# Can FrodoKEM Run in a Millisecond? FPGA Says Yes!

Gökçe Düzyol
National Research Institute of Electronics and Cryptology (UEKAE), TÜBİTAK BİLGEM
Kocaeli, Turkiye
gokce.duzyol@tubitak.gov.tr

Muhammed Said Gündoğan
National Research Institute of Electronics and Cryptology (UEKAE), TÜBİTAK BİLGEM
Kocaeli, Turkiye
said.gundogan@tubitak.gov.tr

Atakan Arslan
National Research Institute of Electronics and Cryptology (UEKAE), TÜBİTAK BİLGEM
Kocaeli, Turkiye
atakan.arslan@tubitak.gov.tr

## Abstract

FrodoKEM is a post-quantum key encapsulation mechanism based on plain Learning With Errors (LWE). In contrast to module-lattice-based schemes, it relies on an unstructured variant of the LWE problem, providing more conservative and better-understood security guarantees.

In this paper, we explore hardware-level parallelization of FrodoKEM's matrix multiplication, which dominates its execution time. Earlier high-throughput work by Howe et al. [9] identified SHAKE as a performance bottleneck and replaced it with the Trivium stream cipher, which renders the implementation incompatible with the FrodoKEM specification. In contrast, our work adheres strictly to the specification, including its use of SHAKE as the PRNG, and introduces a parameterized architecture whose degree of parallelism is set by design-time parameters.

We present operation-specific datapaths for FrodoKEM-640 on Artix-7 with 24 parallel multipliers for key generation and 32 for encapsulation and decapsulation, achieving 976–1077 operations per second, i.e., latencies of 0.93–1.02 ms. Among the earlier specification-compliant designs, this is the lowest latency and the lowest area–time product. The design shows that SHAKE is not a barrier to parallel matrix multiplication and that efficient, standard-compliant FrodoKEM implementations are practical for high-speed cryptographic applications.

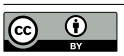
## CCS Concepts

• **Security and privacy → Public key (asymmetric) techniques**; **Hardware security implementation**; • **Hardware →** *Hardware accelerators.*

## Keywords

FrodoKEM, Key Encapsulation Mechanism, Hardware Acceleration, FPGA, Post-quantum Cryptography

**ACM Reference Format:**
Gökçe Düzyol, Muhammed Said Gündoğan, and Atakan Arslan. 2026. Can FrodoKEM Run in a Millisecond? FPGA Says Yes!. In *International Workshop on Hardware and Architectural Support for Security and Privacy 2026 (HASP '26), October 31–November 01, 2026, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3857307.3857317



This work is licensed under a Creative Commons Attribution 4.0 International License.
*HASP '26, Athens, Greece*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2964-5/2026/10
https://doi.org/10.1145/3857307.3857317

## 1 Introduction

The development of quantum computing threatens modern cryptography: public-key systems such as RSA and ECC, based on integer factorization and discrete logarithms, can be broken in polynomial time by Shor's algorithm [19], endangering critical infrastructures. This necessitates the development of quantum-resistant cryptographic alternatives. To address this, NIST launched the Post-Quantum Cryptography (PQC) project in 2016 and selected CRYSTALS-KYBER (standardized as ML-KEM) as the primary key encapsulation mechanism [16]. As an alternative, FrodoKEM remains of interest due to its reliance on standard (unstructured) Learning With Errors (LWE), providing conservative security without algebraic assumptions. This makes FrodoKEM an attractive choice for applications that prioritize long-term robustness against unforeseen cryptanalytic advances. This has led to support from several European governments and to its standardization in ISO/IEC 18033-2 [11] and submission to the IRTF CFRG as Internet-Draft (2025) [14]. However, this conservative design comes at the cost of efficiency: FrodoKEM sacrifices efficiency in terms of key and ciphertext sizes and computational cost compared to schemes like Kyber. While prior work has explored software optimizations and side-channel-resistant implementations, efficient hardware acceleration of FrodoKEM remains challenging due to its computationally intensive matrix operations and the need to retain the specification-compliant PRNG, which can limit opportunities for parallelism and increase the associated hardware cost. In this work, we present a matrix multiplication architecture that parallelizes computation to accelerate FrodoKEM on FPGAs, aiming to meet high throughput demands.

## Contributions

The most time-consuming step in FrodoKEM is matrix multiplication. Earlier high-throughput work parallelized it only after replacing SHAKE with Trivium [9], suggesting that SHAKE's sequential generation of $A$ precludes high parallelism. We show that this is not the case. We identify the constraints that limit parallel matrix multiplication under SHAKE, namely the row-by-row generation of $A$, the two-access limit of dual-port BRAMs, and the sequential sampling of $S$ and $S'$, and design around them without changing the PRNG. Because $A \cdot S$ and $S' \cdot A$ consume $A$ in different orders, we specialize each datapath: a $k \times l \times 8$ architecture for key generation and an $8 \times n \times m$ architecture for encapsulation and decapsulation, whose design-time parameters scale the number of multipliers against BRAM and SHAKE budgets (Section 5).

With 24 multipliers for key generation and 32 for encapsulation and decapsulation, our Artix-7 implementation reaches latencies

of 0.93–1.02 ms (976–1077 ops/sec) using at most 17,066 LUTs and 20 BRAMs per operation. This is the lowest latency and area–time product among the earlier specification-compliant designs [3, 10, 15, 20], whose latencies range from 2 to 32 ms. We also compare our results with designs that appeared after the preprint of this work [5] (Section 6.1). The methodology is not specific to FrodoKEM-640 and is expected to extend to FrodoKEM-976 and FrodoKEM-1344; evaluating these parameter sets is left to future work.

## 2 Related Work

In 2018, Howe et al. presented FPGA and software implementations of FrodoKEM-640 and FrodoKEM-976, the latter on an ARM Cortex-M4F [10]. Their FPGA design balances area and throughput with a single LWE multiplication core and parallel execution, achieving full key exchange in 60 ms (128-bit security) and 135 ms (192-bit security).

In 2021, Buschkowski [3] compared High-Level Synthesis (HLS) with direct hardware implementations, building on the design principles of [10]. The results indicated that the optimized HLS method delivered performance comparable to that of the direct implementation, particularly in terms of latency and resource utilization.

