electronics

MDPI

Article

An Optimized Cross-Platform PQC Implementation Using AVX2 Compiler Intrinsics

Charisee Zhi Ling Yip [1,*], Zhi Qian Seah [2] and Huaqun Guo [1]

[1] Singapore Institute of Technology, Infocomm Technology Cluster, Singapore 828608, Singapore; huaqun.guo@singaporetech.edu.sg
[2] Defence Science & Technology Agency, Singapore 109679, Singapore; szhiqian@dsta.gov.sg
* Correspondence: 2203215@sit.singaporetech.edu.sg

Abstract

Classical cryptographic standards such as RSA and ECC face existential threats from the advent of quantum computing, necessitating an urgent transition to quantum-resistant post-quantum cryptographic (PQC) algorithms. This article presents a cross-platform, AVX2-optimized implementation approach for PQC algorithms, achieving the performance optimization of platform-specific implementations and the portability of reference implementations. The core contribution is a systematic translation methodology that converts Linux AVX2 assembly routines written in AT&T syntax into C compiler intrinsics through an intermediate Intel syntax representation. This approach preserves 256-bit SIMD operations while enabling the optimized implementation to be compiled across supported operating environments. The methodology was applied to the ML-DSA algorithm, for which five AVX2 assembly routines were translated and integrated with the reference implementation, while an existing compiler-intrinsics implementation of ML-KEM was incorporated into the unified library. Functional validation was conducted using positive and negative test conditions, while cross-platform validation was performed by executing the same test program in both Windows and Linux environments and comparing the resulting outputs. Performance was evaluated using 20 runs of 300 iterations for each major cryptographic operation. The optimized implementation achieved more than threefold overall speed improvement for ML-DSA and more than twofold improvement for ML-KEM relative to their reference implementations, with selected operations exceeding fourfold improvement. The results demonstrate that compiler intrinsics can provide a practical approach for retaining AVX2-based performance optimization while improving the portability and integration of PQC implementations.

Keywords: post-quantum cryptography; AVX2; performance optimization; dilithium; kyber; compiler intrinsics

Check for updates

Academic Editors: Andreas Mauthe and Nai-Wei Lo

Received: 13 July 2026
Revised: 25 September 2026
Accepted: 20 October 2026
Published: 22 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Classical cryptography standards such as RSA and Elliptic Curve Cryptography (ECC) form the backbone of secure communications in government, defense, and commercial systems. However, they are becoming increasingly susceptible to quantum computing threats, as quantum algorithms, most notably Shor's algorithm [1], are theoretically capable of solving these problems in polynomial time, rendering these widely deployed standards fundamentally insecure in a post-quantum world [2].

Transitioning to the era of utilizing post-quantum cryptography (PQC) standards is inevitable [3–7]. However, the existence of such standards does not automatically translate to production-ready deployment. It presents its own set of challenges when it comes to ensuring that PQC implementations achieve high performance, are cross-platform, and serve as production-level, DevOps-pipelined cryptographic libraries to support organizational cryptographic needs.

While PQC open-source implementations such as Open Quantum Safe's liboqs library [8] exist and provide cross-platform access to optimized PQC implementations, some of these libraries integrate implementations originating from different upstream projects and optimization approaches depending on the supported algorithm and platform. This work therefore investigates an alternative compiler-intrinsics-based implementation approach, specifically by translating AVX2 assembly routines for ML-DSA into C compiler intrinsics and integrating the resulting implementation with ML-KEM into a unified dynamically linked library. The objective is to investigate whether this approach can retain AVX2 vector operations while reducing dependence on platform-specific assembly syntax and facilitating compilation across the evaluated Windows and Linux environments. Moreover, many platform-specific optimized implementations rely on Linux assembly, while portable C versions lack optimization, highlighting the challenge of achieving both performance and portability within a single implementation approach.

Therefore, this work evaluates a compiler-intrinsics-based approach for developing an optimized cross-platform PQC implementation for AVX2-capable x86 environments. In comparison with portable reference implementations, which prioritize portability and algorithmic clarity, the proposed implementation retains AVX2-based vectorization to improve execution performance. In comparison with platform-specific optimized implementations that rely on Linux assembly written in AT&T syntax, the proposed implementation translates the AVX2 routines into C compiler intrinsics, reducing dependence on platform-specific assembly and enabling compilation across the evaluated Windows and Linux environments.

The proposed implementation integrates ML-DSA [9] and ML-KEM [10] into a unified Dynamic Link Library (DLL), providing a consistent interface for the major cryptographic operations. The implementation is functionally validated against expected algorithm behavior and evaluated across Windows and Linux environments. Its execution performance is subsequently benchmarked against the corresponding reference implementations to quantify the performance benefits obtained through AVX2 vectorization.

The remainder of this paper is organized as follows. In Section 2, we review the importance of PQC and the need for optimized and cross-platform implementations. In Section 3, we present our problem statement in detail, followed by our proposed solution and its results and insights in Sections 4 and 5, respectively. Section 6 highlights the technical innovations of the proposed solution, and Section 7 presents the future work. Finally, we draw a conclusion in Section 8.

2. Literature Review

This section presents the growing need for PQC, the benefits of AVX2 vectorization, and cross-platform portability implementation.

2.1. Growing Need for PQC in Modern Systems

This section covers the rationale behind the growing need for PQC in modern systems and IT infrastructures.

2.1.1. Emerging Quantum Threat Landscape and the Critical Need for Proactive Defense

The emergence of quantum computing poses a danger to the security of traditional, widely used public-key cryptographic systems like RSA and ECC. Quantum algorithms like Shor's algorithm have the theoretical ability to efficiently crack these conventional cryptographic systems. Although current quantum computers do not yet possess a sufficient number of stable qubits to practically execute Shor's algorithm at the scale required to break real-world cryptographic keys, rapid advancements in quantum hardware indicate steady progress towards this capability. Jenefa et al. [11] demonstrated that PQC serves as a proactive defense approach for present and future communications to remain secure even in the presence of quantum computing capabilities.

