High-Throughput GPU Implementation of Dilithium Post-Quantum Digital Signature

Shiyu Shen[1], Hao Yang[2], Wangchen Dai[3], Hong Zhang[1], Zhe Liu[4], and Yunlei Zhao[1]

[1] Fudan University, Shanghai, China
[2] Nanjing University of Aeronautics and Astronautics, Nanjing, China
[3] Sun Yat-sen University, Shenzhen, China
[4] Zhejiang lab, Hangzhou, China

Abstract. Digital signatures are fundamental building blocks in various protocols to provide integrity and authenticity. The development of the quantum computing has raised concerns about the security guarantees afforded by classical signature schemes. CRYSTALS-Dilithium is an efficient post-quantum digital signature scheme based on lattice cryptography and has been selected as the primary algorithm for standardization by the National Institute of Standards and Technology. In this work, we present a high-throughput GPU implementation of Dilithium. For individual operations, we employ a range of computational and memory optimizations to overcome sequential constraints, reduce memory usage and IO latency, address bank conflicts, and mitigate pipeline stalls. This results in high and balanced compute throughput and memory throughput for each operation. In terms of concurrent task processing, we leverage task-level batching to fully utilize parallelism and implement a memory pool mechanism for rapid memory access. We propose a dynamic task scheduling mechanism to improve multiprocessor occupancy and significantly reduce execution time. Furthermore, we apply asynchronous computing and launch multiple streams to hide data transfer latencies and maximize the computing capabilities of both CPU and GPU. Across all three security levels, our GPU implementation achieves over 160× speedups for signing and over 80× speedups for verification on both commercial and server-grade GPUs. This achieves microsecond-level amortized execution times for each task, offering a high-throughput and quantum-resistant solution suitable for a wide array of applications in real systems.

Keywords: Post-quantum cryptography · Digital signature · Dilithium· Parallel processing · GPU.

1 Introduction

Digital signature is a cryptographic primitive that ensures message integrity and authenticity. As a crucial component of information security, digital signature algorithms are widely adopted in various protocols and applications, including Transport Layer Security (TLS) and blockchain systems. However, the emergence of quantum computing threatens classical signature algorithms like RSA and ECDSA [26], which rely on the large integer factorization and discrete logarithm problems vulnerable to quantum attacks. While it is uncertain whether a powerful enough quantum computer will be developed within decades, it is crucial to investigate post-quantum digital signatures to ensure long-term security.

In 2016, the National Institute of Standards and Technology (NIST) launched the post-quantum cryptography (PQC) standardization project to standardize post-quantum digital signature algorithms and public-key encryption/key encapsulation mechanisms (KEM) [5]. By 2022, four algorithms, including one KEM and three signature schemes, were selected after three evaluation rounds. Among these, CRYSTALS-Dilithium [8, 13] was highlighted for its robust security and efficiency, with NIST recommending Dilithium as the preferred choice. Many real-world applications are considering deploying Dilithium for long-term protection. For example, ArielCoin [1], an experimental digital currency, uses Dilithium for verification and authentication; the liboqs library [2] presented by the Open Quantum Safe project integrates Dilithium for a quantum-safe TLS protocol. However, lattice-based schemes like Dilithium suffer from high computational and memory overhead, as well as large IO transfer sizes, which make them a performance bottleneck in server-based scenarios with vast numbers of queries. This underscores the need for optimized Dilithium implementations that prioritize high throughput and efficient memory use in server settings.

GPUs are commonly used for concurrent processing of signatures due to their massive parallelism. Numerous studies have demonstrated the high performance of GPU-based classical signatures and post-quantum KEMs [15, 17, 19, 23, 28]. However, research on post-quantum signatures remains limited [25, 27]. Existing methods often use single or partial threads in a warp to execute tasks, which are inefficient for minimizing IO latency due to uncoalesced data accesses. Studies on the Dilithium algorithms software [4, 11, 13, 16, 24] and hardware implementations [9, 12, 31] focus primarily on compact design, which does not adequately address throughput and real-time demands.

Contributions. In this work, we present a high-throughput GPU implementation of Dilithium. Our main contributions can be summarized as follows:

– For all operations, we develop optimized implementations to enhance performance. Specifically, we parallelize numerous sequential operations in rejection sampling by leveraging CUDA integer intrinsics and warp-level primitives, optimize memory access patterns in number-theoretic transforms, and minimize resource usage in hash functions and inner-product calculations. Additionally, we incorporate a memory pool for efficient memory management.
– We introduce several optimizations to minimize IO latency and improve resource utilization based on the profiling results of our implementation. First, we propose a finely-tuned fusing strategy that strikes a balance between low IO latency and high resource utilization, resulting in optimal performance. Second, we introduce a dynamic task scheduling mechanism to address the occupancy decrease issue when batching multiple signings, which significantly improves resource utilization and reduces execution time. Besides, we asynchronize the computation and launch multiple CUDA streams to hide data transfer latency between the CPU and GPU, fully utilizing the computational capabilities of both CPU and GPU.
– We deploy our implementation on three representative GPUs with varying computing capabilities. For the signing procedure, we achieve 397k to 766k OP/s throughput on a server-grade NVIDIA Tesla A100 GPU, and 488k to 985k OP/s throughput on a desktop NVIDIA RTX 4090 GPU, marking at least a 166$\times$ speedup over single-thread CPU implementation.

Code. Our code is publicly available at `https://github.com/encryptorion-lab/cuDilithium`.

Related Works. Research on accelerating PQC schemes, specifically KEMs [15,17,19,23,28] and digital signatures [25, 27], has adopted two primary computational approaches. The first uses a single thread for all tasks, while the second explores task parallelism using a warp or partial threads within a warp for concurrent execution. In the first approach, Gupta et al. [17] implement both methods in three KEM schemes. Gao et al. [15] examine more granular parallelism with thread counts of 8, 16, and 32 in their NewHope [7] acceleration, comparing latency and throughput across these configurations. However, [17] obtain limited speedups due to not batching tasks, which results in underutilized GPU resources. Meanwhile, this approach makes memory access within a warp uncoalesced, increasing IO latency. The second approach, applied by multiple studies [19, 23, 25, 27, 28], uses a warp for each task execution, typically achieving only about 33% of theoretical streaming multiprocessor (SM) occupancy, leaving substantial GPU resources idle. Specifically, Seo et al. [25] introduced acceleration of Dilithium in an autonomous driving context, adopting an unconventional approach by altering the rejection sampling to use a look-up table for element loading. This method increases memory overhead by requiring additional table inputs, which must be transferred along with the public key and signature, potentially leading to compatibility issues and deviating from the design goal of minimizing communication bandwidth. Further, works like those by Lee et al. and Wan et al. [19,28] have explored using Tensor Cores, necessitating the division of elements into 8-bit segments. While effective for schemes with small data sizes, this approach can be resource-intensive for Dilithium, potentially leading to high overhead.

2 Preliminaries

2.1 Notation

Let $\mathbb{Z}$ be the group of integers. We define $\mathbb{Z}_q$ with its representation in the interval $\mathbb{Z} \cap [-\frac{q}{2}, \frac{q}{2})$, where q is a prime. Let n be a power of 2, we denote $R = \mathbb{Z}[X]/(X^n + 1)$ as the $2n$-th cyclotomic ring and $R_q = R/qR$

Algorithm 1 Dilithium.Gen

```
Output: $pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
1: $\zeta \leftarrow \{0,1\}^{L_1}$
2: $(\rho, \rho', K) \in \{0,1\}^{L_1 \times L_2 \times L_1} := \mathcal{H}(\zeta)$
3: $\hat{\mathbf{A}} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$
4: $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k := \mathsf{ExpandS}(\rho')$
5: $\mathbf{t} := \mathsf{INTT}(\hat{\mathbf{A}} \cdot \mathsf{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_q(\mathbf{t}, d)$
7: $tr \in \{0,1\}^{L_1} := \mathcal{H}(\rho \| \mathbf{t}_1)$
8: $pk := (\rho, \mathbf{t}_1)$
9: $sk := (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
```

as its residue ring modulo q. Lowercase letters represent ring elements, such as $\boldsymbol{f} = \sum_{i=0}^{n-1} f_i X^i$, and vectors, such as $\mathbf{v}$. Bold uppercase letters represent matrices of polynomials, e.g., $\mathbf{A}$. The hat symbol indicates elements in the frequency domain.

The notation $\bmod^{\pm}\alpha$ refers to centered reduction modulo α, outputting values in $(-\lfloor\frac{\alpha+1}{2}\rfloor, \lfloor\frac{\alpha}{2}\rfloor]$. The $\|\cdot\|_\infty$ represents the ℓ_∞-norm, and $[\cdot]$ counts coefficients equal to 1. The operator $\lfloor\cdot\rfloor$ signifies flooring, and $[\cdot]_q$ denotes modular reduction by q. The set $\{0,1\}^l$ indicates an l-bit stream; $\{0,1\}^*$ denotes bit streams of arbitrary length, with $|\cdot|$ showing their bit-length. The operator $\|$ concatenates bit streams converted from elements. Uniform sampling from a finite set S is denoted as $a \leftarrow S$. We define $S_\eta := \{\omega : \omega \in R, \|\omega\|_\infty \leq \eta\}$ and $\tilde{S}_\eta := \{\omega \bmod {}^{\pm}2\eta : \omega\}$. The operator $[\![\mathcal{P}]\!]$ returns 1 if $\mathcal{P}$ is true and 0 otherwise.

