulting speedup is $S = 1$, indicating that parallelization yields no performance gain. Thus, knowing the overhead time t_{overhead} is essential for determining a threshold time $t_{\text{threshold}}$ for selecting a task for the execution in parallel.

Now, let us consider a task from ML-KEM whose execution time on a single core is $t_{\text{single-core}}$. The execution time on a dual-core is then given by

$$t_{\text{dual-core}} = \frac{t_{\text{single-core}}}{2} + t_{\text{overhead}}, \tag{2}$$

where the term $\frac{t_{\text{single-core}}}{2}$ represents the parallel execution of two halves of the task (i.e., $P = 1$), and t_{overhead} is the overhead for using two cores. Based on these quantities, we can also obtain the speedup using the following expression:

$$S = \frac{1}{1-\beta}, \tag{3}$$

in which

$$\beta = \frac{t_{\text{single-core}} - t_{\text{dual-core}}}{t_{\text{single-core}}} \tag{4}$$

is named execution time gain. A positive β indicates a reduction in execution time, while $\beta \leq 0$ indicates no gain or a slowdown.

Figure 4 illustrates the relationship among $t_{\text{single-core}}$, $t_{\text{dual-core}}$, and t_{overhead}. For parallelization to be beneficial, the following condition must be satisfied:

$$t_{\text{single-core}} \geq t_{\text{dual-core}}. \tag{5}$$

Substituting (2) into (5) yields

$$t_{\text{single-core}} \geq 2t_{\text{overhead}}. \tag{6}$$

This inequality leads to the definition of the threshold time, expressed as

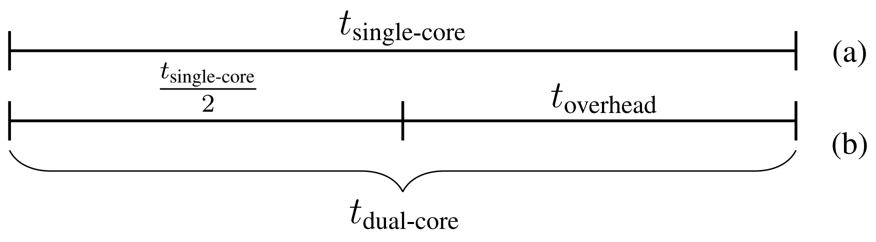
$$t_{\text{threshold}} \triangleq t_{\text{single-core}} - f(t_{\text{overhead}}), \tag{7}$$

where $f(t_{\text{overhead}}) = 2t_{\text{overhead}} + \gamma$, and $\gamma \in \mathbb{R}_+$ is an optional offset that accounts for additional design considerations (e.g., development time for implementing parallelization). In this study, we assume $\gamma = 0$. Thus, if $t_{\text{threshold}} > 0$, parallelizing the task is advantageous; otherwise, if $t_{\text{threshold}} \leq 0$, parallelization does not provide execution time benefits.

Based on experimental results, we observe that the speedup (3), execution time gain (4), and threshold time (7) can all be used to identify functions that benefit from

Figure 4. Illustration of the relationship between $t_{\text{single-core}}$, $t_{\text{dual-core}}$, and t_{overhead}.

(a) Execution time on a single-core. (b) Execution time on dual-core.



parallelization. However, it is important to emphasize that the threshold time is significantly more cost-effective because it requires knowledge only of $t_{\text{single-core}}$ and $f(t_{\text{overhead}})$, whereas the other metrics require both $t_{\text{single-core}}$ and $t_{\text{dual-core}}$. Obtaining $t_{\text{dual-core}}$ for each function is considerably more complex, as it requires developing code for both dual-core and single-core execution modes.