In addition, the "harvest now, decrypt later" (HNDL) threat model [12] further emphasizes the importance and urgency of adopting quantum-resistant cryptography. Under the HNDL model, adversaries may collect and store encrypted data today with the intention of decrypting it in the future once quantum computing becomes sufficiently powerful. Relating to Mosca's Theorem [13], because migrating large-scale cryptographic infrastructures requires significant time and planning, organizations must take immediate action to begin preparing early for the adoption of quantum-safe cryptographic mechanisms to safeguard long-term data confidentiality.

2.1.2. Limitations of Current Cryptography Standards

Current cryptography standards mainly depend on how difficult it is to solve specific mathematical problems, such as the large prime factorization and discrete logarithm problem, which quantum computing can solve efficiently, undermining the long-term security of these algorithms. Azhari et al. [14] highlight that even robust algorithms like RSA and ECC are vulnerable to quantum attacks.

2.1.3. PQC and QKD as Responses to Quantum Computing Threats

There are two broad defensive strategies against quantum computing threats: post-quantum cryptography (PQC), which uses algorithms based on new mathematically hard problems that are resistant to quantum computing threats, as well as quantum key distribution (QKD) [15], which uses quantum-mechanical properties to distribute keys securely.

PQC is a more deployable and scalable option because it can be integrated into existing digital infrastructures, standardized software, and current communication protocols without requiring entirely new systems, allowing the preservation of current system architectures. QKD, on the other hand, is often seen as a useful option because its security goal is rooted in quantum physics rather than only the hardness of mathematical problems.

However, the National Security Agency (NSA) does not recommend QKD for protecting systems unless major limitations are overcome. The NSA states that QKD is only a partial solution, requires special-purpose equipment, raises infrastructure costs, is difficult to validate securely, and introduces insider threat and denial-of-service risks, while PQC is considered more cost-effective and easier to maintain [16].

2.1.4. Global PQC Adoption Efforts

Governments and standardization bodies, led by NIST and the Department of Defense (DoD), have been pushing for the transition towards PQC through structured frameworks and standardization processes. NIST first initiated the PQC standardization effort [17] in 2016 by launching a global competition aimed at the development of quantum-resistant cryptographic algorithms under two primary tracks, key encapsulation mechanisms and digital signature algorithms. After rounds of evaluation, cryptanalysis, and performance testing, NIST selected multiple algorithms, including CRYSTALS-Kyber for key estab-

lishment and CRYSTALS-Dilithium for digital signatures [18]. Both schemes rely on the hardness of module-lattice problems and were later standardized as ML-KEM (FIPS 203) and ML-DSA (FIPS 204), respectively. This global effort highlights the urgency of PQC adoption in critical infrastructures [19].

2.2. High-Performance Acceleration of Cryptographic Algorithms Using AVX2 Vectorization

This section provides a literature review of utilizing AVX2 vectorization for efficient performance of cryptographic algorithms.

2.2.1. Performance Improvements with AVX2 Vectorization

By leveraging parallel processing through AVX2 instruction sets, it can greatly increase the throughput of PQC algorithms. According to Gitonga et al. [20], optimizations make it possible to implement lattice-based schemes in real-world settings without compromising security, increasing the feasibility of PQC. This balances the increase in security needs without compromising on speed efficiencies.

2.2.2. SIMD Vectorization in PQC Workloads

Polynomial multiplication, Number Theoretic Transform (NTT), and hashing are among the computationally demanding PQC operations that are greatly accelerated by Single Instruction Multiple Data (SIMD) extensions like AVX2 and AVX-512. SIMD enables the processing of multiple data elements simultaneously, which is particularly useful for the modular arithmetic and ring-based computations that form a large part of lattice-based cryptography. Wang et al. [21] reported that AVX2-optimized implementations of polynomial ring multiplication can reduce clock cycles by up to 85%, enhancing throughput without compromising algorithmic integrity. Thus, SIMD addresses common PQC bottlenecks like coefficient-wise modular reduction and transform operations, improving overall cryptographic performance.

2.3. Cross-Platform Cryptographic Implementations

This section highlights the importance of cross-platform cryptographic implementations by looking at the disadvantages of platform-specific implementations and advantages of portable intrinsics-based implementations.

2.3.1. Drawbacks of Platform-Specific Implementations

Scalability and cross-architecture adoption in post-quantum systems face significant challenges when utilizing platform-specific cryptography implementations. Nejatollahi et al. [22] demonstrated that PQC algorithm deployment varied considerably across $\times$86, ARM, and embedded platforms due to differences in memory structures, instruction sets, and processing capacity. As a result, ensuring platform interoperability becomes increasingly difficult. Furthermore, it complicates the integration of PQC in multi-device ecosystems like consumer electronics and IoT networks, signifying the need for consistent, portable implementations that ensure cryptographic security across diverse platforms.

2.3.2. Importance of Portability in Production Environments

Portable cryptographic implementations are essential for consistent performance, maintainability, and interoperability across different hardware architectures. A research study by Li et al. [23] showcased that the portability of efficient cryptographic algorithms, such as SM4, depends on their capacity to operate seamlessly on both $\times$86 and ARM platforms and resource-constrained embedded systems. Furthermore, their AVX2-based software optimization approach demonstrated that it supports adaptation to future instruction sets like AVX-512 and NEON with enhanced processing efficiency. This flexibility

ensures that PQC solutions remain relevant across evolving processor ecosystems, improving integration in diverse environments.

2.3.3. Advantages of Intrinsics-Based Cryptographic Implementations

Compiler intrinsics maintain portability across architectures, enabling efficient execution. Intrinsic-based implementations result in less code duplication, simpler maintenance, and reliable performance across hardware platforms as compared to pure assembly. As demonstrated by Benson et al. [24], compiler intrinsics may greatly reduce code complexity and the need for numerous hardware-specific versions while achieving performance comparable to or better than platform-specific SIMD libraries. This approach makes it ideal for PQC systems that need scalable, cross-platform cryptographic acceleration, as it improves long-term maintainability.

3. Problem Statement

This section highlights specific challenges of current implementations.

3.1. Platform-Specific Optimized Implementation

Many systems used by organizations rely on Windows OS. However, existing PQC optimized implementations from official sources are written for and limited to use in Linux environments. This is because performance optimizations are mostly implemented using AVX2 assembly instructions written in AT&T syntax for Linux, which are incompatible with Windows assemblers that use Intel syntax. As a result, organizations operating on Windows infrastructure cannot directly leverage these optimized implementations.