2.2 Ring Arithmetic

Elements in $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ are represented as polynomials of degree less than n. Modular reduction is essential for reducing the size of large integers. The Montgomery reduction [22] and Barrett reduction [10] are two techniques employed for fast and constant-time modular reduction, by replacing the time-consuming division with faster multiplication or bit shift operations.

Multiplication of $\boldsymbol{f} = \sum_{i=0}^{n-1} f_i X^i$ and $\boldsymbol{g} = \sum_{j=0}^{n-1} g_j X^j$ over R_q yields a polynomial $\boldsymbol{h} = \sum_{t=0}^{n-1} h_t X^t \in R_q$, where the coefficients $(h_0, \ldots, h_{n-1})$ are the negacylic convolution of $(f_0, \ldots, f_{n-1})$ and $(g_0, \ldots, g_{n-1})$. Direct multiplication via the schoolbook method has a complexity of $\mathcal{O}(n^2)$, posing a significant performance bottleneck. To accelerate this, the Number-Theoretic Transform (NTT), a variant of the Discrete Fourier Transform (DFT) for integers modulo a prime number, is used. Denoting ψ as the primitive $2n$-th root of unity, the forward and inverse negacyclic NTT for $\boldsymbol{f} \in R_q$ are formulated as $\hat{\boldsymbol{f}} := \mathsf{NTT}(\boldsymbol{f})$ and $\boldsymbol{f} := \mathsf{INTT}(\hat{\boldsymbol{f}})$, where $\hat{f}_j = \sum_{i=0}^{n-1} f_i \psi^{(2i+1)j} \pmod{q}$ and $f_i = \frac{1}{n}\sum_{j=0}^{n-1} \hat{f}_j \psi^{-(2i+1)j} \pmod{q}$. Using NTT, the multiplication is performed as $\boldsymbol{f} \cdot \boldsymbol{g} := \mathsf{INTT}(\mathsf{NTT}(\boldsymbol{f}) \cdot \mathsf{NTT}(\boldsymbol{g}))$. This reduces the complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, as both NTT and INTT have $\mathcal{O}(n \log n)$ complexity.

2.3 CRYSTALS-Dilithium

Dilithium is a lattice-based digital signature scheme, which provides security in the Quantum Random Oracle Model (QROM) under the Module Learning With Errors (MLWE) and a variant of the Module Short Integer Solution (MSIS) assumptions [8, 13]. It employs the *Fiat-Shamir with Aborts* method [20, 21], allowing the secret key holder to prove knowledge without revealing it by generating commitments and responding to random challenges.

The scheme comprises three procedures: key generation (Gen), signing (Sign), and verification (Verify), detailed in Algorithm 1, 2, and 3, respectively. Table 1 lists the parameters of Dilithium for three NIST security levels. As a lattice-based scheme, Dilithium exhibits robust post-quantum security, making it ideal for long-term cryptographic applications. Below are the specifications of the components used in it.

Sampling. In Dilithium, SHAKE128/256 [14] serves as the extendable-output function (XOF) to generate sufficient random bytes from input seeds, while SHAKE256 is used to instantiate the hash function $\mathcal{H}$.

Algorithm 2 Dilithium.Sign

```
Input: $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$, $M \in \{0,1\}^*$
Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
 1: $\mu \in \{0,1\}^{L_2} := \mathcal{H}(tr \| M)$
 2: $\rho' \in \{0,1\}^{L_2} := \mathcal{H}(K \| \mu)$
 3: $\hat{\mathbf{A}} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho); \kappa := 0; (\mathbf{z}, \mathbf{h}) := \perp$
 4: $\hat{\mathbf{s}}_1 := \mathsf{NTT}(\mathbf{s}_1); \hat{\mathbf{s}}_2 := \mathsf{NTT}(\mathbf{s}_2); \hat{\mathbf{t}}_0 := \mathsf{NTT}(\mathbf{t}_0)$
 5: while $(\mathbf{z}, \mathbf{h}) = \perp$ do
 6:     $\mathbf{y} \in S_{\gamma_1}^{\ell} := \mathsf{ExpandMask}(\rho', \kappa)$
 7:     $\mathbf{w} := \mathsf{INTT}(\hat{\mathbf{A}} \cdot \mathsf{NTT}(\mathbf{y}))$
 8:     $\mathbf{w}_1 := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$
 9:     $\tilde{c} \in \{0,1\}^{L1} := \mathcal{H}(\mu \| \mathbf{w}_1)$
10:     $\boldsymbol{c} \in B_\tau := \mathsf{SampleInBall}(\tilde{c}); \hat{\boldsymbol{c}} := \mathsf{NTT}(\boldsymbol{c})$
11:     $\mathbf{z} := \mathbf{y} + \mathsf{INTT}(\hat{\boldsymbol{c}} \cdot \hat{\mathbf{s}}_1)$
12:     $\mathbf{v}_s := \mathsf{INTT}(\hat{\boldsymbol{c}} \cdot \hat{\mathbf{s}}_2)$
13:     $\mathbf{r}_0 := \mathsf{LowBits}_q(\mathbf{w} - \mathbf{v}_s, 2\gamma_2)$
14:     if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ then
15:         $(\mathbf{z}, \mathbf{h}) := \perp$
16:     else
17:         $\mathbf{v}_t := \mathsf{INTT}(\hat{\boldsymbol{c}} \cdot \hat{\mathbf{t}}_0)$
18:         $\mathbf{h} := \mathsf{MakeHint}_q(-\mathbf{v}_t, \mathbf{w} - \mathbf{v}_s + \mathbf{v}_t, 2\gamma_2)$
19:         if $\|\mathbf{v_t}\|_\infty \geq \gamma_2$ or $[\mathbf{h}] > \omega$ then
20:             $(\mathbf{z}, \mathbf{h}) := \perp$
21:     $\kappa := \kappa + \ell$
```

Algorithm 3 Dilithium.Verify

```
Input: $pk = (\rho, \mathbf{t}_1), M \in \{0,1\}^*, \sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
Output: $r$
 1: $\hat{\mathbf{A}} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
 2: $\mu \in \{0,1\}^{L_2} := \mathcal{H}\left(\mathcal{H}\left(\rho \| \mathbf{t}_1\right) \| M\right)$
 3: $\boldsymbol{c} := \mathsf{SampleInBall}(\tilde{c}); \hat{\boldsymbol{c}} := \mathsf{NTT}(\boldsymbol{c})$
 4: $\mathbf{v} := \mathsf{INTT}(\hat{\mathbf{A}} \cdot \mathsf{NTT}(\mathbf{z}) - \hat{\boldsymbol{c}} \cdot \mathsf{NTT}(\mathbf{t}_1 \cdot 2^d))$
 5: $\mathbf{w}_1' := \mathsf{UseHint}_q\left(\mathbf{h}, \mathbf{v}, 2\gamma_2\right)$
 6: $r := [\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!] \& [\![\tilde{c} = \mathcal{H}(\mu \| \mathbf{w}_1')]\!] \& [\![[\mathbf{h}] \leq \omega]\!]$
```

A *rejection sampling* mechanism is applied to ensure sequences follow a uniform distribution by selecting $|\mathcal{B}|$ bits per sample and retaining only those less than $\mathcal{B}$. Using this mechanism, the ExpandA function produces a matrix $\hat{\mathbf{A}}$ with coefficients in the range $[0, q)$, and the ExpandS and ExpandMask functions generate vectors with coefficients in the ranges $[-\eta, \eta]$ and $[-\gamma_1 + 1, \gamma_1]$, respectively. The SampleInBall function generates a sparse polynomial c with τ nonzero coefficients by selecting τ valid positions for nonzero integers, achieved by comparing a random byte with the current loop index.