In 2021, Howe et al. [9] identified SHAKE as a performance bottleneck in FrodoKEM hardware due to its sequential structure, and proposed replacing it with the Trivium stream cipher for higher throughput and lower area. Their design parallelizes matrix multiplication, achieves 14× faster encapsulation and 13× faster decapsulation. However, this deviates from the FrodoKEM specification, which—recommended by the German [6], French [7], and Dutch [17] governments, as well as ISO and IETF—requires AES or SHAKE for randomness generation.

In 2022, Karl et al. [12] proposed a RISC-V–based HW/SW co-design for accelerating FrodoKEM across all three parameter sets. Matrix multiplication, discrete Gaussian sampling, and the Keccak round function are implemented in hardware, while the remaining operations are handled in software. Their design achieves a speedup over plain software implementations on RISC-V; however, it remains slower than a fully hardware implementation.

In 2023, Bos et al. [2] optimized FrodoKEM for resource-constrained devices by reducing the algorithm's memory footprint, notably generating matrix $A$ on-the-fly, which saves memory but requires recomputation. Our design also generates $A$ on-the-fly, but only once, thereby avoiding redundant computations.

In 2025, Urbano-Molano et al. [20] presented an HLS-based implementation derived from the reference C code, applying profiling-guided optimizations to improve FPGA performance.

In 2025, Manzoni et al. [15] presented a unified, DSP-free design with 32 LUT-based $16 \times 5$-bit multipliers that targets low resource usage on small Artix-7 devices. The results of the relevant implementations are summarized in Tables 1–3

Several FrodoKEM accelerators have appeared since the preprint of this work [5]. Deshpande et al. [4] propose a unified, specification-compliant architecture that executes key generation, encapsulation, and decapsulation for all twelve parameter sets (three security levels, SHAKE- and AES-based, standard and ephemeral) at run time, with a synthesis-time parameter $T$ that sets the number of parallel multipliers; their reported area excludes the SHAKE module. Li et al. [13] present an instruction-driven crypto-processor that overlaps the generation of $A$ with matrix multiplication, replaces DSP multipliers with sign–magnitude LUT-based multipliers, and reduces BRAM usage to 14 blocks through compact memory scheduling. Zhu et al. [21] reformulate the multiplication as $4 \times 4$ block operations fed by four parallel SHAKE instances and use DSP-free carry-save Wallace trees, reaching the lowest latency reported to date at the cost of substantially higher logic usage. All three are unified cores supporting all FrodoKEM parameter sets, whereas our design specializes each operation separately. Their results are included in Tables 1–3.

## 3 Background on FrodoKEM

FrodoKEM is a post-quantum key encapsulation mechanism (KEM) based on the Learning With Errors (LWE) problem over unstructured lattices. Unlike Ring/Module-LWE schemes, it avoids algebraic structure, providing a conservative design with strong theoretical robustness against quantum adversaries. It supports three parameter sets: FrodoKEM-640, -976, and -1344, targeting IND-CCA security at NIST Levels 1, 3, and 5, comparable to AES-128, AES-192, and AES-256, respectively [14]. FrodoKEM has two variants: the standard version, which includes a salt and allows key reuse, and eFrodoKEM, which omits the salt and is intended for limited ciphertexts per key. As this distinction has negligible performance impact, the two are largely interchangeable in practical implementations.

A central component of FrodoKEM is the public matrix $A$. Due to the lack of algebraic structure, storing or transmitting $A$ directly would be costly. Instead, a seed is embedded in the key pair and expanded at runtime via `Frodo.Gen`, instantiated with AES-128 or SHAKE-128. AES is common in software, while SHAKE is generally preferred in hardware for efficiency.

Algorithms 1–3 present key generation, encapsulation, and decapsulation for eFrodoKEM-640-SHAKE. This implementation largely follows eFrodoKEM and, with minor modifications, can be adapted to the standard variant, in line with the official specification [1, 8]. The SHAKE-128 extendable-output function used to generate the public matrix $A$ is detailed in Algorithm 4.

For FrodoKEM-640, $n = 640$, $\bar{n} = \bar{m} = 8$, $q = 2^{15}$, $B = 2$, and $\text{len}_\chi = 16$; the inherent 8-fold parallelism exploited in Section 5 follows from $\bar{n} = 8$. Note that the architectural parameters $n$ and $m$ introduced in Section 5.2 are design-time scaling factors and are unrelated to the FrodoKEM dimensions.

## 4 Challenges in Accelerating Matrix Multiplication in FrodoKEM

In hardware implementations of FrodoKEM, matrix multiplication is by far the most computationally demanding operation: in the single-multiplier implementation of Buschkowski [3], it accounts for 99.58%, 98.49%, and 97.38% of the total execution cycles in key generation, encapsulation, and decapsulation, respectively. Therefore, in the proposed implementation we focus on optimizing the matrix multiplication.

Matrix multiplication is realized as sequential matrix–vector ($A \cdot S$) or vector–matrix ($S' \cdot A$) operations. A natural approach to improve parallelism is to increase the number of hardware multipliers; however, several architectural constraints must first be

**Algorithm 1** FrodoKEM.KeyGen

**Function** FrodoKEM.KeyGen
$s \| \text{seed}_{SE} \| z \leftarrow \mathcal{U}(\{0,1\}^{\text{len}_s + \text{len}_{SE} + \text{len}_z})$
$\text{seed}_A \leftarrow \text{SHAKE}(z, \text{len}_{\text{seed}_A})$
$A \in \mathbb{Z}_q^{n \times n}$ via $A \leftarrow \text{Frodo.Gen}(\text{seed}_A)$
$(r^{(0)}, \ldots, r^{(2\bar{n}n-1)}) \leftarrow$
  $\text{SHAKE}(\text{0x5F} \| \text{seed}_{SE},\ 2\bar{n}n \cdot \text{len}_\chi)$
$S^T \leftarrow \text{Frodo.SampleMatrix}($
  $(r^{(0)}, \ldots, r^{(\bar{n}n-1)}),\ \bar{n}, n, T_\chi)$
$E \leftarrow \text{Frodo.SampleMatrix}($
  $(r^{(n\bar{n})}, \ldots, r^{(2n\bar{n}-1)}),\ n, \bar{n}, T_\chi)$
$B \leftarrow AS + E$
$b \leftarrow \text{Frodo.Pack}(B)$
$pkh \leftarrow \text{SHAKE}(\text{seed}_A \| b, \text{len}_{pkh})$
**return** $pk \leftarrow \text{seed}_A \| b,$
  $sk' \leftarrow (s \| \text{seed}_A \| b, S^T, pkh)$
**end function**