3.2. Lack of Cross-Platform Codebase

The need to support multiple environments such as Windows and Linux calls for an implementation approach that works consistently across platforms. While it is technically possible to maintain separate codebases for each environment, this approach introduces practical problems. Separate code paths for different architectures significantly complicate development workflows and raise the risk of subtle implementation inconsistencies between platforms, which in a cryptographic context can have serious security implications.

4. Solution

This section describes in detail the implementation of the proposed solution to achieve the cross-platform AVX2-optimized ML-DSA algorithm. While the integrated library combines both ML-DSA and ML-KEM, the translation work presented in this paper focuses primarily on ML-DSA, as an AVX2 compiler-intrinsics implementation of ML-KEM was already available and is incorporated into the unified library.

4.1. Translation to Compiler Intrinsics

Compiler intrinsics provide a higher-level interface for accessing processor-specific instructions directly from C or C++ code without requiring explicit assembly language. In this work, AVX2 compiler intrinsics are used to represent the corresponding vector operations in the original AVX2 assembly routines, enabling the translated implementation to retain the intended AVX2 operations while being expressed in a C-based implementation. Figure 1. illustrates the workflow of translating the existing AVX2 implementation currently written in Linux .S assembly files into C compiler intrinsics. Arithmetic operations, polynomial transformations, and other AVX2 assembly routines were all translated into their appropriate compiler intrinsic equivalents (for example, vpaddd in assembly is equivalent to _mm256_add_epi32 in compiler intrinsics).

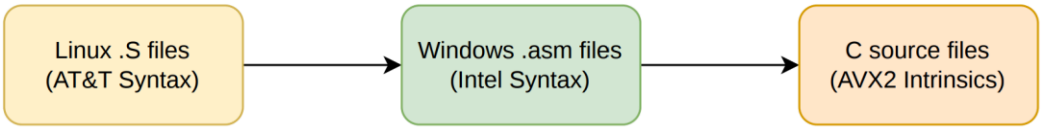


Figure 1. Translation workflow from Linux assembly to C intrinsics implementation.

Instead of direct translation from Linux assembly implementation into compiler intrinsics, the Linux .S assembly files written using AT&T syntax were first translated to the Windows .asm assembly files written using Intel syntax. This intermediate step helps to ensure consistent instruction representation and verifiable instruction behavior of major algorithm functions within assembly language. As both Windows assembly and C compiler intrinsics follow Intel syntax conventions, this alignment ensures a natural and unambiguous mapping between the two representations, reducing the risk of instruction misinterpretation during translation. The Windows assembly files are then used as a structured reference to map assembly instructions to compiler intrinsics, allowing for the final ML-DSA implementation to be in C compiler intrinsics.

During the entire translation process, maintaining the structural behavior of the original instructions is the most important. This involves making sure that the data is aligned correctly for 256-bit vector loading, preserving register usage patterns, and making sure that the operands are in the right sequence, especially for non-commutative arithmetic operations like subtraction and division. Since most of the assembly instructions had a close 1-to-1 mapping to their corresponding compiler intrinsics, maintaining this mapping was key to ensuring the correctness of the code and overall implementation.

4.2. Integration of Intrinsics Implementation

Once all assembly routines were successfully translated, the intrinsics implementation was added to the ML-DSA code structure as a whole. This was done by making sure that its function interfaces are the same as those used in the reference implementation. This ensured that the new AVX2 routines may be called by higher-level algorithm functions like key generation, signature generation, and signature verification, without needing to make major modifications to the main code flow or higher-level functions.

4.3. Ensuring Compliance with NIST PQC Standards

After developing a portable and optimized implementation of ML-DSA, the next step was to ensure that the parameters used in the code implementation and the function execution aligned with the NIST standard specifications accordingly, as seen in the FIPS 204 publication [9] for ML-DSA. Compliance was established by referencing to sections of the publication that described the differences and changes implemented between the original CRYSTALS-Dilithium submission and the final standardized ML-DSA algorithm. These were used to guide the necessary adjustments to ensure that the implementation accurately reflects the NIST-standardized version of the algorithm. Figure 2 illustrates one of the differences mentioned in the publication, which describes increasing the number of bytes used for two of the parameters in ML-DSA.

D.2 Differences Between Version 3.1 of CRYSTALS-DILITHIUM and FIPS 204 Initial Public Draft

ML-DSA increases the length of tr to 512 bits and increases the length of $\tilde{c}$ to 384 and 512 bits for the parameter sets ML-DSA-65 and ML-DSA-87, respectively. In draft ML-DSA, only the first 256 bits of $\tilde{c}$ are used in the generation of c.

Figure 2. Snippet of differences between the original submission and standardized ML-DSA (excerpt from FIPS 204 [9]).

5. Result and Insight

This section describes in detail the results of the solution implementation and the insights observed from the results achieved.

5.1. Assembly-to-Intrinsics Translation of Cryptographic Operations

All five assembly routines (ntt.S, invntt.S, pointwise.S, shuffle.S, and f1600x4.S) of ML-DSA were successfully translated from Linux .S assembly to Windows .asm assembly and finally to C compiler intrinsics. Figures 3–5 illustrate the translation of one of the assembly routines, ntt.S. The process begins with the original implementation in AT&T syntax (Figure 3), followed by its conversion into Intel syntax as ntt.asm (Figure 4), and concludes with its final translation into C using compiler intrinsics as ntt.c (Figure 5).

```
ntt.S
4   .macro butterfly l,h,zl0=1,zl1=1,zh0=2,zh1=2
5   vpmuldq     %ymm\zl0,%ymm\h,%ymm13   AT&T syntax: destination operand last
6   vmovshdup   %ymm\h,%ymm12
7   vpmuldq     %ymm\zl1,%ymm12,%ymm14
8
9   vpmuldq     %ymm\zh0,%ymm\h,%ymm\h
10  vpmuldq     %ymm\zh1,%ymm12,%ymm12
11
12  vpmuldq     %ymm0,%ymm13,%ymm13
13  vpmuldq     %ymm0,%ymm14,%ymm14
14
15  vmovshdup   %ymm\h,%ymm\h
16  vpblendd    $0xAA,%ymm12,%ymm\h,%ymm\h
17
18  vpsubd      %ymm\h,%ymm\l,%ymm12
19  vpaddd      %ymm\h,%ymm\l,%ymm\l
20
21  vmovshdup   %ymm13,%ymm13
22  vpblendd    $0xAA,%ymm14,%ymm13,%ymm13
```