It is important to analyze how the parameter `KYBER_K` influences the speedup. When `KYBER_K` is even—enabling a balanced distribution of tasks on a dual-core (i.e., $P = 1$ and $N = 1$)—a speedup approaching $S = 2$ is expected if $t_{\text{single-core}} \gg 2t_{\text{overhead}}$ (i.e., $\epsilon \to 0$, meaning the impact of the parallelization overhead becomes negligible. However, when `KYBER_K` is odd, one core must process a portion of the task sequentially, which prevents full parallelization and results in a lower speedup. For instance, in the specific case of `KYBER_K` $= 3$, two of the three operation sets can be executed in parallel, while the third must be processed separately, resulting in $P = \frac{1}{2}$. Consequently, the speedup tends to $S = \frac{4}{3}$ if $t_{\text{single-core}} \gg 2t_{\text{overhead}}$ (i.e., $\epsilon \to 0$). One can note that `KYBER_K` $= 4$, we consider $N = 2$ and $P = 1$, and, consequently, the speedup approximates $S = 2$ when $t_{\text{single-core}} \gg 2t_{\text{overhead}}$ (i.e., $\epsilon \to 0$).

In general, the threshold time $t_{\text{threshold}}$ serves as a useful parameter for evaluating the effectiveness of task parallelization when accounting for the overhead associated with task creation and synchronization.

3.2.1. Threshold Time Value

The determination of t_{overhead} requires precise measurement of execution time, as synchronization latency between processor cores can vary depending on the architecture, operating system, and other factors specific to the execution environment. To estimate this value, we conducted an experiment in which two empty *tasks* were created and assigned to different cores of the ESP32 microcontroller. This setup allows the exclusive measurement of synchronization cost. The execution time was recorded from the initiation to the completion of both tasks, yielding a value of $t_{\text{overhead}} = 68\ \mu s$. Consequently, the minimum threshold for a function to benefit from parallel execution on the ESP32 is given by

$$t_{\text{threshold}} = t_{\text{single-core}} - 136\ \mu\text{s}. \tag{8}$$

Note that (8) enables evaluation and prediction of the system's behavior when distributing *tasks* across dual cores. However, in practice, execution time is also affected by other variables, such as inter-task interference and hardware-specific synchronization overheads.

To further validate this approach, an additional experiment was conducted using the `polyvec_add` function, which was selected due to its short runtime ($t_{\text{single-core}} \approx 47.3$ μs) and tunable computational complexity. This function is employed in the case of `KYBER_K` $= 2$, for which we assume $P = 1$. The experiment consisted of repeating the `polyvec_add` function $n = 1, \ldots, 30$ times in both single-core and dual-core configurations and measuring their corresponding execution times, denoted $t_{\text{single-core}}$ and $t_{\text{dual-core}}$, respectively. The execution time gain was then computed by

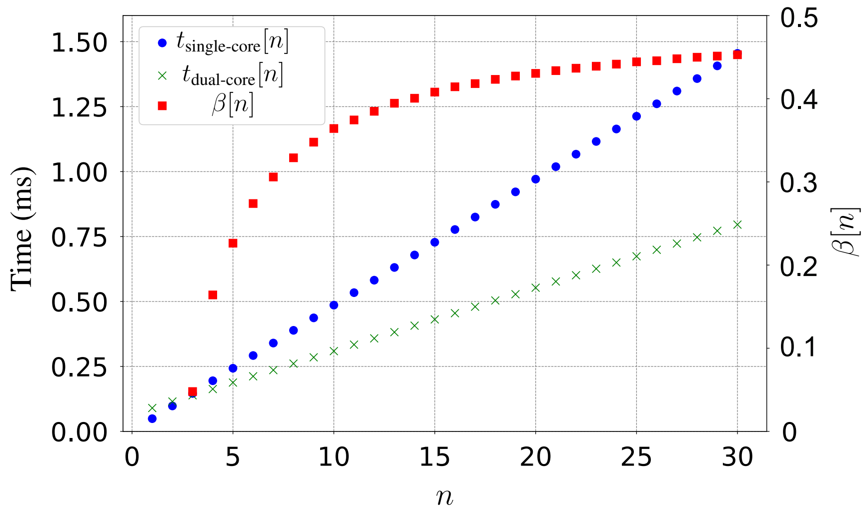
$$\beta[n] = \frac{\Delta t_{\text{single-core}}[n] - \Delta t_{\text{dual-core}}[n]}{\Delta t_{\text{single-core}}[n]}, \tag{9}$$

where $\Delta t_{\text{single-core}}[n]$ denotes the execution time of the nth repetition in the single-core setting, and $\Delta t_{\text{dual-core}}[n]$ refers to the corresponding time in the dual-core setting. A value of $\beta[n] > 0$ indicates a performance gain through parallelization, while $\beta[n] \leq 0$ implies no gain or a performance degradation.