**Algorithm 2** FrodoKEM.Encaps

**Function** FrodoKEM.Encaps$pk$
Parse $pk = (\text{seed}_A \| b)$
$\mu \leftarrow \mathcal{U}(\{0,1\}^{\text{len}_\mu})$
$pkh \leftarrow \text{SHAKE}(pk, \text{len}_{pk})$
$(\text{seed}_{SE} \| k) \leftarrow$
  $\text{SHAKE}(pkh \| \mu,\ \text{len}_{seed_{SE}} + \text{len}_k)$
$(r^{(0)}, \ldots, r^{(2\bar{m}n + \overline{mn}-1)}) \leftarrow$
  $\text{SHAKE}(\text{0x96} \| \text{seed}_{SE},\ (2\bar{m}n + \overline{mn}) \cdot \text{len}_\chi)$
$S' \leftarrow \text{Frodo.SampleMatrix}($
  $(r^{(0)}, \ldots, r^{(\bar{m}n-1)}),\ \bar{m}, n, T_\chi)$
$E' \leftarrow \text{Frodo.SampleMatrix}($
  $(r^{(\bar{m}n)}, \ldots, r^{(2\bar{m}n-1)}),\ \bar{m}, n, T_\chi)$
$A \leftarrow \text{Frodo.Gen}(\text{seed}_A)$
$B' \leftarrow S'A + E'$
$c_1 \leftarrow \text{Frodo.Pack}(B')$
$E'' \leftarrow \text{Frodo.SampleMatrix}($
  $(r^{(2\bar{m}n)}, \ldots, r^{(2\bar{m}n + \overline{mn}-1)}),\ \bar{m}, \bar{n}, T_\chi)$
$B \leftarrow \text{Frodo.Unpack}(b, n, \bar{n})$
$V \leftarrow S'B + E''$
$C \leftarrow V + \text{Frodo.Encode}(\mu)$
$c_2 \leftarrow \text{Frodo.Pack}(C)$
$\text{ss} \leftarrow \text{SHAKE}(c_1 \| c_2 \| k, \text{len}_{ss})$
**return** $(c_1 \| c_2, \text{ss})$
**end function**

addressed. BRAMs are true dual-port blocks with at most two accesses per cycle, so concurrent requests from multiple multipliers can create bottlenecks. Moreover, SHAKE generates $S$ column-wise and $S'$, $E$, and $E'$ row-wise in a strictly sequential manner, requiring these matrices to be precomputed and stored. Matrix $A$, which requires about 819,200 B, cannot be fully stored on the FPGA and is instead generated row by row using a double-buffering scheme [10], where one buffer feeds the multiplier while the other prepares the next row. In Section 5, we describe how our architecture overcomes these challenges.

**Algorithm 3** FrodoKEM.Decaps

**Function** FrodoKEM.Decaps$c_1 \| c_2, sk'$
Parse $sk' = (s \| \text{seed}_A \| b, S^T, pkh)$
$B' \leftarrow \text{Frodo.Unpack}(c_1, \bar{m}, n)$
$C \leftarrow \text{Frodo.Unpack}(c_2, \bar{m}, \bar{n})$
$M \leftarrow C - B'S$
$\mu' \leftarrow \text{Frodo.Decode}(M)$
Parse $pk = (\text{seed}_A \| b)$
$(\text{seed}'_{SE} \| k') \leftarrow \text{SHAKE}(pkh \| \mu', \text{len}_{seed_{SE}} + \text{len}_k)$
$(r^{(0)}, \ldots, r^{(2\bar{m}n + \overline{mn}-1)}) \leftarrow$
  $\text{SHAKE}(\text{0x96} \| \text{seed}'_{SE},\ (2\bar{m}n + \overline{mn}) \cdot \text{len}_\chi)$
$S' \leftarrow \text{Frodo.SampleMatrix}($
  $(r^{(0)}, \ldots, r^{(\bar{m}n-1)}),\ \bar{m}, n, T_\chi)$
$E' \leftarrow \text{Frodo.SampleMatrix}($
  $(r^{(\bar{m}n)}, \ldots, r^{(2\bar{m}n-1)}),\ \bar{m}, n, T_\chi)$
$A \leftarrow \text{Frodo.Gen}(\text{seed}_A)$
$B'' \leftarrow S'A + E'$
$E'' \leftarrow \text{Frodo.SampleMatrix}($
  $(r^{(2\bar{m}n)}, \ldots, r^{(2\bar{m}n + \overline{mn}-1)}),\ \bar{m}, \bar{n}, T_\chi)$
$B \leftarrow \text{Frodo.Unpack}(b, n, \bar{n})$
$V \leftarrow S'B + E''$
$C' \leftarrow V + \text{Frodo.Encode}(\mu')$
**if** $B' \| C = B'' \| C'$ **then**

  **return** $ss \leftarrow \text{SHAKE}(c_1 \| c_2 \| k', \text{len}_{ss})$
**else**

  **return** $ss \leftarrow \text{SHAKE}(c_1 \| c_2 \| s, \text{len}_{ss})$
**end if**
**end function**

**Algorithm 4** Frodo.Gen using SHAKE-128

**Require:** Seed $\text{seedA} \in \{0,1\}^{\text{len}_{\text{seedA}}}$
**Ensure:** Pseudorandom matrix $A \in \mathbb{Z}_q^{n \times n}$
**for** $i = 0$ **to** $n - 1$ **do**
  $b \leftarrow \langle i \rangle \| \text{seedA} \in \{0,1\}^{16 + \text{len}_{\text{seedA}}}$
  $\langle c_{i,0} \rangle \| \langle c_{i,1} \rangle \| \cdots \| \langle c_{i,n-1} \rangle \leftarrow \texttt{SHAKE128}(b, 16n)$ where each $\langle c_{i,j} \rangle \in \{0,1\}^{16}$
  **for** $j = 0$ **to** $n - 1$ **do**
    $A_{i,j} \leftarrow c_{i,j} \bmod q$
  **end for**
**end for**