Figure 3. Representative AVX2 assembly instructions in AT&T syntaxes.

```
ntt.asm
14  butterfly MACRO l, h, zl0:=<1>, zl1:=<1>, zh0:=<2>, zh1:=<2>
15      vpmuldq ymm13, ymm&h, ymm&zl0    Intel syntax: destination operand first
16      vmovshdup ymm12, ymm&h
17      vpmuldq ymm14, ymm12, ymm&zl1
18
19      vpmuldq ymm&h, ymm&h, ymm&zh0
20      vpmuldq ymm12, ymm12, ymm&zh1
21
22      vpmuldq ymm13, ymm13, ymm0
23      vpmuldq ymm14, ymm14, ymm0
24
25      vmovshdup ymm&h, ymm&h
26      vpblendd ymm&h, ymm&h, ymm12, 0AAh
27
28      vpsubd ymm12, ymm&l, ymm&h
29      vpaddd ymm&l, ymm&l, ymm&h
30
31      vmovshdup ymm13, ymm13
32      vpblendd ymm13, ymm13, ymm14, 0AAh
```

Figure 4. Corresponding Intel syntax representation used as an intermediate translation reference.

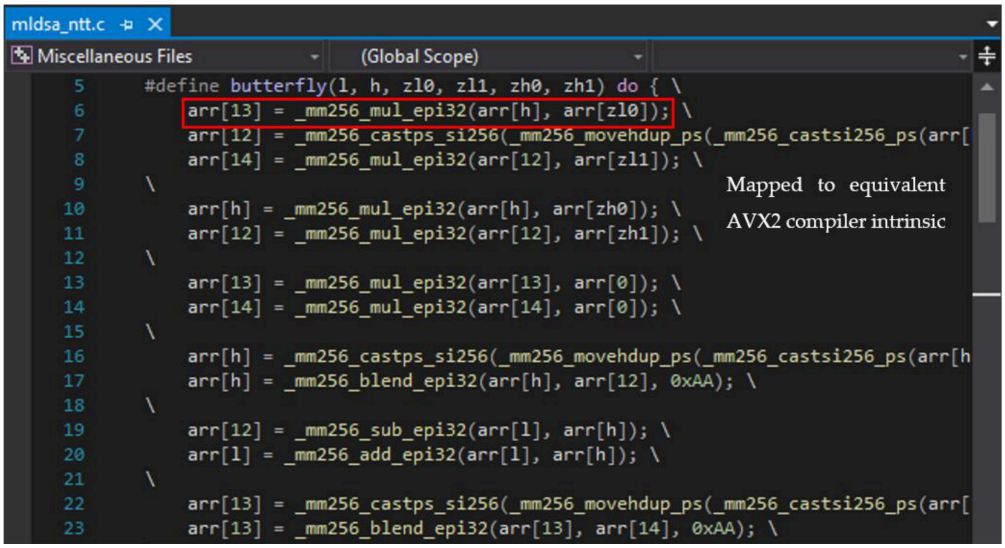


Figure 5. Representative C compiler-intrinsics implementation corresponding to the translated AVX2 operations.

5.1.1. Number Theoretic Transform (NTT) Routine (ntt.S)

The ntt.S routine implements the Number Theoretic Transform (NTT), which is used to perform efficient polynomial multiplication. During translation, a key aspect to focus on is the translation of the butterfly computation and ensuring correct alignment for 256-bit AVX2 vector operations.

5.1.2. Inverse NTT Routine (invntt.S)

The invntt.S routine performs the inverse NTT, mapping polynomials from the NTT domain back to the standard domain. The implementation preserves the inverse butterfly structure and ensures precise coefficient scaling, maintaining mathematical correctness while optimizing for SIMD throughput.

5.1.3. Pointwise Polynomial Multiplication (pointwise.S)

The pointwise.S routine performs element-wise modular multiplication of polynomial coefficients in the NTT domain. During translation, particular attention was given to preserving the structure of the operation and optimizing the interleaving of multiplication with Montgomery reduction to maximize SIMD lane utilization.

5.1.4. Vector Shuffle Operation (shuffle.S)

The shuffle.S routine reorders coefficients to satisfy the lane-crossing requirements of the NTT's butterfly computations. The translation focuses on using in-lane shuffles for early stages, and cross-lane permute intrinsics for later stages, ensuring coefficients are correctly paired across the full 256-bit register for parallel arithmetic.

5.1.5. Keccak-f[1600] Permutation (f1600x4.S)

The f1600x4.S routine implements the Keccak-f[1600] permutation in a four-way parallel configuration using AVX2 vectorization. This permutation forms the core of the SHAKE-based hash functions used in ML-DSA. The translation preserved the structure of the 24 permutation rounds and the parallel processing of four Keccak states, ensuring that the vectorized state transformations remained consistent with the original implementation.

5.2. Functional Algorithm Validation

A simple test program was written to validate the ML-DSA algorithm and ensure correct functional behavior with expected results obtained for each of the major algorithm functions. Negative test conditions like tampered signatures were also executed to put the algorithm to the test under undesirable input conditions. Bytes at different positions of the signature (ML-DSA signatures are a concatenation of 3 components—challenge polynomial, response vector, and hint vector) were randomly tampered with to ensure that the signature verification function can reject tampered signatures regardless of which signature component has been tampered. Edge cases like wrong message lengths or mismatched messages were also executed to validate the robustness of the algorithm. Figure 6. illustrates the negative test conditions and edge cases.

```
if(smlen != MLEN + CRYPTO_BYTES) {
    printf(stderr, "Signed message lengths wrong\n");
    return -1;
}
if(mlen != MLEN) {
    printf(stderr, "Message lengths wrong\n");
    return -1;
}
for(j = 0; j < MLEN; ++j) {
    if(m2[j] != m[j]) {
    printf(stderr, "Messages don't match\n");
    return -1;
    }
}

printf("Tampering in progress\n");
randombytes((uint8_t *)&j, sizeof(j));
do {
    randombytes(&b, 1);
} while(!b);
sm[j % (MLEN + CRYPTO_BYTES)] += b;
ret = crypto_sign_open(m2, &mlen, sm, smlen, ctx, CTXLEN, pk);
if(!ret) {
    printf(stderr, "Trivial forgeries possible\n");
    return -1;
}
```

Figure 6. Negative test scenarios.