Figure 5 presents the curves of $\Delta t_{\text{single-core}}[n]$, $\Delta t_{\text{dual-core}}[n]$, and $\beta[n]$. The intersection of $\Delta t_{\text{single-core}}[n]$ and $\Delta t_{\text{dual-core}}[n]$ occurs around $\hat{t}_{\text{threshold}} \approx 142$ μs, identifying the point beyond which parallel execution becomes advantageous. The difference $t_{\text{threshold}} - \hat{t}_{\text{threshold}} = -6$ μs does not indicate a real discrepancy but is attributed to the measurement precision limitation for the `polyvec_add` function, whose execution time increases with computational complexity. Nonetheless, this experiment supports the appropriateness of considering $t_{\text{threshold}} = 136$ μs.

Finally, this experiment demonstrates that the execution time gain $\beta[n]$ asymptotically approaches a value close to 0.5 as $n \to \infty$. Thus, the parallel speedup can be expressed as $S = \lim_{n\to\infty} \frac{1}{1-\beta[n]}$. Given that $\lim_{n\to\infty} \beta[n] \approx 0.47$, and assuming $N = 2$ and $P = 1$, manipulation of (1) yields $\epsilon = 0.03$.

Figure 5. Execution time analysis of the `polyvec_add` function for `KYBER_K` $= 2$.



4. Implementation Details

This section presents the architectural and system-level strategies adopted in the parallel implementation of ML-KEM scheme on the dual-core ESP32 platform. The design aimed to minimize overhead, promote deterministic execution, and maintain reproducibility within the limitations of embedded systems. The implementation was built on *FreeRTOS*, the real-time operating system natively supported by the ESP32. To ensure predictable performance and avoid the latency associated with dynamic memory management, all parallel tasks are allocated statically at system startup.

For each function targeted for parallelization, such as `polyvec_ntt` and `polyvec_basemul_acc_montgomery`, two tasks were instantiated and permanently pinned to core 0 and core 1, respectively. These tasks are reused across all parallel operations to further reduce runtime overhead. To enable concurrent processing with minimal synchronization, we partition the data structures—primarily polynomial vectors—evenly between the two cores. This approach allows each task to operate independently on disjoint memory regions, thereby eliminating the need for mutual exclusion mechanisms.

The synchronization is handled using binary semaphores, which offer a lower overhead for signaling between independent cores. A central control task orchestrates the execution flow: it first signals both worker tasks to begin a parallel operation and then waits to receive a completion signal from each. This protocol guarantees that both workers have finished before the main program continues, ensuring algorithmic correctness.

The decision to use semaphores rather than `mutexes` was intentional. While mutexes are appropriate for protecting shared resources, semaphores offer lower overhead and are better suited for signaling across independent processing units. In this context, semaphores provided an efficient and reliable means of coordinating task execution between the two cores.

5. Comparative Analysis

Aiming to validate the effectiveness of the proposed function selection criterion, this section discusses the experimental results obtained from the implementation of ML-KEM-512 (`KYBER_K` $= 2$) on the ESP32 microcontroller when non-selective and the proposed function selection procedures are considered. All performance metrics reported in this section were obtained through 101 repeated executions of each operation on the ESP32 platform running *FreeRTOS*. The median execution time was adopted to mitigate the effects of sporadic variations in task scheduling and clock drift. In this sense, Subsection 5.1 shows how the function selection procedure succeed to take advantage of dual-core. In the sequel, Subsection 5.2 focuses the gains obtained when the proposed function selection procedure is applied. Our implementation can be found in [Azevedo and Lagrota 2025].

5.1. Using the Function Selection Procedure

This subsection evaluates the execution time of the functions from Section 2.2.1 for the case `KYBER_K` $= 2$ in ML-KEM scheme, implemented both on a single-core (without parallelization) and on dual-core (with parallelization). Table 1 summarizes, in milliseconds, the values of $t_{\text{single-core}}$, $t_{\text{dual-core}}$, $t_{\text{threshold}}$, corresponding execution time gain β and speedup value S for each function.