**return** $A$

## 5 An Architecture Realization for FrodoKEM-640-SHAKE

In this section, we propose an architecture that parallelizes the matrix multiplication in the FrodoKEM-640-SHAKE algorithm and achieves the smallest latency among the specification-compliant designs that predate this work. Our design implements the complete FrodoKEM-640-SHAKE flow; however, since the optimization focus is on the matrix-multiplication part, only this portion is shown in Figures 1 and 2. Throughout this section, an $N\times$ speed-up denotes

**Table 1: FrodoKEM-640 Key Generation.**

| | SC | LUT | FF | Slices | DSP | BRAM | Freq. (MHz) | Ops/sec | Latency (ms) | Area×Time (ENS×Secs) |
|---|---|---|---|---|---|---|---|---|---|---|
| **This Work** | ✓ | 12,246 | 8,859 | 3,793 | 24 | 18 | 147 | 976 | 1.024 | 9.95 |
| [20] | ✓ | 35,398 | 20,648 | – | 1 | 10.5 | 135 | 67 | 14.88 | 163.79 |
| [20]* | ✓ | 28,838 | 20,731 | – | 4 | 12 | 167 | 83 | 12.03 | – |
| [3] | ✓ | 12,687 | 6,719 | – | 1 | 6 | 167 | 51 | 19.74 | 87.79 |
| [10] | ✓ | 3,771 | 1,800 | 1,035 | 1 | 6 | 167 | 51 | 19.65 | 45.41 |
| 1x: [9] | × | 971 | 433 | 290 | 1 | 0 | 191 | 59 | 16.95 | 6.61 |
| 4x | × | 1,174 | 781 | 355 | 4 | 0 | 185 | 226 | 4.43 | 3.34 |
| 8x | × | 1,679 | 1,570 | 532 | 8 | 0 | 182 | 445 | 2.25 | 2.99 |
| 16x | × | 2,587 | 2,994 | 855 | 16 | 0 | 172 | 840 | 1.19 | 2.92 |
| [15]† | ✓ | 19,082 | 5,331 | – | 0 | 8 | 62.5 | 469 | 2.132 | 13.52 |
| *Designs published after our preprint [5]* | | | | | | | | | | |
| [4]‡ T=16 | ✓ | 19,044 | 6,232 | – | 16 | 43 | 155 | 740 | 1.35 | 19.99 |
| T=32 | ✓ | 24,721 | 7,511 | – | 32 | 48 | 145 | 1382 | 0.72 | 13.59 |
| T=64 | ✓ | 37,632 | 10,180 | – | 64 | 89 | 138 | 2599 | 0.38 | 12.79 |
| [13]† | ✓ | 13,423 | 6,042 | – | 0 | 14 | 207 | 1164 | 0.859 | 5.24 |
| [21]† | ✓ | 56,418 | 25,124 | – | 0 | 32 | 138 | 2959 | 0.338 | 6.89 |

SC: Standard Compliance. *: Stratix IV platform. †: single unified core supporting all three operations and all parameter sets. ‡: unified core; area excludes the SHAKE module, so the Area×Time value is a lower bound. T: number of parallel multipliers. Slices for [3, 4, 13, 15, 20, 21] estimated as LUT/4.

**Table 2: FrodoKEM-640 Key Encapsulation.**

| | SC | LUT | FF | Slices | DSP | BRAM | Freq. (MHz) | Ops/sec | Latency (ms) | Area×Time (ENS×Secs) |
|---|---|---|---|---|---|---|---|---|---|---|
| **This Work** | ✓ | 15,908 | 12,111 | 5,075 | 32 | 20 | 137 | 1077 | 0.928 | 11.31 |
| [20] | ✓ | 52,068 | 31,095 | – | 5 | 13 | 124 | 33 | 30.15 | 484.35 |
| [20]* | ✓ | 36,706 | 30,820 | – | 8 | 18 | 164 | 43 | 23.22 | – |
| [3] | ✓ | 11,396 | 7,024 | – | 1 | 10 | 167 | 50 | 19.96 | 97.98 |
| [10] | ✓ | 6,745 | 3,528 | 1,855 | 1 | 11 | 167 | 51 | 19.89 | 81.76 |
| 1x: [9] | × | 4,246 | 2,131 | 1,180 | 1 | 0 | 190 | 58 | 17.24 | 22.06 |
| 4x | × | 4,620 | 2,552 | 1,338 | 4 | 0 | 183 | 221 | 4.52 | 7.85 |
| 8x | × | 5,155 | 3,356 | 1,485 | 8 | 0 | 177 | 427 | 2.34 | 5.34 |
| 16x | × | 5,796 | 4,694 | 1,692 | 16 | 0 | 171 | 825 | 1.21 | 3.98 |
| [15]† | ✓ | 19,082 | 5,331 | – | 0 | 8 | 62.5 | 458 | 2.183 | 13.84 |
| *Designs published after our preprint [5]* | | | | | | | | | | |
| [4]‡ T=16 | ✓ | 19,044 | 6,232 | – | 16 | 43 | 155 | 716 | 1.40 | 20.66 |
| T=32 | ✓ | 24,721 | 7,511 | – | 32 | 48 | 145 | 1306 | 0.77 | 14.39 |
| T=64 | ✓ | 37,632 | 10,180 | – | 64 | 89 | 138 | 2338 | 0.43 | 14.22 |
| [13]† | ✓ | 13,423 | 6,042 | – | 0 | 14 | 207 | 1142 | 0.876 | 5.34 |
| [21]† | ✓ | 56,418 | 25,124 | – | 0 | 32 | 138 | 2755 | 0.363 | 7.40 |

SC: Standard Compliance. *: Stratix IV platform. †: single unified core supporting all three operations and all parameter sets. ‡: unified core; area excludes the SHAKE module, so the Area×Time value is a lower bound. T: number of parallel multipliers. Slices for [3, 4, 13, 15, 20, 21] estimated as LUT/4.