Validating the functional behavior of the ML-DSA algorithm ensured that the functional correctness of the algorithm was maintained during the translation from assembly language to C compiler intrinsics.

5.3. NIST Known Answer Test Validation

To further validate the correctness of the implemented cryptographic algorithms, official NIST Known Answer Test (KAT) vectors were used for ML-DSA and ML-KEM. The implementations were validated against the expected outputs specified in the respective NIST test vectors for ML-DSA-87 and ML-KEM-1024, corresponding to NIST Security Level 5, the highest security level evaluated in this work. The validation was limited to these highest security-level parameter sets, while the remaining parameter sets are outside the scope of this study. The KATs verify the correctness of the implemented cryptographic operations against standardized reference outputs. The test results are summarized in Tables 1 and 2, with all tested cases successfully matching the expected outputs.

Table 1. Results summary of test case execution for ML-DSA.

Function	Total Number of Test Cases	Passed	Failed
Key Generation	25	25	0
Signature Generation	90	90	
Signature Verification	45	45	
Total	160	160	

Table 2. Results summary of test case execution for ML-KEM.

Function	Total Number of Test Cases	Passed	Failed
Key Generation	25	25	0
Encapsulation	25	25	
Decapsulation	10	10	
Total	60	60	

5.4. Cross-Platform Algorithm Validation

To evaluate cross-platform consistency, the same ML-DSA test program was compiled and executed in both Windows and Linux environments using the compiler-intrinsics implementation. The same algorithm functions and test inputs were used in both environments, and the resulting outputs were compared. Consistent outputs across the two environments were obtained for all three ML-DSA operations (key generation, signature generation, and signature verification), demonstrating that the translation to compiler intrinsics preserved the functional behavior and correctness of the implementation across the evaluated platforms. The validation therefore provides evidence of cross-platform functional portability between the evaluated Windows and Linux environments.

5.5. Dynamic Link Library (DLL) Cryptographic Implementation

To produce a compiled PQC library, the final output was to compile both the compiler-intrinsics versions of ML-DSA and ML-KEM algorithms into a single Dynamic Link Library (DLL), serving as a production-level cryptographic library to support various organizational cryptographic needs. This was achieved by configuring the project to build a Dynamic Link Library (.dll) and exporting the required functions using the "__declspec(dllexport)" attribute prior to compilation. As a result, the core functions for the PQC algorithms were exported as shown in Figure 7.

```
File Type: DLL

  Section contains the following exports for MLDSA-DLL.dll

    00000000 characteristics
    FFFFFFFF time date stamp
        0.00 version
           1 ordinal base
           6 number of functions
           6 number of names

    ordinal hint RVA      name

          1    0 00017290 pqcrystals_dilithium5_avx2_keypair = pqcrystals_dilithium5_avx2_keypair
          2    1 00018670 pqcrystals_dilithium5_avx2_sign = pqcrystals_dilithium5_avx2_sign
          3    2 00019090 pqcrystals_dilithium5_avx2_verify_signature = pqcrystals_dilithium5_avx2_verify_signature
          4    3 0001B410 pqcrystals_kyber1024_avx2_dec = pqcrystals_kyber1024_avx2_dec
          5    4 0001B220 pqcrystals_kyber1024_avx2_enc = pqcrystals_kyber1024_avx2_enc
          6    5 0001B110 pqcrystals_kyber1024_avx2_keypair = pqcrystals_kyber1024_avx2_keypair
```

Figure 7. PQC algorithm functions exposed via DLL.

A simple test program was used to validate that the ML-DSA key generation, signature generation, and signature verification functions, as well as the ML-KEM key generation, key encapsulation, and key decapsulation functions, were functioning as expected when

being utilized as external functions. This DLL ensures that the cryptographic algorithms can be easily implemented by various teams and programs without needing to dabble in the detailed code and its complexities.

5.6. Performance Benchmarking of Algorithm Execution Speed

Performance benchmarking of algorithm execution speed was conducted on both ML-DSA and ML-KEM algorithms to assess and quantify speed improvements achieved by the optimized AVX2 implementations in comparison to their reference implementations. This ensures the provision of a comprehensive performance evaluation of the unified library.

Each algorithm function was executed with 20 runs of 300 iterations each, and the average execution time for each major algorithm function and the whole algorithm was recorded. The average results were used for performance benchmarking to minimize the impact of external factors such as system latency and any computational background processes. We executed our AVX2 proposed implementation and reference implementation on Machine A, and the results are presented in Tables 3 and 4.

Table 3. Results of performance benchmarking for ML-DSA.

Algorithm Function	AVX2	Reference	Speed Improvement
Key Generation	136 µs	278 µs	>2×
Signature Generation	148 µs	636 µs	>4×
Signature Verification	85 µs	291 µs	>3×
Overall	370 µs	1206 µs	>3×

Table 4. Results of performance benchmarking for ML-KEM.

Algorithm Function	AVX2	Reference	Speed Improvement
Key Generation	79 µs	171 µs	>2×
Encapsulation	77 µs	169 µs	>2×
Decapsulation	25 µs	116 µs	>4×
Overall	181 µs	456 µs	>2×

The AVX2 optimized implementations demonstrated significant improvements in speed performance across all cryptographic operations. These improvements are due to the use of 256-bit SIMD registers, which allow multiple data elements to be processed in parallel. Computationally intensive operations such as the Number Theoretic Transform (NTT) and pointwise polynomial multiplications, among other operations, benefited from the AVX2 vectorization, resulting in faster execution compared to the standard 64-bit register operations used in the reference implementation.

As shown in Table 3, different functions of the ML-DSA algorithm had different speed improvements. Key generation achieved slightly over a twofold speed increase, while signature generation and verification showed larger improvements, with signature generation achieving a more than fourfold increase in execution speed. Overall, AVX2 achieved slightly over a threefold increase in execution speed when executing the ML-DSA algorithm across all major functions.