Bits Extraction and Hints. To reduce communication bandwidth, Dilithium utilizes optimizations that compress both the public key and signature, which can occasionally cause verification failures. To mitigate this, a hint is incorporated into the signature to ensure robustness. The following functions are used to compute high- and low-order bits and the hint. The $\mathsf{Power2Round}_q(a, d)$ function divides an integer a into $(a_0, a_1) := (a \bmod {}^{\pm}2^d, (a - a_0)/2^d)$. The $\mathsf{LowBits}_q(r, \alpha)$ and $\mathsf{HighBits}_q(r, \alpha)$ functions extract the low- and high-order bits r_1 and r_0, respectively. By evaluating $\mathcal{P} := (r - r_0 = q - 1)$, where $r_0 := (r \bmod {}^{+}q) \bmod {}^{\pm}\alpha$, the outputs are $(r_1, r_0) := (0, r_0 - 1)$ if $\mathcal{P}$ is true, and $(r_1, r_0) := ((r - r_0)/\alpha, r_0)$ otherwise. The $\mathsf{MakeHint}_q(z, r, \alpha)$ computes $r_1 := \mathsf{HighBits}_q(r, \alpha)$ and $z_1 := \mathsf{HighBits}_q(r + z, \alpha)$, returning $[\![r_1 \neq v_1]\!]$. The $\mathsf{UseHint}_q(h, r, \alpha)$ computes $m := (q - 1)/\alpha$, extracts (r_1, r_0), and outputs r_1 if $h = 0$. Otherwise, it returns $(r_1 + 1) \bmod {}^{+}m$ if $r_0 > 0$, and $(r_1 - 1) \bmod {}^{+}m$ if $r_0 \leq 0$. All these can be extended to ring elements by applying them coefficient-wise.

Table 1: Parameter specifications of Dilithium for three NIST security levels

Level	n	q	(k, ℓ)	d	τ	γ_1	γ_2	η	β	ω	L_1	L_2
2	256	8380417	(4,4)	13	39	2^{17}	95232	2	78	80	256	512
3	256	8380417	(6,5)	13	49	2^{19}	261888	4	196	55	256	512
5	256	8380417	(8,7)	13	60	2^{19}	261888	2	120	75	256	512

Rejection Loop. During signing, a rejection stage is necessary to ensure the generated $\mathbf{z}$ does not reveal information about sk. The security requirement fails if $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$. Additionally, if any coefficient of the low-order bits of $\mathbf{A}\mathbf{y} - c\mathbf{s}_2$ exceeds $\gamma_2 - \beta$, security and correctness are compromised. The signing loop repeats until these conditions are met, with the expected number of repetitions being 4.25, 5.1, and 3.85 for the three security levels, respectively.

2.4 GPU Basics

In the CPU-GPU collaborative computing model, the CPU acts as a host, dispatching kernels to the GPU. In synchronous computing, the CPU waits for the GPU to complete its tasks, while in asynchronous computing, the CPU continues with other tasks. Figure 1 illustrates the architecture and computational model.

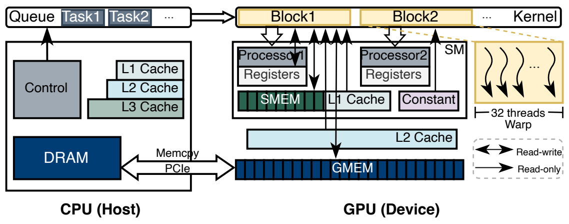


Fig. 1: The architecture and host-device computational model.

Modern GPUs facilitate highly parallel kernel processing using numerous concurrent threads. The CUDA programming model offers a direct interface for accessing hardware resources. Threads are organized into thread blocks, with multiple blocks forming a grid. GPUs contain various memory types, including registers, constant memory (CMEM), shared memory (SMEM), global memory (GMEM), local memory (LMEM), etc. The memory hierarchy ensures low-latency data access, with registers offering the fastest access, and LMEM having the same latency as GMEM. Threads within a block share SMEM, which has higher bandwidth and lower latency than local or global memory. GMEM, accessible to all threads, has the highest IO latency and is accessed through the SM L1 cache and GPU L2 cache.

During execution, blocks are assigned to streaming multiprocessors (SM), with every 32 threads forming a warp for parallel processing. Occupancy, the ratio of active warps per SM to the maximum possible number of active warps, reflects resource utilization. CUDA offers warp-level primitives like *warp shuffle* for synchronized register data exchange and *warp vote* for combining values across threads in a tree-reduction pattern and broadcasting the result.

Instructions are executed in warps, with optimal throughput achieved when all threads in a warp execute the same instruction. Divergence reduces active threads per cycle and performance. Different types of instructions are scheduled to specific pipelines; for example, the Arithmetic Logic Unit (ALU) handles most bit manipulation and logic instructions, while the Load Store Unit (LSU) issues load and store instructions for memory access.

3 Design Overview

3.1 Implementation Challenges and Bottlenecks

Dilithium involves extensive matrix and vector operations, encompassing both linear and complex operations with specific data paths. Accelerating it presents several challenges:

- **Massive Sequential Operations.** Many Dilithium operations, such as rejection sampling and number counting, require threads to track the state of their neighbors. These computations involve comparing elements with a preset bound, storing valid ones, and counting them. The results depend on the validity of previous elements, introducing sequential computations that are difficult to parallelize. This data-dependent conditional computation can lead to an unbalanced workload among threads, resulting in thread divergence and decreased performance.
- **Specific Datapath.** Numerous Dilithium operations have specific data paths. For instance, in NTT, data accessed by threads is interleaved, with changing intervals as computation progresses. This can cause issues like uncoalesced memory access and bank conflicts, increasing latency. Additionally, manipulating bit streams, such as compression, concatenation, and expansion, is required. Efficient methods for processing compressed data [30] is not suitable for evaluation over finite field here, which presents challenges like stride reads and writes, alignment, and balancing thread workloads.
- **High Memory Consumption.** Dilithium involves extensive linear operations on matrices and vectors, along with memory-intensive operations like hash functions. This leads to high memory overhead and IO latency-dominant computations. Previous implementations used registers to improve access speed, but excessive register usage results in increased writes to local memory with slower access speeds. Furthermore, excessive memory requests can overutilize the memory input/output (MIO) pipeline, causing warp stalls.

3.2 Operation-level Design Rationale

Warp-level implementation, which is widely used in previous works, offers substantial benefits such as coalesced memory access and IO latency reduction. The sequential computations make a warp an ideal unit for sharing registers among threads. Thus, we adopt this design. We divide operations in Gen, Sign, and Verify into several computational tasks, each corresponding to a block. A kernel batches multiple blocks to process same tasks simultaneously, providing high-throughput computation of Dilithium instances. We exploit the following designs for blocks:

- **Single Warp Design.** Each block is initialized with one warp. The SM occupancy under this configuration is suboptimal. For example, with a compute capability of 8.6 and an equal partition between the L1 cache and SMEM, the maximum occupancy of each multiprocessor hits only 33%, with a warp occupancy of 16%. However, this design facilitates parallelizing sequential operations, and reduces waste in hash functions where parallelism is limited to 25.
- **Quartic Warps Design.** Using 3 or 4 warps per block achieves 100% theoretical occupancy. However, 3 warps are not compatible with all Dilithium parameter sets. Using 4 warps is ideal, providing a maximum of 40 registers per thread and 4 KB of shared memory per block. This design offers better performance for most arithmetic operations.

Our implementation combines both designs, denoted as “SWarp” and “QWarp” in the following context. This provides flexibility in choosing the most appropriate implementation based on specific circumstances, ensuring optimal performance and compatibility when fusing operations.

3.3 Batching Multiple Tasks

To achieve high-throughput implementation and fully utilize GPU resources, we batch multiple tasks in the implemented kernels to process multiple Dilithium instances simultaneously. The following aspects is taken into account:

– **Memory Management.** A pooling mechanism ensures efficient and secure memory access. Meanwhile, it should be fine-tuned for data alignment and optimal storage sequences to process bit stream.
– **Task Scheduling.** Unlike constant-time schemes, the randomness in Dilithium causes varying repetitions in the signing procedure across different instances. When batching multiple signing instances, some execution units may become idle while waiting for others to complete if tasks are bound to units. Therefore, an efficient task scheduling mechanism is needed to minimize hardware resource waste.
– **Streaming to Hide Latency.** Streaming, a common technique in batching processes, asynchronizes computation to hide data transfer latency. At the same time, we should remain the synchronization between CPU and GPU, which is required in task scheduling.

4 Implementation Details

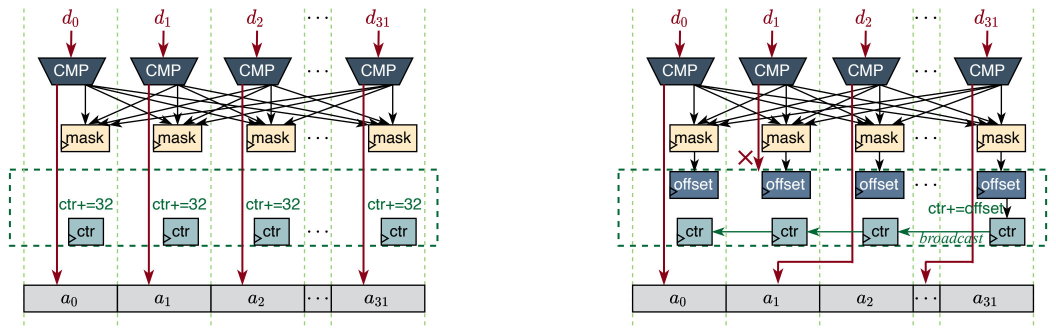
In this section, we detail our GPU implementation, covering NTT, hash functions, rejection sampling, and inner-product. By optimizing these operations for GPU architecture, we aim to maximize parallelism and enhance overall performance. Below, we discuss the specific techniques employed and the innovative strategies adopted to overcome these obstacles.