We can see that functions with index < 6 (i.e., $t_{\text{threshold}} < 0$) show no benefit from parallelization, which is confirmed by $\beta \leq 0$, and, consequently, $S \leq 0$. These include `polyvec_from_bytes`, `polyvec_add`, `polyvec_to_bytes`,

Table 1. Performance comparison: wo/ and w/ parallelization.

Index	Function	$t_{\text{single-core}}$	$t_{\text{dual-core}}$	$t_{\text{threshold}}$	β	S
1	`polyvec_frombytes`	0.048	0.106	< 0	-1.208	0.453
2	`polyvec_add`	0.049	0.106	< 0	-1.163	0.462
3	`polyvec_tobytes`	0.073	0.113	< 0	-0.548	0.646
4	`poly_tomont`	0.065	0.097	< 0	-0.492	0.670
5	`polyvec_decompress`	0.103	0.106	< 0	-0.029	0.972
6	`polyvec_reduce`	0.120	0.120	$= 0$	0.000	1.000
7	`polyvec_compress`	0.179	0.156	> 0	0.128	1.147
8	`polyvec_basemul_acc_montgomery`	0.448	0.341	> 0	0.239	1.314
9	`indcpa_noise_eta`	1.074	0.638	> 0	0.406	1.684
10	`polyvec_ntt`	0.802	0.471	> 0	0.413	1.704
11	`indcpa_noise_keypair`	1.363	0.787	> 0	0.423	1.733
12	`gen_matrix`	2.049	1.163	> 0	0.432	1.761
13	`polyvec_invntt_tomont`	1.210	0.666	> 0	0.450	1.818

`poly_tomont`, `polyvec_decompress`, and `polyvec_reduce`. For example, `polyvec_add` grows from 0.049 ms (single-core) to 0.106 ms (dual-core), and `polyvec_decompress` from 0.103 ms to 0.106 ms—showing that synchronization overhead dominates any parallel gain.

In contrast, functions with index > 6 (i.e., $t_{\text{threshold}} > 0$)—those exceeding 136 μs on single-core—do benefit from dual-core execution. Specifically, `polyvec_compress`, `polyvec_basemul_acc_montgomery`, `indcpa_noise_eta`, `polyvec_ntt`, `indcpa_noise_keypair`, `gen_matrix`, and `polyvec_invntt_tomont` exhibit $\beta > 0$, and, consequently $S > 1$. Here, the execution time is sufficient to amortize overhead time, yielding clear performance improvements. Moreover, as $t_{\text{single-core}} \gg 2\,t_{\text{overhead}}$, we observe $\beta \to 0.5$, in agreement with the theoretical speedup, which is obtained from (1) when $N = 2$, $P = 1$, and $\epsilon \to 0$.

In summary, only functions whose single-core runtime exceeds the time threshold ($t_{\text{threshold}} > 0$) achieve meaningful acceleration under dual-core execution. However, setting $\gamma \neq 0$ can prevent the parallelization of functions that offer only marginal execution time gains, thereby reducing the overall effort required to develop parallelized code.

5.2. Performance Validation

To validate the effectiveness of the proposed function selection procedure, we perform a performance comparison between the three main routines of the Application Programming Interface (API) of ML-KEM—`keygen`, `encap`, and `decap`—responsible for key pair generation, encapsulation, and decapsulation, respectively. These routines are among the most computationally intensive operations in ML-KEM protocol, as they heavily rely on a specific set of functions detailed in Section 2.2.1. Since the functions within this set are central to the overall performance of the algorithm, their behavior has a profound impact on the execution time of the entire system. Furthermore, the routines `keygen`, `encap`, and `decap` are fundamental to ML-KEM workflow, and their performance directly influences the efficiency of the cryptographic operations. In this sense, the following implementations of these routines are considered:

1. **Implementation Reference (Impl. Ref.)**: all functions in the routines are executed on a single-core, serving as the reference for performance comparison. The

execution time is denoted by $t_{\text{Impl. Ref.}}$.