Similarly, the benchmarking results for ML-KEM, shown in Table 4, indicate performance improvements across all major functions. Key generation and encapsulation demonstrated slightly over twofold speed increase, while decapsulation achieved improvements exceeding fourfold. Overall, the AVX2 implementation achieved more than twice the execution speed of the reference implementation for the ML-KEM algorithm across all major functions.

The varying levels of speed improvement observed are primarily due to differences in the number of computationally intensive operations amongst the different major functions, which ultimately affects the extent to which each function benefits from vectorized operations. Functions that have a larger number of complex computations tend to achieve greater performance speed improvements from AVX2 optimizations. The performance benchmarking results demonstrate the effectiveness of AVX2 vectorization in improving the computational efficiency of PQC algorithms, enhancing their feasibility in practical applications while maintaining correctness and compatibility with the reference implementations.

It should be noted that the ML-KEM compiler-intrinsics implementation was developed independently by a different staff member within the same organization. Hence, the ML-KEM results are presented as corroborating evidence that similar outcomes are achieved across PQC algorithms, and not as an original contribution of this paper.

5.7. Comparison with Existing PQC Implementations

The AVX2 proposed implementation is compared with the existing PQC implementations in terms of optimized implementations and cross-platform support. The comparison is summarized in Table 5. To enable a meaningful comparison of execution times, we note that the detailed hardware specifications and compiler versions of Machine A were not recorded during the original experiments, and Machine A is no longer accessible. Consequently, the reference implementation and the Open Quantum Safe (liboqs) implementation were executed on Machine B. All implementations were compiled using the same optimization level (−O2), which is the standard production-recommended optimization setting supported by major compilers, including GCC, Clang, and MSVC. This ensured a consistent compilation baseline across all evaluated implementations.

Table 5. Comparison with existing pqc implementations.

ML-DSA Implementation	Optimized?	Cross-Platform?	Remarks
Reference Implementation	No	Yes	Provides a portable implementation without vectorized optimizations
Open Quantum Safe (liboqs)	Yes	Depends on algorithm	Broad PQC library incorporating optimized implementations for supported architectures; however, these optimizations are predominantly expressed as platform-specific assembly written for Linux
Proposed Implementation	Yes	Yes	Uses AVX2 compiler intrinsics to retain vectorized optimization while enabling the implementation to be compiled across the evaluated Windows and Linux environments

Since the reference implementation was benchmarked on both Machine A and Machine B, it was used as a common baseline to bridge the results obtained on the two platforms. Specifically, the execution times of the AVX2 liboqs implementation on Machine B were normalized relative to the reference implementation, allowing them to be compared with the proposed AVX2 implementation measured on Machine A. This normalization approach enabled a fair comparison between the AVX2 proposed implementation and the AVX2 liboqs implementation, as reported in Tables 6 and 7.

Table 6. Performance comparison of ML-DSA between the proposed AVX2 compiler-intrinsics and AVX2 liboqs implementations.

ML-DSA Algorithm Function	AVX2 Proposed Implementation	AVX2 Liboqs
Key Generation	136 µs	123 µs
Signature Generation	148 µs	223 µs
Signature Verification	85 µs	129 µs
Overall	370 µs	475 µs

Table 7. Performance comparison of ML-KEM between the proposed AVX2 compiler-intrinsics and AVX2 liboqs implementations.

ML-KEM Algorithm Function	AVX2 Proposed Implementation	AVX2 Liboqs
Key Generation	79 µs	58 µs
Encapsulation	77 µs	62 µs
Decapsulation	25 µs	42 µs
Overall	181 µs	162 µs

Although the specific hardware specifications of Machine A are unavailable and therefore cannot be reported, the use of a common reference baseline and a consistent compilation configuration reduces the impact of platform-dependent factors. Consequently, the relative performance differences presented in Tables 6 and 7 primarily reflect differences between the implementations rather than variations in the experimental environment.

The above comparison shows that liboqs already provides broad cross-platform support and architecture-specific optimization. However, these optimizations are predominantly expressed as platform-specific assembly written for Linux. While liboqs's .S assembly files can in principle be compiled on Windows, doing so is non-trivial, as the standard MSVC toolchain cannot build .S files directly, and it does not link cleanly for every algorithm outside of the Linux context due to missing or platform-specific source dependencies. Achieving the same optimized code path on two platforms with liboqs may therefore require maintaining and validating minimally two separate low-level implementations, i.e., one for Windows and another one for Linux.

The contribution in this paper lies in investigating the use of C compiler intrinsics as a single, portable representation of AVX2-optimized ML-DSA routines, reducing dependence on platform-specific assembly syntax while retaining explicit AVX2 vector operations, which are particularly significant for ML-DSA and ML-KEM given that the NTT and inverse-NTT operations central to polynomial multiplication are among the most computationally expensive steps and benefit most from vectorization, across the evaluated Windows and Linux environments.

Table 6 presents a quantitative comparison of algorithm performance between the proposed AVX2 compiler-intrinsics implementation and the AVX2-optimized liboqs implementation of ML-DSA. The results show a mixed but overall favorable performance profile: liboqs achieves modestly faster key generation, while the proposed implementation outperforms liboqs on signature generation and verification, resulting in a faster overall execution time. This indicates that translating AVX2 assembly routines into C compiler intrinsics preserves its performance speed relative to a mature, open-source assembly library, while offering the added benefit of a single portable C codebase rather than platform-specific assembly.

Table 7 presents a similar corresponding comparison for ML-KEM. The results show that the two implementations perform closely overall, with execution times differing by only

tens of microseconds across all functions and an overall minimal gap of 181 µs versus 162 µs. This close performance margin indicates that the compiler-intrinsics approach retains most of the vectorized performance benefit of ML-KEM's assembly variant, further supporting the viability of this approach as a practical alternative to platform-specific assembly.

6. Contributions and Technical Innovations

The solution introduces several technical innovations achieved beyond existing implementations as detailed in this section.