4.1 Rejection Sampling

Parallelizing sequential computations of rejection sampling and number counting is a major challenge in accelerating Dilithium. These operations involve comparing elements with a preset bound and storing valid ones. The main goal is to share states among threads to perform data-dependent computations and avoid unbalanced workloads. Thus, we employ CUDA integer intrinsics and warp-level primitives to ensure all threads perform the same operation. Algorithm 4 provides our implementation of rejection sampling in ExpandA function. We set a predicate argument in each thread to record the comparison result so that the entire state of the warp can be obtained through warp voting. If all 32 arguments are valid like Fig. 2a, no additional computation is required. Otherwise, each thread computes its local inclusive state by an integer intrinsic function to get the offset of the write address as in Fig. 2b. As only the local counter of the last thread captures correct total numbers, we use warp shuffle synchronization to broadcast it to all threads for correctness in subsequent iterations.

Algorithm 4 Optimized Rejection sampling in ExpandA

```
1: __shared__ $\mathbf{s}[n], \mathbf{buf}[len]$
2: $ctr := 0$
3: for $round \in [0, 8)$ do
4:     $pos := round * 32 * 3 + \texttt{threadIdx.x} * 3$
5:     $t := \mathbf{buf}[pos : pos + 2]$
6:     $sign := [(t - q) \gg 31] \& 1$                              ▷ Compare with $q$
7:     $mask :=$ __ballot_sync(0xFFFFFFFF, $sign$)
8:     if $mask ==$ 0xFFFFFFFF then                                 ▷ All accept
9:         $\mathbf{s}[ctr + \texttt{threadIdx.x}] := t$
10:        $ctr := ctr + 32$
11:    else                                                         ▷ Reject
12:        $mask := mask \ll (31 - \texttt{threadIdx.x})$
13:        $offset :=$ __popc($mask$)
14:        if $(ctr + offset \leq n) \& sign$ then
15:            $\mathbf{s}[ctr + offset - 1] = t$
16:        $ctr := ctr +$ __shfl_sync(0xFFFFFFFF, $offset$, 31)
17: __syncwarp()
```


d_0 d_1 d_2 ··· d_{31}
CMP
mask
ctr+=32
ctr
a_0 a_1 a_2 ··· a_{31}
offset
ctr+=offset
broadcast

(a) All accept. (b) Rejection.

Fig. 2: Computations in rejection sampling and number counting. Note number counting does not need to write d_i.

4.2 Hash Functions

We employ warp-level primitives to implement a warp-based SHAKE128/256 following the methodology described in [23]. However, [23] has limitations. The excessive register usage leading to increased local memory writes with slow access speeds. Additionally, each thread accesses a 64-bit value through eight consecutive 8-byte reads (or writes), causing strided GMEM access. Consequently, this leads to uncoalesced memory access, increased IO latency, and overutilization of the memory input/output (MIO) pipeline that forces warps to stall.

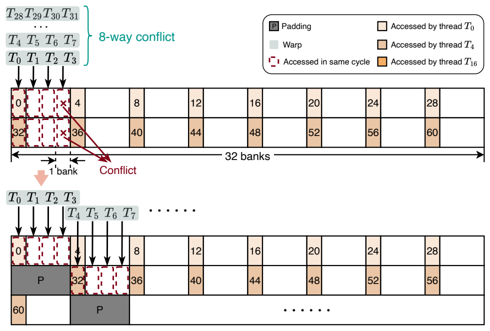
To address these issues, instead of precomputing and loading all constants into registers [23], we compute some constants during execution and store round constants in CMEM, reducing GMEM access. We align input and output streams during absorb and squeeze phases, replacing byte-wise operations in [23] with a single GMEM access. This adjustment results in wider but fewer loads and stores, reducing pipeline pressure. We further refine the computational flow, especially branches handling insufficient inputs, which lessens thread divergence and improves performance. In the internal state permutation function, 24 permutation rounds are executed by 25 threads using warp-shuffle to access states in other threads. Our optimizations significantly enhance performance, throughput, and occupancy, significantly reducing memory use and access.

4.3 Number-Theoretic Transform

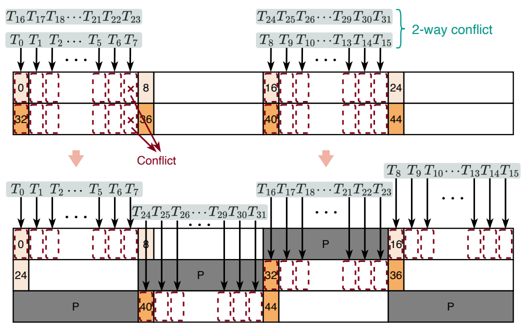
We follow [8] and apply the negacyclic NTT with Cooley-Tukey and Gentleman-Sande algorithms for fast polynomial multiplication. Transforming an n-dimensional polynomial requires $\log n$ levels. With Dilithium fixed at $n = 256$, we implement in-place constant-time 8-level NTT and INTT, utilizing both SWarp and QWarp for different scenarios.

In QWarp, we utilize 128 threads for each $\mathsf{NTT}/\mathsf{INTT}$ and devise a 2-per-thread implementation that performs a radix-2 butterfly operation on two coefficients at a time. The distance between coefficient indices is 2^{8-i} at level ith, where $i \in [1, 8]$. Between levels, we use SMEM to store temporal outputs for data exchange and prepare input for next stage. The SWarp approach trades more registers for faster data access and reduced IO latency. Each thread loads eight coefficients into registers and performs a radix-8 $\mathsf{NTT}/\mathsf{INTT}$, forming merged three-level [6] processing. We then use SMEM for data exchange and access elements required by following three levels. Data transfers between SMEM and registers are only needed before the 4th and 7th levels, where the eight loaded coefficients are either continuous or at an interval of 4. Considering the high access frequency of the pre-computed roots in batch computations, we cache the table in SMEM to offer low latency load.

To optimize each level individually, we unroll inner loops of the 8 levels in both versions while finely tuning execution flow to reduce pipeline stalls. For QWarp, since processed coefficients fall within same



(a) Solving 8-way conflicts between the 3rd and 4th levels in SWarp by padding 4 units every 32 coefficients.



(b) Solving 2-way conflicts between the 4th and 5th levels in QWarp by padding 8 units every 16 coefficients.

Fig. 3: Strategies to solve 8-way and 2-way conflicts in SWarp and QWarp. The SMEM units are organized into 32 memory banks. Bank conflicts occur when multiple threads access the same memory bank, causing serialized accesses.

warp during last five levels of NTT (and the first five levels of INTT), we can reduce thread synchronization instructions between these levels. Additionally, we fuse the multiplications of n^{-1} and roots at last level of INTT to reduce number of Montgomery multiplications.

Another observation is that strided SMEM accesses can cause bank conflicts where multiple threads access the same memory bank. To prevent stalls and maintain memory throughput, we carefully pad SMEM so that accesses within a warp fall into individual banks. Fig. 3 presents visualization of positions that bank conflicts are triggered in NTT and our solution to avoid it. Below, we describe only NTT for simplicity, as the situation is similar for INTT. In the two data exchange stages of SWarp, eight threads issue instructions to access units in same bank, causing 8-way bank conflicts. Therefore, we pad four units every 32 coefficients in first exchange stage and one unit every eight coefficients in second stage. In QWarp, 2-way conflicts exist at last five levels of NTT (and the first five levels of INTT). At ith level ($i \in [3, 7]$), we pad 2^{7-i} units every 2^{8-i} coefficients when writing to SMEM to ensure conflict-free load in next level. Through these approaches, we reduce overall pipeline stalls and improve compute and memory throughput for both NTT and INTT.

4.4 Inner-Product Computation

The three procedures in Dilithium require computing the inner-product of a $k \times \ell$ matrix $\mathbf{A}$ with a $\ell \times 1$ vector, denoted as $\mathbf{As}_1$, $\mathbf{Ay}$, and $\mathbf{Az}$, respectively. Since $\mathbf{A}$ is stored in NTT domain, this operation entails transforming the vector to the NTT domain, performing point-wise multiplication and accumulation, and transforming back to the normal domain. However, the choice of computational flow and memory type for storing elements affects memory consumption, IO latency, and SM occupancy, which in turn significantly impacts performance. Meanwhile, the polynomial nature of matrix and vector elements precludes the use of optimized libraries like cuBlas [3]. This makes us devise specific optimizations for such scenario.