$N$ parallel multipliers relative to a single-multiplier datapath, which is the ideal gain for the matrix-multiplication phase.

We first introduce a general methodology based on the parameters $k$, $l$, $n$, and $m$ to obtain higher parallelism while fully adhering to

**Table 3: FrodoKEM-640 Key Decapsulation.**

| | SC | LUT | FF | Slices | DSP | BRAM | Freq. (MHz) | Ops/sec | Latency (ms) | Area×Time (ENS×Secs) |
|---|---|---|---|---|---|---|---|---|---|---|
| **This Work** | ✓ | 17,066 | 12,868 | 5,503 | 32 | 20 | 133 | 1052 | 0.95 | 11.99 |
| [20] | ✓ | 43,616 | 23,081 | – | 5 | 18.5 | 122 | 31 | 31.53 | 473.90 |
| [20]* | ✓ | 29,216 | 22,946 | – | 12 | 22 | 161 | 41 | 23.89 | – |
| [3] | ✓ | 11,792 | 7,451 | – | 1 | 10 | 162 | 48 | 20.76 | 103.96 |
| [10] | ✓ | 7,220 | 3,549 | 1,992 | 1 | 16 | 162 | 49 | 20.72 | 108.32 |
| 1x [9] | × | 10,518 | 2,299 | 2,933 | 1 | 0 | 190 | 57 | 17.54 | 53.19 |
| 4x | × | 11,581 | 2,818 | 3,424 | 4 | 0 | 174 | 208 | 4.81 | 18.39 |
| 8x | × | 13,128 | 3,737 | 3,710 | 8 | 0 | 164 | 391 | 2.55 | 11.50 |
| 16x | × | 14,528 | 5,335 | 4,020 | 16 | 0 | 160 | 763 | 1.31 | 7.36 |
| 1x § | × | 4,466 | 2,152 | 1,254 | 1 | 12.5 | 162 | 49 | 20.41 | 77.63 |
| 4x § | × | 4,841 | 2,661 | 1,345 | 4 | 12.5 | 161 | 192 | 5.21 | 21.85 |
| 8x § | × | 5,476 | 3,479 | 1,558 | 8 | 12.5 | 156 | 372 | 2.69 | 12.93 |
| 16x § | × | 6,881 | 5,081 | 1,947 | 16 | 12.5 | 149 | 710 | 1.41 | 8.45 |
| [15]† | ✓ | 19,082 | 5,331 | – | 0 | 8 | 62.5 | 449 | 2.227 | 14.12 |
| *Designs published after our preprint [5]* | | | | | | | | | | |
| [4]‡ T=16 | ✓ | 19,044 | 6,232 | – | 16 | 43 | 155 | 698 | 1.43 | 21.19 |
| T=32 | ✓ | 24,721 | 7,511 | – | 32 | 48 | 145 | 1246 | 0.80 | 15.08 |
| T=64 | ✓ | 37,632 | 10,180 | – | 64 | 89 | 138 | 2147 | 0.47 | 15.49 |
| [13]† | ✓ | 13,423 | 6,042 | – | 0 | 14 | 207 | 1133 | 0.883 | 5.39 |
| [21]† | ✓ | 56,418 | 25,124 | – | 0 | 32 | 138 | 2725 | 0.367 | 7.48 |

SC: Standard Compliance. *: Stratix IV platform. †: single unified core supporting all three operations and all parameter sets. ‡: unified core; area excludes the SHAKE module, so the Area×Time value is a lower bound. T: number of parallel multipliers. Slices for [3, 4, 13, 15, 20, 21] estimated as LUT/4. §: BRAM-based versions reported in [9].

the FrodoKEM specification. These parameters characterize design-time architectural scaling and enable systematic exploration of different area–time trade-offs. For each operation, we select and present a concrete parameter set that meets our target execution time, and this configuration is implemented in our design. Although the methodology is not inherently device-specific, the implementation presented in this work targets a *Xilinx Artix-7* FPGA.

## 5.1 Key Generation Implementation Details

Key generation is dominated by the $A \cdot S$ multiplication, making it the primary target for architectural acceleration. In FrodoKEM-640, the eight columns of $S$ naturally allow eight parallel matrix–vector multiplications, since each generated row of $A$ can be reused across all eight columns of $S$, with $S$ precomputed. Beyond this inherent 8-fold parallelism, additional acceleration is constrained because each row of $A$ must be generated sequentially by SHAKE.

We introduce a general architectural method that further parallelizes each matrix–vector multiplication in $A \cdot S$ by a factor of $k \times l$, in addition to the inherent 8-fold structure. The method yields a fixed parallelism factor of $8kl$ for a chosen parameter set, resulting in a corresponding number of DSP-based multipliers and a proportional speed-up.

In particular, with $k = 3$ and $l = 1$, our implementation achieves a 24× speed-up, meeting the performance target and enabling key generation in 1.024 ms. Matrix $A$ is generated on-the-fly with double buffering; a single 0.5-BRAM unit with dual-port access suffices for $k = 2$. For higher parallelism ($k = 3, 4$), the active row of $A$ and the 8 columns of $S$ are partitioned across $\lceil \frac{k}{2} \rceil$ storage units each, enabling full parallel feeding of $8k$ multipliers, at the cost of

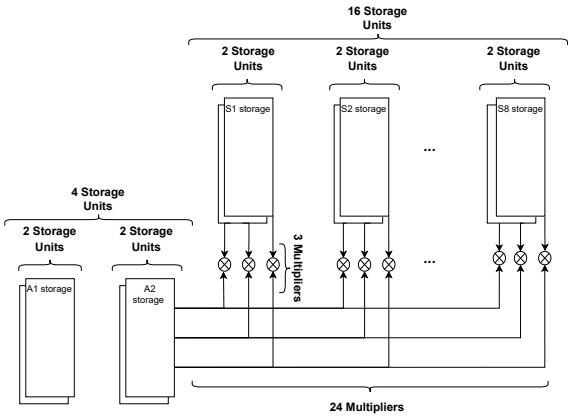



**Figure 1: Parallel $k \times l \times 8$ Architecture Instantiated with $k = 3$ and $l = 1$ for $A \cdot S$ Multiplication**

increased BRAM utilization. As $k$ increases, SHAKE must supply data fast enough for $8k$ multipliers. In FrodoKEM-640, one row has 640 elements, so row multiplication finishes in approximately $\frac{640}{k}$ cycles; if SHAKE cannot generate the next row within this time, it becomes the bottleneck. An optimized SHAKE helps, but for large $k$ a scalable solution is to use $l$ parallel SHAKE generators with separate buffers, producing $l$ rows simultaneously and enabling $8kl$ multiplications per cycle at the cost of $l$-fold higher resources.