6.1. Bridging Research PQC Implementations to Practical Systems

This solution bridges the gap between research-level PQC implementations and practical system deployment by adapting the ML-DSA implementation for reliable execution in a controlled development environment and integrating the existing ML-KEM compiler-intrinsics implementation into a unified library. As research implementations tend to be experimental, additional steps are required to further develop and validate them before they can be used in organizational settings. Therefore, this work focused primarily on the translation, testing, and integration of the ML-DSA implementation, while the existing ML-KEM implementation was incorporated and tested as part of the unified library. This demonstrates the transition from research implementations towards practical deployment in real-world operating environments.

6.2. Integrated PQC Cryptographic Library

An integrated PQC cryptographic library was developed by combining ML-DSA and ML-KEM into a unified DLL. The library allows applications to access multiple PQC algorithms through a consistent, simplified, and well-defined interface by calling the specific major functions of ML-DSA and ML-KEM, without requiring direct interaction with low-level cryptographic implementations. As the DLL only exposes the intended public functions while restricting access to internal functions, it reduces the exposed interface surface and limits how the library can be invoked by external applications. Overall, it improves usability and provides a practical approach to utilizing PQC algorithms to protect operational systems and defend against quantum computing threats.

6.3. Cross-Platform PQC Implementation and Adaptation

Implementing cross-platform PQC algorithms will allow the algorithms to execute consistently across multiple environments, including both Linux and Windows platforms. Translations between code syntaxes were implemented and validated, where the algorithm could perform consistently in various operating environments utilizing different compiler configurations, assembly syntax conventions, and instruction-set architectures. By providing compatibility across platforms, it improves the portability and flexibility of the PQC implementations, allowing them to be more easily integrated into various system environments. This establishes a stronger technical foundation for future development and supports broader adoption of PQC algorithms in organization-wide practical applications.

7. Future Work

This article delved into the research and development of the ML-DSA and ML-KEM algorithm implementation, as well as the compilation of both PQC algorithms into an integrated DLL. While NIST has standardized multiple post-quantum cryptographic algorithms, this work focuses primarily on ML-KEM and ML-DSA because they constitute the principal NIST-recommended algorithms for key establishment and digital signatures, respectively. Therefore, implementing ML-KEM and ML-DSA addresses the two essential

cryptographic services required by almost all secure communication systems. Furthermore, current industry migration efforts predominantly adopt ML-KEM and ML-DSA. Concentrating on these two algorithms enables in-depth implementation, optimization, and benchmarking while maintaining a manageable project scope and maximizing practical impact for real-world quantum-safe system deployment.

Building on the outcomes of this research, future work may involve further research and development of additional PQC algorithms, such as Hamming Quasi-Cyclic (HQC) [25], which is another algorithm for key encapsulation. Such an extension would investigate the applicability of the current translation, integration, and validation approach to additional PQC algorithms, followed by appropriate functional, portability, and performance evaluation. Expanding the current research and implementation to incorporate additional PQC algorithms would allow for more flexibility in the adoption of PQC in different environments, due to the availability of a wider range of cryptographic standards. This would strengthen long-term cryptographic resilience and support the continued adaptation to the evolving PQC standardization landscape.

The present work does not independently evaluate timing leakage, memory-access leakage, alignment-dependent behavior, compiler-dependent transformations, or other implementation-level side-channel characteristics. Therefore, the proposed implementation is not claimed to have been independently validated for constant-time behavior or side-channel resistance. Hence, side-channel and constant-time analysis of the compiler-intrinsics implementation is another area for future investigation.

8. Conclusions

This article presented the development of an optimized, cross-platform PQC library addressing a critical gap in the practical deployment of quantum-resistant cryptography in organizational IT infrastructures. By translating AVX2 assembly routines into C compiler intrinsics, the work demonstrates that hardware-level performance and cross-platform portability are not mutually exclusive goals, a combination that existing open-source implementations have not achieved.

The compilation of both ML-DSA and ML-KEM algorithms into a unified DLL further lowers the barrier to wider organizational adoption, providing development teams with a consistent, standards-compliant cryptographic interface without requiring direct engagement with low-level implementation details. Compliance with NIST FIPS 203 and FIPS 204 specifications was verified through functional validation and testing against expected algorithm outputs, ensuring that performance gains were achieved without compromise to cryptographic correctness.

As quantum computing capabilities continue to advance with increasing PQC adoption, the importance of bridging the gap between research-level implementations and production-ready deployment will only increase. This work establishes a practical and extensible foundation for that transition, demonstrating that with the right implementation approach, organizations need not choose between performance, portability, and standards compliance in securing their infrastructures against quantum computing threats.

Author Contributions: Original draft preparation: C.Z.L.Y.; Supervision and paper revision: H.G.; Supervision: Z.Q.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author(s).

Acknowledgments: The authors deeply acknowledge the supervisor and the organization for their help and support for this project. The authors would also like to thank the Singapore Institute of Technology (SIT) for their continued support and guidance.

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

Abbreviations

The following abbreviations are used in this manuscript:

ARM	Advanced RISC Machine
ASM	Assembly Language
AT&T	American Telephone and Telegraph
AVX2	Advanced Vector Extensions 2
AVX-512	Advanced Vector Extensions 512
CRYSTALS	Cryptographic Suite for Algebraic Lattices
CPU	Central Processing Unit
DEVOPS	Development and Operations
DLL	Dynamic Link Library
DOD	Department of Defense
ECC	Elliptic Curve Cryptography
FIPS	Federal Information Processing Standards
HNDL	Harvest Now, Decrypt Later
HQC	Hamming Quasi-Cyclic
IoT	Internet of Things
ML-DSA	Module-Lattice-Based Digital Signature Algorithm
ML-KEM	Module-Lattice-Based Key-Encapsulation Mechanism
MSVC	Microsoft Visual C++
NIST	National Institute of Standards and Technology
NSA	National Security Agency
NTT	Number Theoretic Transform
OS	Operating System
PQC	Post-Quantum Cryptography
QKD	Quantum Key Distribution
RSA	Rivest-Shamir-Adleman
SHAKE	Secure Hash Algorithm and KECCAK
SIMD	Single Instruction Multiple Data
SM4	ShangMi 4