Column-Major and On-the-Fly Computation. Traditional inner-product computation uses a row-major approach that caches the entire ℓ-dimensional vector and requires ℓ KB storage in SMEM. This can lead to excessive SMEM consumption and reduced SM occupancy. Instead, we apply a column-major approach, which stores an accumulator of dimension k. Here, each polynomial in vector is loaded at a time using 2 registers per thread. We also allocate $2k$ registers as accumulators in each thread for the results. This method offers another advantage. For $\mathbf{As}_1$ and $\mathbf{Az}$, since the multiplication occurs only once, storing the entire $\hat{\mathbf{A}}$ is unnecessary. Thus, we generate $\hat{\mathbf{A}}$ in column order and sample one polynomial of $\boldsymbol{s}_{1,j}$ (or unpacked $\boldsymbol{z}_j$) at a time, multiplying the corresponding elements and accumulating the results. This on-the-fly computation eliminates the need for additional memory allocation beyond a single buffer used as an accumulator to store k polynomials during processing. Moreover, by computing polynomials one at a time, we can fuse operations

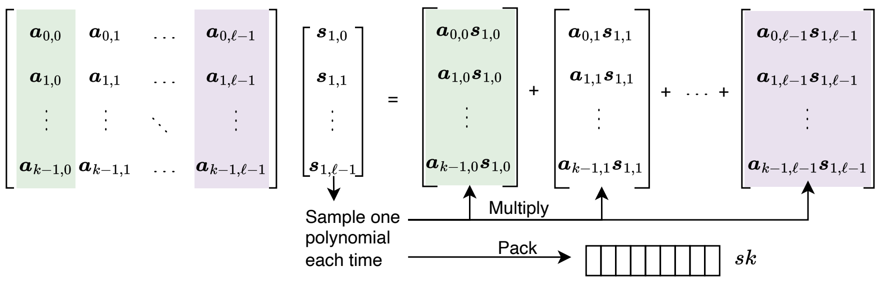


Fig. 4: Column-major computation of $\mathbf{As}_1$ in Gen.

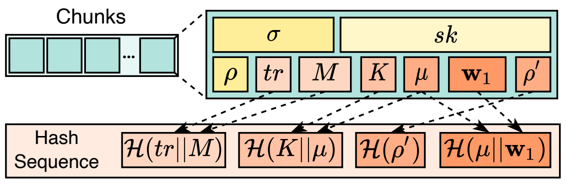


Fig. 5: The implemented task-level memory pool (example in Sign) that stores the elements needed in a task processing.

such as sampling, packing, and unpacking into the process, enabling data reuse and reducing data transfers. Figure 4 illustrates column-major computation in Gen function and demonstrates how packing of the secret key can be fused with this process.

4.5 Memory Pool

To ensure efficient memory access during concurrent task processing, we implement a task-level memory pool with a fixed size. This mechanism addresses two key considerations: element storage order and alignment requirements.

First, we arrange the order of elements for signing and verification processes according to the flow of hash functions. This arrangement ensures contiguous addresses for input byte streams, as shown in Fig. 5, eliminating overhead from stream concatenation. Second, we meet alignment requirements using pitch allocation for linear memory storage of seeds and streams. Moreover, since the granularity of L2 memory requests, such as an L1TEX request, is a 128-byte cache line (comprising 4 consecutive 32-byte sectors per L2 request), we align the streams to 256 bytes, which include the combinations of seeds for the hash calls. This alignment improves the memory request pattern to the L2 cache line.

Implementing this mechanism reduces the costs of frequent memory allocation and deallocation, enhancing performance. This approach further optimizes memory management, enabling efficient handling of multiple parallel tasks while maintaining high performance.

5 Optimizations for Task Resource Usage

This section outlines optimizations to enhance memory and resource usage, address bottlenecks, and ensure efficient hardware utilization. We analyze profiling results, introduce fusing strategies, and explore task scheduling to maximize parallelism. These optimizations significantly enhance performance and resource efficiency in our implementation.

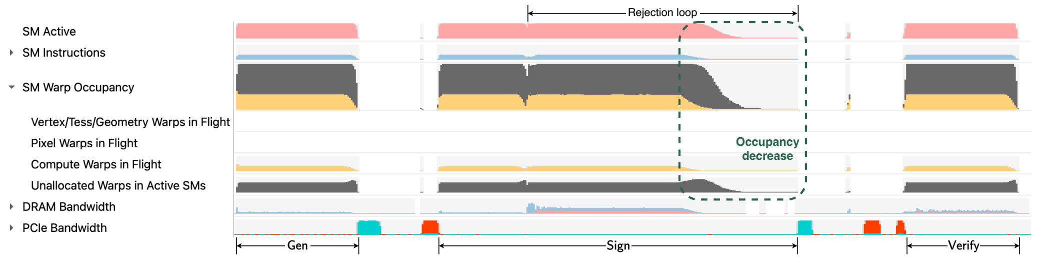


Fig. 6: Status of the SM during batch execution of multiple tasks (SWarp). During the latter part of the signing process, the hardware resource utilization is insufficient.

5.1 Profiling Results and Analysis

We profiled the implemented kernels using NVIDIA Nsight Compute and System tools, which revealed several areas for improvement in resource usage.

High Consumption of SHAKE. Despite being well-optimized, the SHAKE implementation consumes substantial resources. First, warp-level processing results in a high number of registers per thread, as each thread must load multiple elements, while using alternative memory types could increase latency. Second, the high volume of memory requests increases pipeline pressure. Third, occupancy levels are suboptimal. While batching, such as having four warps in a block with one warp computing a SHAKE, could improve occupancy, this method is not easily compatible with the early evaluation technique [24].

Waste of Hardware Resource in Rejection Loop. Figure 6 illustrates the states of SMs when batching Gen, Sign, and Verify instances. The SM occupancy decreases when the processing bar reaches 60% of the Sign procedure, indicating most tasks are completed and corresponding blocks become idle. While the average number of repeat rounds is around 4, some worst cases require dozens of rounds, leaving many blocks idle waiting for a few tasks to finish. This inefficiency in hardware resource usage needs addressing.

Insufficient Use of CPU. Synchronization leads to waste of CPU resources by enforcing a predetermined execution order. During processing, the CPU hosts the kernels to the GPU and waits for the results, which gives rise to two issues. First, the GPU waits for the CPU to transfer data, and subsequent CPU tasks cannot proceed until preceding ones finish, which underutilize the CPUs computational capability and reducing performance. Second, synchronization prevents hiding data transfer latency, hindering the overlap of computation and communication. This inefficiency is particularly detrimental for GPU computing, where data transfers are significant bottlenecks. Addressing these issues is crucial for fully harnessing both side in modern systems.

5.2 Finely-Tuned Fusing Strategy

Accelerating each operation separately is straightforward but overlooks the correlation among operations, which may introduce significant kernel launching and data transfer overheads. Since the operations exhibit both internal and external data dependencies, we adapt and fuse the kernels to address the concern. This allows us to store data in registers and SMEM, reducing IO latency and data access by reusing stored data. To avoid over-fusing, which can lead to excessive resource consumption and decreased SM occupancy and performance, we finely tune the methods.

Fusing in Rejection Loop. The computation of the rejection loop is illustrated in Fig. 7. In QWarp, $\hat{\boldsymbol{c}}$ is computed and stored in registers, while in SWarp it is stored in GMEM to improve occupancy. The next computation stage begins only when the ℓ_∞-norm of the currently evaluated element is within the preset threshold. We apply the early evaluation technique [24] and ensure compatibility between the ℓ_∞-norm checking and arithmetic operations. Consequently, in each iteration, we perform computation polynomial-wise and immediately check the coefficients as they are generated. This technique enables a more timely rejection.

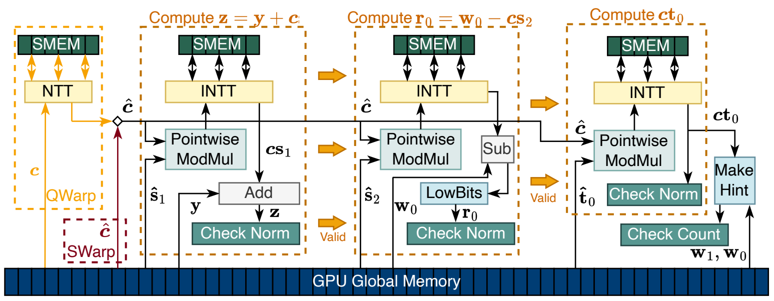


Fig. 7: Computational flow in the rejection loop of Sign.

Fusing in SWarp implementation. In SWarp approach, we fuse the Gen, the entire rejection loop in Sign, and the Verify processes into separate kernels. The maximum SM occupancy here is only 33%, so fusing high-consumption kernels, such as SHAKE, with others has little impact on overall occupancy but retains intermediate values in registers and SMEM. Combined with early evaluation [24], this approach significantly reduces overall memory requests, especially time-consuming GMEM access.

Fusing in QWarp implementation. Here, we aim to improve the occupancy of arithmetic operations. Given the high consumption of SHAKE, fusing it with arithmetic kernels might negatively impact the efficiency. Thus, in the Sign procedure, we implement the rejection loop as four kernels, responsible for sampling $\mathbf{y}$, computing the commitment $\mathbf{w}$ and decomposing, sampling $\boldsymbol{c}$, and finally computing and validating signatures, respectively. The primary consideration is to separate SHAKE from arithmetic operations, ensuring the hash does not significantly reduce occupancy.