2. **Implementation** $\#2$ **(Impl.** $\#1$**)**: all functions in the routines are executed on a dual-core, allowing us to evaluate the effects of indiscriminate parallelization. The execution time is denoted by $t_{\text{Impl. }\#1}$.
3. **Implementation** $\#3$ **(Impl.** $\#2$**)**: refers to the use of the proposed function selection procedure to decide the functions in the routine that are executed on a dual-core, while the remaining functions are executed on a single-core. According to Table 1 in Subsection 5.1, the functions with indexes between 1 and 6 are executed on single-core, while the functions with indexes between 7 and 13 are executed on dual-core. The execution time is denoted by $t_{\text{Impl. }\#2}$.

Table 2 presents the execution times of `keygen`, `encap`, and `decap` routines, denoted by t_{keygen}, t_{encap}, and t_{decap}, respectively. It also includes the total execution time for each implementation, which is given by

$$t_{\text{total}} = t_{\text{keygen}} + t_{\text{encap}} + t_{\text{decap}}. \tag{10}$$

Performance is compared using the execution time gain for these routines, which is adapted from (4) and given by

$$\beta_{\#i} = \frac{t_{\text{Impl. Ref.}} - t_{\text{Impl. }\#i}}{t_{\text{Impl. Ref.}}}. \tag{11}$$

where, $i \in \{1, 2\}$. Moreover, we calculate the resulting speedup as follows:

$$S_{\#i} = \frac{1}{1 - \beta_{\#i}}. \tag{12}$$

All execution times in Table 2 are given in milliseconds. **Impl.** $\#1$ achieves a substantial speedup over **Impl. Ref.**, as evidenced by $S_{\#1}$. **Impl.** $\#2$ delivers a further, albeit smaller, gain relative to **Impl.** $\#1$, as indicated by $S_{\#2} \gtrsim S_{\#1}$. Overall, the results in Table 2 validate the proposed Function Selection Procedure and underscore the importance of selecting the appropriate functions for parallelization in the routines aiming to minimize the total execution time of ML-KEM scheme. Moreover, they emphasize that experimental evaluation on the target multi-core (i.e., EXP32 microcontroller) is mandatory, as the choice of which functions to parallelize has a significant impact on performance improvements.

Table 2. Performance comparison between the three implementations.

Description	**Impl. Ref.**	**Impl.** $\#1$	**Impl.** $\#2$	$\beta_{\#1}$	$\beta_{\#2}$	$S_{\#1}$	$S_{\#2}$
t_{keygen}	7.460	5.273	5.141	0.293	0.311	1.414	1.451
t_{encap}	8.990	6.468	6.352	0.281	0.293	1.391	1.414
t_{decap}	11.214	8.240	8.120	0.265	0.276	1.361	1.381
t_{total}	27.665	19.993	19.621	0.277	0.291	1.383	1.410

6. Conclusion

This work focused on a dual-core implementation of ML-KEM-512 scheme (`KYBER_K` $= 2$) on an ESP32 microcontroller, driven by a cost-effective function selection procedure based on single-core execution time and parallelization overhead. By eliminating the need for distinct single- and dual-core code, our proposal streamlines development and reduces engineering effort. We also highlighted how Amdahl's Law can guide this investigation. Experimental results on a dual-core ESP32 microcontroller confirm that the procedure effectively identifies functions that benefit most from parallel execution, yielding substantial latency reductions, which can be useful for IoT devices.

Overall, the proposed function selection procedure cost-effectively ensures that the benefits of parallelization outweigh the overhead associated with task management and synchronization related to parallelization.

Although the approach proved effective for the target use case, it is important to acknowledge its limitations. In cases where data dependencies are stronger or inter-core coordination becomes more complex, the model may not fully capture the real performance dynamics, and further prototyping would be necessary. The proposed methodology is best suited for functions that are trivially parallelizable and operate over independent memory segments. Future work could involve adapting this strategy for more complex cryptographic schemes or testing it on heterogeneous hardware architectures.

Acknowledgments

This research was supported in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) under Grant 001, Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under grants 445958/2024-3 and 314741/2020-8, and Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) under grant APQ-04623-22.