In our implementation, we employed an efficient SHAKE design and observed that, for $k = 3$, SHAKE did not constitute a bottleneck. Therefore, no additional SHAKE units were required ($l = 1$). Treating $S$ as eight independent vectors provides an inherent 8-fold parallelism, since each generated row of $A$ is reused across all eight columns of $S$. To support this, $S$ and $E$ are precomputed and stored for constant access. Further gains come from processing three elements of $A$ and $S$ in parallel ($k = 3$). Even/odd BRAM partitioning is used to overcome dual-port limitations, and SHAKE outputs are buffered to minimize latency. With $l = 1$, one element per column is produced in each cycle, resulting in eight parallel outputs that are accumulated through four dual-port `B_storage` units.

## 5.2 Encapsulation & Decapsulation Implementation Details

Encapsulation and decapsulation share the $S' \cdot A$ multiplication as their main bottleneck, so they are optimized jointly.

Unlike key generation, where $A$ multiplies $S$ from the left, these operations require right-side multiplication. Since $A$ is generated on-the-fly row by row, each newly generated row of $A$ must be multiplied with all rows of $S'$ and accumulated into the corresponding partial sums of $B'$. This procedure corresponds to lines 10 and 13 of Algorithms 2 and 3 for encapsulation and decapsulation, respectively. Dropping primes for clarity, each update takes the form

$$b_{i,j} = b_{i,j} + s_{i,k} \cdot a_{k,j} \tag{1}$$

Effectively, Equation (1) is parallelized across all $b_{i,j}$ elements, and each time a row of matrix $A$ is generated, all corresponding $b_{i,j}$ values are updated accordingly.

This approach is also applicable when $m$ rows of matrix $A$ are generated in parallel. In this case, the value of $b_{i,j}$ is updated as follows:

$$b_{i,j} = b_{i,j} + s_{i,k} \cdot a_{k,j} + \cdots + s_{i,k+m-1} \cdot a_{k+m-1,j} \tag{2}$$

We first describe the general $8{\times}n{\times}m$ architecture for accelerating $S'A$ multiplication and then explain how, in our design, the factors $8\times, 2\times, 2\times$ (with $n = 2$ and $m = 2$) combine to yield an overall $32\times$ speed-up. This configuration reaches latencies of 0.928 ms for encapsulation and 0.95 ms for decapsulation using 20 BRAM blocks.

Since matrix $S$ has 8 columns, the multiplication $S' \cdot A$ can be parallelized into eight vector–matrix multiplications. Additional parallelism can be obtained in two ways: by increasing the number of elements $n$ taken from each row of $A$ per cycle, or by increasing the number of rows $m$ of $A$ processed simultaneously.

Using $n$ elements from each row of $A$ per cycle yields an $n$-fold speedup, but BRAM I/O limits require distributing $A$ across multiple dual-port blocks. The storage for $B'$ also grows with $n$ since each element must be read and updated during accumulation, leading to an area–throughput trade-off: $n$-fold acceleration costs $\lceil n/2 \rceil$ times more BRAM for $A$ and $n$ times more for $B'$. Increasing $n$ is further limited by SHAKE throughput, which sets an upper bound. Greater parallelism can instead be achieved by generating $m$ rows of $A$ in parallel using double buffering, giving an $m$-fold speedup but requiring $m$-times more BRAM for $A$ and $\lceil m/2 \rceil$ times more for $S'$. In total, the design performs $8nm$ multiplications per cycle, scaling DSPs and BRAM linearly with $8nm$, and thus offering a scalable area–throughput trade-off, determined directly by the choice of design-time parameter sets.

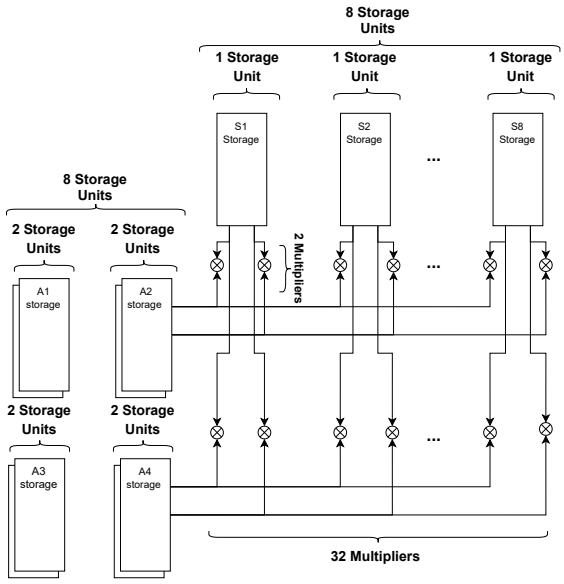



**Figure 2: Parallel $8 \times n \times m$ Architecture Instantiated with $n = m = 2$ for $S' \cdot A$ Multiplication**

In our proposed design, the parameters are chosen as $n = 2$ and $m = 2$. The $8\times$ gain comes from precomputing $S'$ and storing its rows in separate `S_storage` units, enabling parallel computation of all 8 rows of $B'$. A second SHAKE module ($m = 2$) doubles throughput by generating two rows of $A$ in parallel, while computing two columns of $B'$ per cycle ($n = 2$) further doubles the column-wise rate. Each parallel $B'$ element requires a dedicated dual-port BRAM for read and write. Thus, 16 units (0.5 BRAM each) are needed to sustain 16 updates per cycle, and together with the storage required for $S$ and $E$, this results in a total BRAM usage of 20 blocks for the complete design.