5.3 Dynamic Task Scheduling

The mathematical expectation of repetitions in the Sign of Dilithium is around 4, but in some worse cases it may reach dozens of times. When batching multiple Sign instances, some blocks become idle after returning valid signatures, but computation does not finish until the instance with the most repetitions is complete. This results in wasted GPU hardware resources and low utilization.

To address this, we propose a dynamic scheduling strategy that is general and applicable to this abort framework. The main idea is to predicate κ, and use execution units that become idle in the next round to compute this prediction. This allows to obtain results of the same seed but with different κ. The final output is the valid signature with the smallest κ. However, it causes a mismatch between the original tasks sequence of instances and its execution sequence on the GPU. To ensure a single low-latency device-to-host data transfer of results, we must maintain the same stored order in the memory pool as in the original sequence. Thus, we construct several maps for this process and corresponding look-up tables (LUT). In our setting, tasks in signing are kept in a queue, and we batch $\varPhi$ tasks, delegating them to $\varPsi$ concurrent processing units on the device.

The first map bridges signing tasks to GPU execution units through a task LUT of size $\varPhi$ and an execution LUT of size $\varPsi$. The second map tracks output validity from execution units using a state LUT of size $\varPsi$, recording states and nonces. The device updates both LUTs each round, and typically no units are idle when execution LUT is full. When remaining tasks cannot fill execution LUT completely, we predicate κ. Another issue is the competition of the execution units, where multiple units output valid signatures but with different κ. In this case, we use another map to links the signatures with smaller κ to the correct position in the memory pool, for writing from temporal to original pool.

Figure 8 illustrates the computational flow and data structure. Data transfers between the host and device are asynchronous. Our prediction and scheduling strategy significantly reduces the number of execution rounds required. In our implementation, one block computes tasks in one instance. Therefore, we recommend

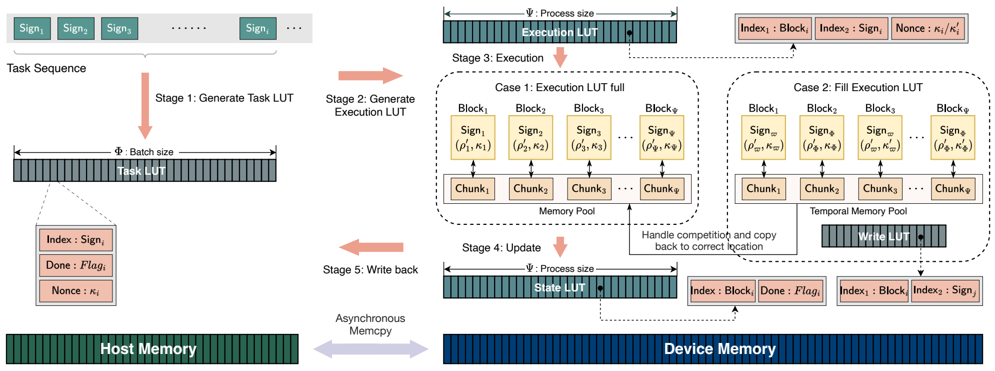

Sign_1
Sign_2
Sign_3
Sign_i
Task Sequence
Stage 1: Generate Task LUT
Φ : Batch size
Task LUT
Index : Sign_i
Done : $Flag_i$
Nonce : κ_i
Stage 2: Generate Execution LUT
Stage 5: Write back
Asynchronous Memcpy
Host Memory
Ψ : Process size
Execution LUT
Index_1 : Block_i
Index_2 : Sign_i
Nonce : κ_i/κ_i'
Stage 3: Execution
Case 1: Execution LUT full
Block_1
Block_2
Block_3
Block_Ψ
Sign_1 (ρ_1', κ_1)
Sign_2 (ρ_2', κ_2)
Sign_3 (ρ_3', κ_3)
Sign_Ψ $(\rho_\Psi', \kappa_\Psi)$
Chunk_1
Chunk_2
Chunk_3
Chunk_Ψ
Memory Pool
Case 2: Fill Execution LUT
Sign_ϖ $(\rho_\varpi', \kappa_\varpi)$
Sign_Φ $(\rho_\Phi', \kappa_\Phi)$
Sign_ϖ $(\rho_\varpi', \kappa_\varpi')$
Sign_Φ $(\rho_\Phi', \kappa_\Phi')$
Temporal Memory Pool
Write LUT
Handle competition and copy back to correct location
Stage 4: Update
Ψ : Process size
State LUT
Index : Block_i
Done : $Flag_i$
Index_1 : Block_i
Index_2 : Sign_j
Device Memory

Fig. 8: The structure of the task scheduler. The left side represents the CPU (host), and the right side is the GPU (device).

setting Ψ, the number of concurrent processing units, to match the maximum active blocks which can be determined through kernel profiling.

5.4 Asynchronization and Streaming

Asynchronization enhances the overall performance of both sides by allowing concurrent task execution without waiting for the completion of previous tasks. This approach effectively hides data transfer latency, leading to better throughput and reduced idle time. By employing asynchronization and efficient data transfer management, we can maximize resource utilization and performance.

To fully utilize the computational resources, we launch multiple CUDA streams, where each executes a partition of the entire task set with the task scheduling technique above. To enable CUDA streams, we asynchronously manage data transfers between the CPU and GPU. However, the scheduling requires synchronization between the CPU and GPU, as the CPU acts as a scheduler to allocate tasks. To alleviate the impact of synchronization, we employ multiple threads on the CPU side. Each CPU thread is assigned a unique CUDA stream and executes a partition of the tasks. This approach not only ensures that the CPU and GPU can work concurrently, but also diminishes data transfer latency, leading to improved performance and resource utilization.

5.5 Versatility of Proposed Techniques

Our proposed techniques have a high degree of versatility, and the applicability extends well beyond a singular use case. The multiple warp design is also suitable for accelerating other PQC schemes. Many GPU crypto libraries [18, 29] use 4 or 8 warps per task to improve occupancy. The implemented operations, such as NTT and rejection sampling, are common in many PQC schemes and can be adapted with minimal parameter changes. Our techniques for optimizing memory access and balancing pipelines are general and applicable in various scenarios. Additionally, the Fiat-Shamir with Aborts framework, which is widely applied, can benefit from our task scheduling mechanism.

6 Experimental Results

6.1 Experimental Setup

We compile the C/C++ code using g++ 12.2.0 and the GPU implementations with CUDA 11.8 on an Arch Linux system with kernel 5.15. Our implementation is deployed and tested on three GPUs: a NVIDIA Tesla A100 80G PCIe, a NVIDIA GeForce RTX 4090, and a NVIDIA GeForce RTX 3090 Ti. The performance of the

Table 2: Performance and profiling results for operations in Dilithium2 on a 3090 Ti GPU. The throughput reports the achieved percentage of utilization with respect to the theoretical maximum.

Operation	Type	Time (μs)	Occupancy (%)		Throughput (%)		Memory Usage (B)	
			Theoretical	Achieved	Compute	Memory	Registers	SMEM
ExpandA	(SWarp, 4)	4,805.54	75	67.13	98.33	98.33	52	3,488
ExpandMask	(SWarp, 1)	1,205.25	33.33	32.08	98.17	98.17	52	752
SampleInBall	(SWarp, 1)	532.45	33.33	32.00	97.33	97.33	54	1,288
NTT	(SWarp, 1)	22.53	33.33	29.93	37.50	89.80	38	1,536
	(QWarp, 1)	20.48	100	90.38	91.22	91.22	20	2,560
INTT	(SWarp, 1)	15.36	33.33	28.57	58.21	61.04	33	1,536
	(QWarp, 1)	20.48	100	90.62	90.71	90.71	20	2,560
Inner-product	(SWarp, 1)	856.99	33.33	31.77	27.98	76.17	80	1,152
	(QWarp, 1)	346.18	75	74.10	68.58	80.37	56	2,560
Rejection loop	(SWarp, 1)	470.21	33.33	30.41	41.93	74.48	54	3,540
	(QWarp, 1)	299.04	83.33	77.58	81.93	81.93	48	2,688

CPU baseline is obtained on an Intel(R) XEON W7-2495X CPU with 24 cores. The throughput is measured in operations per second (OP/s). The reported performance results are the medians of 100 executions, where each execution batches 10,000 tasks. The latency of host-device data transfer is also included.

6.2 Performance

Below, we evaluate the performance of our implementation by profiling all operations for both SWarp and QWarp.

Kernel Profiling Results. As an illustrative example, we present profiling results for our Dilithium2 implementation in Table 2, comparing SWarp and QWarp methodologies across various arithmetic operations. Our optimized solution almost reaches theoretical maximum occupancy, showing our methods effectively enhance performance, throughput, and occupancy. For instance, in ExpandA, the optimized hash implementation enables more warps within a block. Consequently, we can sample four polynomials simultaneously with four warps, increasing resource utilization. When examining the arithmetic operations, QWarp implementations consistently outperform SWarp across several metrics. It achieves higher occupancy levels and more efficient GPU resource utilization. Additionally, QWarp delivers more balanced throughput, ensuring a more uniform performance across different operations. Notably, the increased SMEM usage in QWarp is a trade-off for avoiding bank conflicts through additional padding, which does not affect overall occupancy as it stays within hardware limitations.