References

Abdulrahman, A., Kannwischer, M. J., and Lim, T.-H. (2025). Enabling microarchitectural agility: Taking ML-KEM & ML-DSA from cortex-m4 to m7 with SLOTHY. Cryptology ePrint Archive, Paper 2025/366.

Azevedo, B. and Lagrota, V. (2025). Dual core ML-KEM on ESP32. `https://github.com/beatrizufjf/Dual-core-ML-KEM-on-ESP32.git`.

Bernstein, D. J. and Lange, T. (2017). Post-quantum cryptography—dealing with the fallout of physics success. Cryptology ePrint Archive, Paper 2017/314.

Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehle, 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.

Bos, J. W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J., Schwabe, P., Stehlé, D., and Tibouchi, M. (2021). CRYSTALS-Kyber Algorithm Specifications and Supporting Documentation (Round 3). `https://pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf`. Accessed April 2025.

Chen, L., Jordan, S., Liu, Y.-K., Moody, D., Peralta, R., Perlner, R., and Smith-Tone, D. (2016). Report on Post-Quantum cryptography. Technical report, National Institute of Standards and Technology.

Ferro, L. F. C., Rampazzo, F. J. A., and Henriques, M. A. A. (2021). Estudos de otimização do algoritmo de criptografia pós-quântica crystals-kyber. In *Anais do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG)*.

Hill, M. D. and Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, 41(7):33–38.

Huang, J., Zhao, H., Zhang, J., Dai, W., Zhou, L., Cheung, R. C. C., Koç, c. K., and Chen, D. (2024). Yet another improvement of plantard arithmetic for faster kyber on low-end 32-bit iot devices. *IEEE Transactions on Information Forensics and Security*, 19:3800–3813.

Junior, G. G. and Henriques, M. A. A. (2022). Redução do consumo de memória no algoritmo de criptografia pós-quântica saber em microcontroladores arm cortex-m0+. In *Seminários em Engenharia de Computação e Sistemas Digitais*.

Jurcut, A., Niculcea, T., Ranaweera, P., and Le-Khac, N.-A. (2020). Security considerations for internet of things: A survey. *SN Comput. Sci.*, 1(4).

Kannwischer, M. J., Rijneveld, J., Schwabe, P., and Stoffelen, K. (2019). Efficient masking of Kyber on ARM cortex-m4. In *Proceedings of the 12th Workshop on Embedded Systems Security (WESS)*. ACM.

Kannwischer, M. J., Schwabe, P., and Stoffelen, K. (2020). pqm4: Testing and benchmarking nist pqc on the arm cortex-m4. In *Proceedings of CARDIS 2020: Smart Card Research and Advanced Applications*. Springer.

Kim, Y., Yoon, S., and Seo, S. C. (2024). Vectorized implementation of kyber and dilithium on 32-bit cortex-a series. *IEEE Access*, 12:104414–104428.

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.

Mosca, M. (2018). Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security & Privacy*, 16(5):38–41.

Nagrare, T., Sindhwad, P., and Kazi, F. (2023). BLE protocol in IoT devices and smart wearable devices: Security and privacy threats.

NIST (2016). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. `https://csrc.NIST.gov/projects/post-quantum-cryptography`. Accessed: 2024-12-30.

NIST (2024). Stateless hash-based digital signature standard. Technical report, Washington, D.C.

NIST, G. M. D. (2024a). Module-Lattice-Based digital signature standard. Technical report, Gaithersburg, MD.

NIST, G. M. D. (2024b). Module-lattice-based key-encapsulation mechanism standard. Technical report, Gaithersburg, MD.

NIST, G. M. D. (2024c). Module-lattice-based key-encapsulation mechanism standard. Technical report, Gaithersburg, MD.

NIST, G. M. D. (2024d). Module-lattice-based key-encapsulation mechanism standard. Technical report, Gaithersburg, MD.

Segatz, F. and Hafiz, M. I. A. (2025). Efficient implementation of CRYSTALS-KYBER key encapsulation mechanism on ESP32.

Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 124–134, Los Alamitos, CA. IEEE.

Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509.

Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice*. Pearson, Upper Saddle River, NJ, 7 edition.