References

1. Shor, P.W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.* **1997**, *26*, 1484–1509. [CrossRef]
2. Bhatia, V.; Ramkumar, K. An Efficient Quantum Computing technique for cracking RSA using Shor's Algorithm. In *IEEE 5th International Conference on Computing Communication and Automation (ICCCA), Greater Noida, India, 2020*; IEEE: Piscataway, NJ, USA, 2020; pp. 89–94. [CrossRef]
3. Bene, F.; Kiss, A. Public Key Infrastructure in the Post-Quantum Era. In *2023 IEEE 17th International Symposium on Applied Computational Intelligence and Informatics (SACI), Timisoara, Romania, 2023*; IEEE: Piscataway, NJ, USA, 2023; pp. 000077–000082. [CrossRef]
4. Zhou, R.; Guo, H.; Teo, F.E.C. Investing Post-Quantum Cryptography to Secure Transmitted Data via Mobile Communication. *Electronics* **2026**, *15*, 1275. [CrossRef]
5. Chamola, V.; Jolfaei, A.; Chanana, V.; Parashari, P.; Hassija, V. Information security in the post quantum era for 5G and beyond networks: Threats to existing cryptography, and post-quantum cryptography. *Comput. Commun.* **2021**, *176*, 99–118. [CrossRef]
6. Kudelski Security Team. Quantum Attack Resource Estimate: Using Shor's Algorithm to Break RSA vs DH/DSA VS ECC. 24 August 2021. Available online: https://kudelskisecurity.com/research/quantum-attack-resource-estimate-using-shors-algorithm-to-break-rsa-vs-dh-dsa-vs-ecc (accessed on 12 December 2025).

7. Westerbaan, B.; Rubin, C.D. Defending Against Future Threats: Cloudflare Goes Post-Quantum. The Cloudflare Blog, 3 October 2022. Available online: https://blog.cloudflare.com/post-quantum-for-all/ (accessed on 12 October 2023).
8. Open-Quantum-Safe. GitHub—Open-Quantum-Safe/Liboqs: C Library for Prototyping and Experimenting with Quantum-Resistant Cryptography. GitHub. Available online: https://github.com/open-quantum-safe/liboqs (accessed on 10 October 2025).
9. National Institute of Standards and Technology (U.S.). Module-Lattice-Based Digital Signature Standard. Aug. 2024. Available online: https://doi.org/10.6028/nist.fips.204 (accessed on 13 October 2023).
10. NIST. Module-Lattice-Based Key-Encapsulation Mechanism Standard. FIPS 203 Federal Information Processing Standards Publication. Available online: https://doi.org/10.6028/NIST.FIPS.203 (accessed on 13 August 2024).
11. Jenefa, A.; Josh, F.T.; Taurshia, A.; Kumar, K.R.; Kowsega, S.; Naveen, E. PQC Secure: Strategies for Defending Against Quantum Threats. In *2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS)*; IEEE: Piscataway, NJ, USA, 2023; pp. 1799–1804. [CrossRef]
12. Mascelli, J.; Rodden, M. Harvest Now Decrypt Later': Examining Post-Quantum Cryptography and the Data Privacy Risks for Distributed Ledger Networks. *Financ. Econ. Discuss. Ser.* **2025**, *093*. [CrossRef]
13. Mosca, M. Cybersecurity in an Era with Quantum Computers: Will We Be Ready? *IEEE Secur. Priv.* **2018**, *16*, 38–41. [CrossRef]
14. Azhari, R.; Salsabila, A.N. Analyzing the Impact of Quantum Computing on Current Encryption Techniques. *IAIC Trans. Sustain. Digit. Innov.* **2024**, *5*, 148–157. [CrossRef]
15. Bennett, C.H.; Brassard, G. Quantum cryptography: Public key distribution and coin tossing. *Theor. Comput. Sci.* **2014**, *560*, 7–11. [CrossRef]
16. NSA. National Security Agency/Central Security Service > Cybersecurity > Quantum Key Distribution (QKD) and Quantum Cryptography QC. Available online: https://www.nsa.gov/Cybersecurity/Quantum-Key-Distribution-QKD-and-Quantum-Cryptography-QC/ (accessed on 8 February 2026).
17. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, U.S. Department of Commerce, "Post-Quantum Cryptography | CSRC | CSRC.". Available online: https://csrc.nist.gov/projects/post-quantum-cryptography (accessed on 11 February 2026).
18. NIST. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. July 2022. Available online: https://csrc.nist.gov/pubs/ir/8413/upd1/final (accessed on 15 March 2026).
19. Rawal, B.S.; Curry, P.J. Challenges and opportunities on the horizon of post-quantum cryptography. *APL Quantum* **2024**, *1*, 026110. [CrossRef]
20. Gitonga, C.K. The Impact of Quantum Computing on Cryptographic Systems: Urgency of Quantum-Resistant Algorithms and Practical Applications in Cryptography. *Eur. J. Inf. Technol. Comput. Sci.* **2025**, *5*, 1–10. [CrossRef]
21. Wang, B.; Gao, H.; Yang, F. Fast Implementation of Multiplication on Polynomial Rings. *Secur. Commun. Netw.* **2022**, *2022*, 1–9. [CrossRef]
22. Nejatollahi, H.; Dutt, N.; Ray, S.; Regazzoni, F.; Banerjee, I.; Cammarota, R. Post-Quantum Lattice-Based Cryptography Implementations. *ACM Comput. Surv.* **2019**, *51*, 1–41. [CrossRef]
23. Li, X.; Xie, M.; Huang, C. Fast Software Implementation of the SM4 Algorithm. In *Proceedings of the 2024 8th International Conference on High Performance Compilation, Computing and Communications*; ACM: New York, NY, USA, 2024; pp. 127–132. [CrossRef]
24. Benson, L.; Ebeling, R.; Rabl, T.; Hasso Plattner Institute; University of Potsdam. Evaluating SIMD Compiler-Intrinsics for database Systems. In *Workshops*; CEUR-WS: Aachen, Germany, 2023; pp. 1–14. Available online: https://ceur-ws.org/Vol-3462/ADMS5.pdf (accessed on 9 January 2026).
25. NIST. NIST Selects HQC as Fifth Algorithm for Post-Quantum Encryption. Available online: https://www.nist.gov/news-events/news/2025/03/nist-selects-hqc-fifth-algorithm-post-quantum-encryption (accessed on 11 March 2025).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.