Performance on Different GPUs. Table 3 lists the performance of our implementation and the comparisons. The results for the C and AVX2 implementations are obtained by running the official implementation[5] on our platform. For the closed-source work [25], we use their reported results obtained on a Jetson AGX Xavier GPU. Notably, in [25], the authors replaced the time-consuming rejection sampling with a simple data loading based on known positions, which may lead to application incompatibilities. The works [11] and [31] represent state-of-the-art Neon-based implementation and FPGA design of Dilithium. For our GPU implementation, we report the performance using 10 streams, where each stream executes 1,000 tasks. In the scheduling, the concurrent processing size is 2,512. Compared to the CPU baseline, our implementation achieves 82×-93× improvement for Gen, 166×-181× improvement for Sign, and 88×-109× improvement for Verify on the A100 GPU. Furthermore, our implementation demonstrates over 40× improvement compared to the AVX2 implementation across all security levels on 4090 GPU.

[5] `https://github.com/pq-crystals/dilithium`

Table 3: Throughput of C and AVX2 implementations on CPU, our implementations on three different GPUs, and related works on GPU [25], ARM Cortex-A72 [11], and FPGA [31]. The metric is the operations per second (OP/s). The speedups refer to the comparison with CPU reference implementation.

Level		CPU		Our work			Related Works		
		Ref	AVX2	A100	4090	3090 Ti	[25]	[11]	[31]
2	Gen	20,552	59,132	1,859,231 (**91**×)	2,704,605 (**132**×)	1,418,283 (**69**×)	84,993	5,561	23,217
	Sign	4,624	20,260	765,855 (**166**×)	984,803 (**213**×)	562,534 (**122**×)	33,965	2,310	3,448
	Verify	18,942	57,027	2,057,013 (**109**×)	2,873,936 (**152**×)	1,484,420 (**78**×)	67,738	5,498	21,904
3	Gen	11,007	33,806	1,018,960 (**93**×)	1,492,832 (**136**×)	769,191 (**70**×)	51,099	2,908	16,555
	Sign	2,845	12,664	513,468 (**181**×)	649,498 (**228**×)	372,873 (**131**×)	14,875	1,377	2,167
	Verify	11,886	34,596	1,207,752 (**102**×)	1,683,357 (**142**×)	865,119 (**73**×)	44,502	3,352	15,671
5	Gen	7,617	21,768	624,070 (**82**×)	888,971 (**117**×)	463,949 (**61**×)	31,800	1,916	11,051
	Sign	2,394	10,305	396,894 (**166**×)	488,006 (**204**×)	271,835 (**114**×)	20,396	1,044	1,977
	Verify	7,340	21,736	643,829 (**88**×)	964,374 (**131**×)	488,732 (**67**×)	27,511	1,961	10,716

6.3 Effectiveness of Proposed Optimizations

We evaluate the impact of our optimization techniques in NTT and compare our work with open-source alternatives, using Dilithium2 as an example to show the effectiveness.

SHAKE. Table 4 presents the profiling results of computing $\mu := \mathcal{H}(tr|M)$ using the implementation from [23] and our optimized SHAKE256 implementation. Our implementation shows a 5.8% increase in compute and memory throughput compared to [23], resulting in a 14.1% reduction in execution time. Moreover, our implementation significantly reduces memory consumption, requiring 37.2% fewer registers, leading to a 125% increase in theoretical occupancy and a 118.0% increase in achieved occupancy. It also demonstrates substantial reductions in the number of instructions and L1 cache requests. GMEM instructions are reduced by 34.2%, and LMEM usage is completely eliminated. L1 cache requests for loads and stores decrease by 60.6% and 97.7%, respectively. Similarly, L2 cache interactions with GMEM and L1 cache are reduced by 87.35% and 94.81%, respectively. These optimizations significantly enhance the overall performance and resource utilization of our SHAKE256 implementation.

Speedup Breakdown of Memory Optimizations. We use the computation of $\hat{\mathbf{t}}_0$ in the signing procedure as a representative example to demonstrate the impact of our optimizations. This process involves unpacking the secret key to obtain $\mathbf{t}_0$ and then computing $\mathsf{NTT}(\mathbf{t}_0)$. We present the original implementation alongside our step-by-step optimizations, and Fig. 9 illustrates the throughput results, execution time, and total global memory access for various optimization techniques. In the basic implementation, we launch an unpacking kernel, store $\mathbf{t}_0$ in GMEM, and subsequently launch another kernel for NTT. We then apply three successive optimizations. The first optimization entails kernel fusion without altering the memory access pattern, leading to an 18.9% reduction in execution time. The second optimization involves merging loops in the unpacking and NTT processes, using registers to store intermediate values. This results in a 1.4× improvement in compute throughput, a 47.2% decrease in execution time, and a 45.4% reduction in GMEM access. Finally, the third optimization addresses bank conflict resolution, producing a kernel with balanced compute and memory throughput and enhancing execution time by 2.7× compared to the basic implementation.

Table 4: Profiling results of our SHAKE256 implementation on a 3090 Ti GPU, compared with [23]. Pipe utilization refers to the utilization of peak instructions executed.

Improved Performance							
	Throughput (%)		Execution	Occupancy (%)		Pipe Utilization (%)	
	Compute	Memory	Time (μs)	Theoretical	Achieved	LSU	ALU
[23]	90.03	90.03	121.98	33.33	30.43	91.58	28.68
Ours	95.25	95.25	104.80	75	66.34	97.65	31.70
	(**+5.8%**)	(**+5.8%**)	(**−14.1%**)	(**+125%**)	(**+118.01%**)	(**+6.63%**)	(**+10.55%**)

Reduced Resource Usage							
	Register	Memory Instructions		L1/TEX Cache Requests		L2 Cache (KB)	
	Numbers	Global	Local	Loads	Stores	L1/TEX	Global
[23]	78	410 K	650 K	660 K	430 K	49645.82	24853.76
Ours	49	270 K	0	260 K	10 K	2578.82	3142.63
	(**−37.18%**)	(**−34.15%**)	(**−100.00%**)	(**−60.61%**)	(**−97.67%**)	(**−94.81%**)	(**−87.35%**)

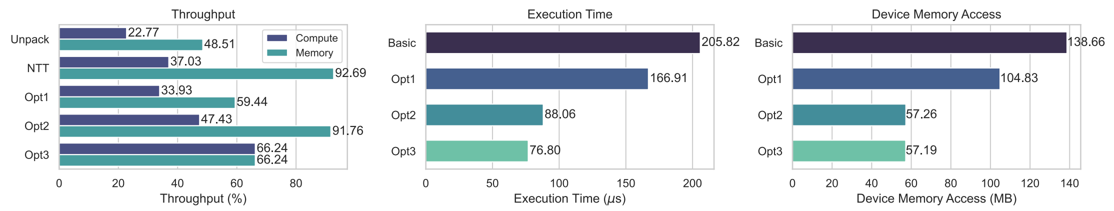


Fig. 9: Comparisons of throughput, execution time, and device memory usage between the basic implementation and the step-by-step application of the three optimizations.

Table 5: Performance comparisons with [25] on A100.

Operations	NTT			Rejection Sampling		
Work	[25]	Ours		[25]		Ours
		SWarp	QWarp	LUT	Load	
Time (μs)	1,596.42	28.67	19.45	154,254.33	40.96	442.37

Comparisons with Related Work. Table 5 compares our implementation with [25]. Since [25] is closed-source and only presents the overall performance of their Dilithium implementation on an AGX Xavier GPU, we implement their documented methods for NTT and rejection sampling to obtain comparative performance on the same platform. Their NTT implementation does not exploit different types of memory, resulting in lower access speeds and higher execution times. For rejection sampling, [25] divided the process into preparing a LUT and loading data, treating the first phase as a pre-computation. While this approach improves signing performance by eliminating sampling, it is only suitable when the matrix $\hat{\mathbf{A}}$ remains unchanged, leading to potential incompatibilities in general scenarios.