# 6 Results

This section presents the results of our FrodoKEM architecture. Tables 1–3 summarize performance for key generation, encapsulation, and decapsulation. All reported area and frequency figures for our design are post-implementation results. The proposed design was developed using Vivado 2023.1. Consistent with the other works listed in the comparison tables, all benchmarks were targeted to the Xilinx Artix-7 platform, except for the ones that are stated as Stratix IV in the tables. The **SC** column denotes *standard compliance*, i.e., whether an implementation follows the official

FrodoKEM specification. Among the listed works, only Howe et al. [9] is non-compliant, as it replaces SHAKE with Trivium for pseudorandomness generation.

For key generation, our 24-multiplier datapath achieves a latency of 1.024 ms, compared to 12–20 ms for the earlier specification-compliant designs [3, 10, 20] and 2.13 ms for the unified design of Manzoni et al. [15]. Howe et al. [9] report 1.19 ms with 16 multipliers, but their design replaces SHAKE with Trivium and is therefore not specification-compliant.

For encapsulation and decapsulation, our 32-multiplier datapath reaches 0.928 ms and 0.95 ms, respectively, using 32 DSPs and 20 BRAMs. In contrast, [3, 10, 20] require 19–32 ms, [15] requires 2.18 ms and 2.23 ms, and even the fastest design in [9] achieves only 1.31 ms with Trivium.

To normalize efficiency with respect to logic usage, we report the area–time metric defined as the product of the Equivalent Number of Slices (ENS) and the execution time (in seconds). The ENS value is computed as

$$\text{ENS} = \text{DSP} \times 100 + \text{BRAM} \times 196 + \text{Slices}, \tag{3}$$

where DSP and BRAM counts are converted into slice-equivalent units for Xilinx Artix-7 devices. Since [3, 4, 13, 15, 20, 21] do not report the number of slices, we estimate it using

$$\text{Slices} = \frac{\#\text{LUTs}}{4}, \tag{4}$$

because one slice contains four LUTs on Artix-7 FPGAs.

Among the designs that predate this work, our Area×Time value is significantly lower than that of the earlier specification-compliant designs [3, 10, 15, 20] and within 1.6–3.4× of the Trivium-based design [9], which benefits from a lighter PRNG. In particular, compared with the unified DSP-free design of Manzoni et al. [15], our implementation is 2.1–2.4× faster and achieves a lower Area×Time product in all three operations (9.95/11.31/11.99 versus 13.52/13.84/14.12), although their design uses fewer BRAMs (8) and no DSPs.

## 6.1 Comparison with Subsequent Designs

Tables 1–3 also include designs that appeared after the preprint of this work [4, 13, 21] (Section 2). All of them are unified cores supporting all three operations and all parameter sets, so a single area figure covers the complete KEM, whereas our figures are reported per operation.

At the same multiplier count ($T = 32$), the design of Deshpande et al. [4] is faster (0.72/0.77/0.80 ms), but it uses 48 BRAMs against our 18–20 and yields a higher Area×Time product (13.59/14.39/15.08 versus 9.95/11.31/11.99), even though its area excludes the SHAKE module. Li et al. [13] achieve lower latency and Area×Time by avoiding DSPs and reducing memory to 14 BRAMs. In cycle count, however, our datapaths require fewer cycles (approximately 150k/127k/127k, obtained as latency × frequency, versus 178k/181k/183k reported in [13]), so the latency difference stems mainly from their higher clock frequency (207 MHz versus 133–147 MHz). Zhu et al. [21] report the lowest latency (0.34–0.37 ms) using four parallel SHAKE instances, at the cost of 56,418 LUTs, 3.3–4.6× more than any of our datapaths.

Whether a unified or an operation-specific architecture is preferable depends on the deployment scenario. When all three primitives must be available on the same device at once, an operation-specific design pays permanently for its replicated multiplication logic, and a shared datapath uses the area better. When the primitives are used at different times, dynamic partial reconfiguration can load a specialized datapath into the same region on demand, so the resident area is the largest of the primitives rather than their sum. A unified architecture therefore shares hardware in space, whereas ours shares it in time.

In summary, our design achieves the lowest latency among the earlier specification-compliant designs, and subsequent works have built on and further improved these results.

# 7 Further Discussion

## 7.1 Comparison with ML-KEM

Table 4 reports the implementation results of ML-KEM-512, which targets the same NIST security level as FrodoKEM-640. ML-KEM attains a much better AT product and latency because its Module-LWE structure admits highly optimized number theoretic transform based arithmetic that reduces the multiplication complexity from $O(n^2)$ to $O(n \log n)$, whereas FrodoKEM relies on standard LWE over unstructured lattices, a more conservative design without algebraic assumptions that is regarded as potentially more robust against future cryptanalytic techniques. We therefore include this comparison mainly for completeness: although ML-KEM-512 and FrodoKEM-640 share a nominal security level, their mathematical foundations and hardware acceleration opportunities differ substantially, so a direct performance comparison is not strictly fair.

**Table 4: ML-KEM-512 [18]**

| | LUT | FF | Slices | DSP | BRAM | Freq. (MHz) | Ops/sec | Latency ($\mu$s) | Area×Time (ENS×Secs) |
|---|---|---|---|---|---|---|---|---|---|
| **KeyGen** | | | | | | | 188,679 | 5.3 | |
| **Encaps** | 13,999 | 12,503 | 5,273 | 2 | 0 | 208 | 138,888 | 7.2 | 0.12 |
| **Decaps** | | | | | | | 99,009 | 10.1 | |

## 7.2 Side-Channel Analysis

Our design follows the constant-time requirements of the specification: all loop bounds have fixed, public limits, and the final validity check and key selection are implemented with fixed latency, so execution time does not depend on secret data. However, all hardware components, including the DSP blocks, memories, and SHAKE core, exhibit data-dependent power and EM behavior, and no masking or hiding countermeasures were applied. The effect of our 24- and 32-multiplier datapaths, which may either amplify exploitable leakage or add algorithmic noise, has not been quantified. A dedicated leakage assessment, e.g., using TVLA, is left to future work.