6.4 Sensitivity Study

We conduct a sensitivity study of our implementation under various execution settings, using Dilithium2 as a representative example. Three parameters are involved: the number of batched tasks on the host side (Φ), the number of concurrent processing tasks on the device side (Ψ), and the number of launched CUDA streams. First, we examine the performance of Sign over processing sizes Ψ ranging from 1000 to 10000, and detail the throughput results for three batch sizes (1000, 5000, and 10000) in Figure 10. The results show a periodic performance pattern, with a rise followed by a sustained decline within each period. As Ψ increases,

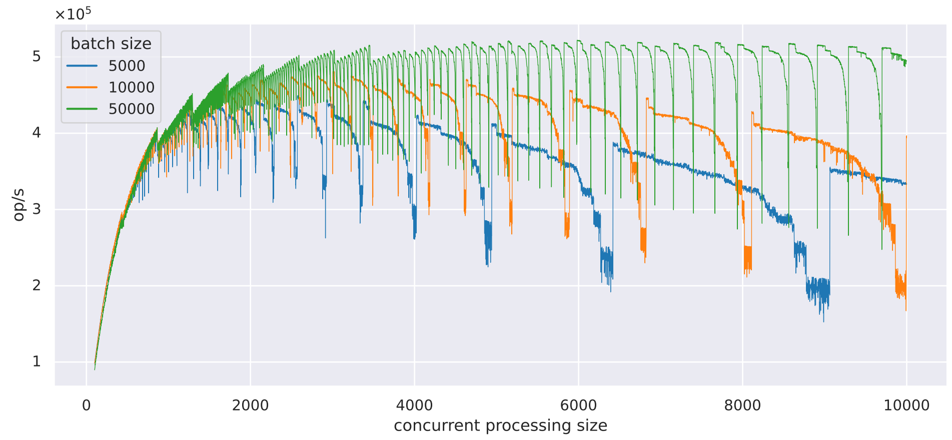


Fig. 10: Sensitivity to the concurrent processing sizes (Ψ) under three batch sizes (Φ).

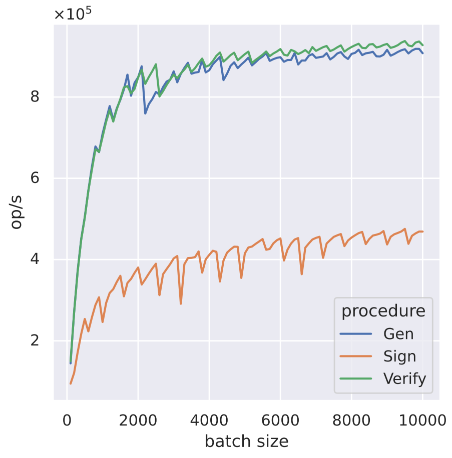
the overall throughput initially improves before declining. For all three batch sizes, the optimal point is reached at around 2000-3000, e.g., for $\Psi = 10000$ the optimal processing size is $\Psi = 2512$. Notably, since the randomness of the scheme affects the number of rounds, we can only obtain an optimal interval. Next, we fix Ψ to 2512 for Sign and examine the throughput of all three procedures under different Φ. We vary the number of launched streams from 1 to 16 to test the effectiveness of hiding IO latency. Figures 11a and 11b demonstrate that as the batch size increases, the throughput first rises rapidly and then begins to plateau at around 6000. The throughput of the three procedures mostly reaches optimal levels when launching around 10 to 12 streams.

7 Conclusion

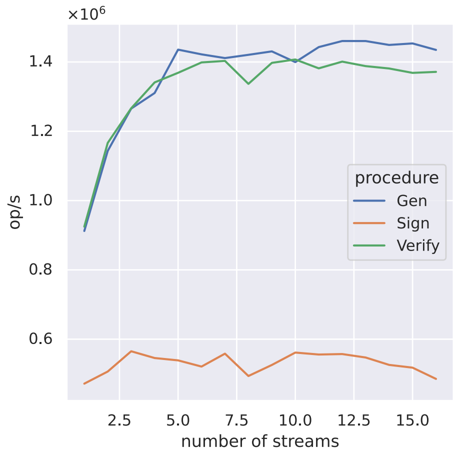
In this work, we present a high-throughput GPU implementation of Dilithium that significantly enhances performance. Through computational and memory optimizations, we have mitigated performance bottlenecks, memory issues, and IO latency. Our results demonstrate the GPUs capacity to boost lattice-based cryptography, offering insights for future work. The experimental outcomes underscore our implementations capability to deliver real-time, high-throughput solutions, advancing the development and integration of post-quantum cryptography.

References

1. Ariel Core (release 0.22.1). `https://github.com/ArielCoinOrg/arielcoin` (Nov 2022), ArielCoin
2. liboqs (release 0.7.2). `https://github.com/open-quantum-safe/liboqs` (Aug 2022), Open Quantum Safe (OQS) project
3. NVIDIA cuBLAS. `https://github.com/NVIDIA/CUDALibrarySamples` (Jun 2024)
4. Abdulrahman, A., Hwang, V., Kannwischer, M.J., Sprenkels, A.: Faster Kyber and Dilithium on the Cortex-M4. In: Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13269, pp. 853–871. Springer (2022)
5. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R.: Status report on the third round of the NIST post-quantum cryptography standardization process. US Department of Commerce, NIST (2022)
6. Alkim, E., Bilgin, Y.A., Cenk, M., Gérard, F.: Cortex-M4 optimizations for {R, M}LWE schemes. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(3), 336–357 (2020)



(a)



(b)

Fig. 11: Sensitivity to the (a) batch sizes and (b) number of streams under fixed concurrent processing sizes.

7. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - A new hope. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 327–343. USENIX Association (2016)
8. Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé: CRYSTALS-Dilithium: Algorithm specifications and supporting documentation. Submission to the NIST's post-quantum cryptography standardization process (2020)
9. Banerjee, U., Ukyab, T.S., Chandrakasan, A.P.: Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(4), 17–61 (2019)
10. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Advances in Cryptology - CRYPTO 1986. Lecture Notes in Computer Science, vol. 263, pp. 311–323 (1986)
11. Becker, H., Hwang, V., Kannwischer, M.J., Yang, B., Yang, S.: Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(1), 221–244 (2022)
12. Beckwith, L., Nguyen, D.T., Gaj, K.: High-performance hardware implementation of lattice-based digital signatures. IACR Cryptol. ePrint Arch. p. 217 (2022)
13. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium: A lattice-based digital signature scheme. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 238–268 (2018)
14. Dworkin, M.J., et al.: SHA-3 standard: Permutation-based hash and extendable-output functions (2015)
15. Gao, Y., Xu, J., Wang, H.: cuNH: Efficient GPU implementations of post-quantum KEM NewHope. IEEE Trans. Parallel Distributed Syst. **33**(3), 551–568 (2022)
16. Greconici, D.O.C., Kannwischer, M.J., Sprenkels, A.: Compact Dilithium implementations on Cortex-M3 and Cortex-M4. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(1), 1–24 (2021)
17. Gupta, N., Jati, A., Chauhan, A.K., Chattopadhyay, A.: PQC acceleration using GPUs: FrodoKEM, NewHope, and Kyber. IEEE Trans. Parallel Distributed Syst. **32**(3), 575–586 (2021)
18. Jung, W., Kim, S., Ahn, J.H., Cheon, J.H., Lee, Y.: Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 114–148 (2021)
19. Lee, W., Seo, H., Zhang, Z., Hwang, S.O.: TensorCrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU. IEEE Access **10**, 20616–20632 (2022)
20. Lyubashevsky, V.: Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In: Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5912, pp. 598–616. Springer (2009)

21. Lyubashevsky, V.: Lattice signatures without trapdoors. In: Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7237, pp. 738–755. Springer (2012)
22. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of computation **44**(170), 519–521 (1985)
23. Ono, T., Bian, S., Sato, T.: Automatic parallelism tuning for Module Learning with Errors based post-quantum key exchanges on GPUs. In: IEEE International Symposium on Circuits and Systems, ISCAS 2021, Daegu, South Korea, May 22-28, 2021. pp. 1–5. IEEE (2021)
24. Ravi, P., Gupta, S.S., Chattopadhyay, A., Bhasin, S.: Improving speed of Dilithium's signing procedure. In: Smart Card Research and Advanced Applications. pp. 57–73. Lecture Notes in Computer Science (2020)
25. Seo, S.C., An, S.: Parallel implementation of CRYSTALS-Dilithium for effective signing and verification in autonomous driving environment. ICT Express **9**(1), 100–105 (2023)
26. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997)
27. Sun, S., Zhang, R., Ma, H.: Efficient parallelism of post-quantum signature scheme SPHINCS. IEEE Trans. Parallel Distributed Syst. **31**(11), 2542–2555 (2020)
28. Wan, L., Zheng, F., Fan, G., Wei, R., Gao, L., Wang, Y., Lin, J., Dong, J.: A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator. In: Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13556, pp. 514–534. Springer (2022)
29. Yang, H., Shen, S., Dai, W., Zhou, L., Liu, Z., Zhao, Y.: Phantom: A cuda-accelerated word-wise homomorphic encryption library. IEEE Transactions on Dependable and Secure Computing pp. 1–12 (2024)
30. Zhang, F., Zhai, J., Shen, X., Mutlu, O., Du, X.: Poclib: A high-performance framework for enabling near orthogonal processing on compression. IEEE Trans. Parallel Distributed Syst. **33**(2), 459–475 (2022)
31. Zhao, C., Zhang, N., Wang, H., Yang, B., Zhu, W., Li, Z., Zhu, M., Yin, S., Wei, S., Liu, L.: A compact and high-performance hardware architecture for CRYSTALS-Dilithium. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(1), 270–295 (2022)