# 8 Conclusion

In this work, we introduce a technique to accelerate FrodoKEM by parallelizing the matrix multiplication while strictly adhering to the original specification. An example implementation for FrodoKEM-640 demonstrates speeds of 976–1077 operations per second. By introducing a parameterized methodology based on $k, l, n$ and $m$

we enabled systematic exploration of area–time trade-offs and demonstrated an efficient realization on a Xilinx Artix-7 FPGA. The results confirm that focusing architectural optimizations on the matrix-multiplication datapath yields substantial performance gains without deviating from the FrodoKEM specification. Future work includes extending the proposed approach to other FrodoKEM parameter sets and device families, as well as exploring further optimizations under power and side-channel constraints.

## References


[1] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. 2024. FrodoKEM: Learning With Errors Key Encapsulation — Preliminary Standardization Proposal. https://frodokem.org/files/FrodoKEM_standard_proposal_20241205.pdf

[2] Joppe W. Bos, Olivier Bronchain, Frank Custers, Joost Renes, Denise Verbakel, and Christine van Vredendaal. 2023. Enabling FrodoKEM on Embedded Devices. *IACR TCHES* 2023, 3 (2023), 74–96. doi:10.46586/tches.v2023.i3.74-96

[3] Fabian Buschkowski. 2021. *Efficient Hardware Implementation of Post-Quantum Cryptography using High-Level Synthesis.* Master's Thesis. Chair for Embedded Security, Ruhr University Bochum, Bochum, Germany. Advisor: Georg Land, Supervisor: Prof. Dr.-Ing. Tim Güneysu.

[4] Sanjay Deshpande, Patrick Longa, and Jakub Szefer. [n. d.]. Accelerating FrodoKEM in Hardware. 881–907 pages. doi:10.46586/tches.v2026.i3.881-907

[5] Gökçe Düzyol, Muhammed Said Gündoğan, and Atakan Arslan. 2025. Can FrodoKEM Run in a Millisecond? FPGA Says Yes! *Cryptology ePrint Archive, Paper 2025/1312* (2025).

[6] Federal Office for Information Security, Germany. 2025. Cryptographic Mechanisms: Recommendations and Key Lengths. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf

[7] French Cybersecurity Agency. 2023. ANSSI Views on the Post-Quantum Cryptography Transition (2023 Follow-Up). https://cyber.gouv.fr/sites/default/files/document/follow_up_position_paper_on_post_quantum_cryptography.pdf

[8] Lewis Glabush, Patrick Longa, Michael Naehrig, Chris Peikert, Douglas Stebila, and Fernando Virdia. 2025. FrodoKEM: A CCA-Secure Learning With Errors Key Encapsulation Mechanism. Cryptology ePrint Archive, Paper 2025/1861. doi:10.62056/ayivom2hd

[9] James Howe, Marco Martinoli, Elisabeth Oswald, and Francesco Regazzoni. 2021. Exploring Parallelism to Improve the Performance of FrodoKEM in Hardware. *Journal of Cryptographic Engineering* 11, 4 (Nov. 2021), 317–327. doi:10.1007/s13389-021-00258-7

[10] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. 2018. Standard Lattice-Based Key Encapsulation on Embedded Devices. *IACR TCHES* 2018, 3 (2018), 372–393. doi:10.13154/tches.v2018.i3.372-393

[11] ISO/IEC. 2026. ISO/IEC 18033-2:2006/Amd 2:2026 – Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers – Amendment 2. International Organization for Standardization. https://www.iso.org/standard/86890.html

[12] Patrick Karl, Tim Fritzmann, and Georg Sigl. 2022. Hardware Accelerated FrodoKEM on RISC-V. In *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS).* IEEE, Prague, Czech Republic, 154–159. doi:10.1109/DDECS54261.2022.9770148

[13] Kai Li, Jiahao Lu, Fu Yao, Guang Zeng, Dongsheng Liu, Shengfei Gu, Zhengpeng Zhao, and Jiachen Wang. 2026. A High Performance and Efficient Post-Quantum Crypto-Processor for FrodoKEM. *arXiv preprint arXiv:2601.16500* (2026).

[14] Patrick Longa, Joppe W. Bos, Stephan Ehlen, and Douglas Stebila. 2025. FrodoKEM: Key Encapsulation from Learning With Errors. Internet-Draft, IETF. https://datatracker.ietf.org/doc/draft-longa-cfrg-frodokem

[15] Giuseppe Manzoni, Shekoufeh Neisarian, and Elif Bilge Kavun. 2025. An Optimized FrodoKEM Implementation on Reconfigurable Hardware. In *International Workshop on Lightweight Cryptography for Security and Privacy.* Springer, 97–113. doi:10.1007/978-3-032-15541-2_6

[16] National Institute of Standards and Technology. 2024. *ML-KEM: Module-Lattice-Based Key-Encapsulation Mechanism.* Federal Information Processing Standards Publication FIPS 203. National Institute of Standards and Technology. Accessed: 2025-06-26. doi:10.6028/NIST.FIPS.203

[17] Netherlands National Communications Security Agency. 2023. The PQC Migration Handbook. https://publications.tno.nl/publication/34641918/oicFLj/attema-2023-pqc.pdf

[18] Ziying Ni, Ayesha Khalid, Máire O'Neill, Weiqiang Liu, et al. 2023. HPKA: A high-performance CRYSTALS-Kyber accelerator exploring efficient pipelining. *IEEE Trans. Comput.* 72, 12 (2023), 3340–3353. doi:10.1109/TC.2023.3296899

[19] Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *35th FOCS.* IEEE Computer Society Press, Santa Fe, NM, USA, 124–134. doi:10.1109/SFCS.1994.365700

[20] Fernando Aparicio Urbano-Molano and Jaime Velasco-Medina. 2025. FrodoKEM Hardware Implementation for Post-Quantum Cryptography. *IEEE Latin America Transactions* 23, 10 (2025), 922–930. doi:10.1109/TLA.2025.11150633

[21] Jianchi Zhu, Yaodong Wei, Wenbo Kong, Xingrui Gu, and Jing Tian. 2026. A Fast, Configurable and DSP-Free Hardware Implementation of FrodoKEM. *Cryptology ePrint Archive, Paper 2026/1170* (2026).