HI-Kyber: A novel high-performance implementation scheme of Kyber based on GPU

Xinyi Ji*, Jiankuo Dong*, Pinchang Zhang*, Tonggui Deng*, Jiafeng Hua†, and Fu Xiao*
*School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China
†Xidian University, Xi'an, China

Abstract—**CRYSTALS-Kyber, as the only public key encryption (PKE) algorithm selected by the National Institute of Standards and Technology (NIST) in the third round, is considered one of the most promising post-quantum cryptography (PQC) schemes. Lattice-based cryptography uses complex discrete alogarithm problems on lattices to build secure encryption and decryption systems to resist attacks from quantum computing. Performance is an important bottleneck affecting the promotion of post quantum cryptography. In this paper, we present a High-performance Implementation of Kyber (named HI-Kyber) on the NVIDIA GPUs, which can increase the key-exchange performance of Kyber to the million-level. Firstly, we propose a lattice-based PQC implementation architecture based on kernel fusion, which can avoid redundant global-memory access operations. Secondly, We optimize and implement the core operations of CRYSTALS-Kyber, including Number Theoretic Transform (NTT), inverse NTT (INTT), pointwise multiplication, etc. Especially for the calculation bottleneck NTT operation, three novel methods are proposed to explore extreme performance: the sliced layer merging (SLM), the sliced depth-first search (SDFS-NTT) and the entire depth-first search (EDFS-NTT), which achieve a speedup of 7.5%, 28.5%, and 41.6% compared to the native implementation. Thirdly, we conduct comprehensive performance experiments with different parallel dimensions based on the above optimization. Finally, our key exchange performance reaches 1,664 kops/s. Specifically, based on the same platform, our HI-Kyber is 3.52× that of the GPU implementation based on the same instruction set and 1.78× that of the state-of-the-art one based on AI-accelerated tensor core.**

Index Terms—**PQC, Kyber, NTT, Polynomial Multiplication, GPU**

I. Introduction

Public key cryptography is an important fundamental component of secure communication. However, quantum computing based on Shor [1] and Gorver [2] will break currently widely used public key cryptography (such as Rivest Shamir Adleman (RSA) and elliptic curve cryptography (ECC)) in polynomial time in the future. Many research institutions and enterprises (such as Google, IBM [3], etc.) have made significant progress in the field of quantum computing. Therefore, in response to this challenge, NIST has initiated a process to solicit quantum-safe cryptographic algorithms worldwide [4], which can replace public-key cryptographic algorithms. In 2022, NIST announced the Round 3 results of Post-Quantum Cryptography Standardization Process, including four selected algorithms (CRYSTALS-Kyber [5], CRYSTALS-DILITHIUM, Falcon, SPHINCS+) and four candidate algorithms (BIKE [6], Classic McEliece [7], HQC [8], SIKE). Among the selected algorithms, CRYSTALS-KYBER is the only public key encryption and key-establishment algorithm.

Corresponding author: Fu Xiao.

A. Related works

As the only PKE scheme selected in Round 3 of the NIST PQC Standardization Process, the performance of Kyber undoubtedly receive significant attention in future research. To expedite cryptographic algorithms, researchers have applied various accelerators and parallel instructions to enhance prominent computational workloads. Botros et al. [9] presented an optimized software implementation of the module-lattice-based key-encapsulation mechanism Kyber for the ARM Cortex-M4 microcontroller. On the same embedded platform, Alkim et al. [10] and Abdulrahman et al. [11] completed the further optimization implementation of Kyber. Becker et al. [12] and Nguyen et al. [13] completed software implementations using Advanced Single-Instruction Multiple-Data (SIMD) vector instructions (a.k.a. NEON instructions) based on ARM chips with more abundant resources. And with Intel similar SIMD architecture (AVX2), Hwang et al. [14] verified NTT Multiplications for Kyber and other NIST PQC KEM Lattice Finalists.

Compared to traditional CPUs (Intel Core series, etc.), GPUs have more logical computing units and cores, providing significant performance advantages. In 2006, NVIDIA introduced a general-purpose computing platform and programming model named CUDA (Compute Unified Device Architecture) [15]. By leveraging the parallel computing engine in NVIDIA GPUs, CUDA enables the efficient solution of complex computational problems. Plenty of works [16]–[20] have achieved notable results in studying cryptography parallel acceleration on the GPU. Gao et al. [18] applied GPU to the elliptic curve cryptography (ECC) algorithm. And their throughput performance has exceeded ten million per second, which has made a great breakthrough in performance compared with the CPU implementation. As for GPU-based Kyber implementation, Gupta et al. [21] implemented the optimization of subfunctions of Kyber-1024 in NVIDIA Tesla V100 of Volta architecture, especially the number-theoretic transform and Keccak, which are the two most time-consuming algorithms. Lee et al. [19] focused on the implementation of three different fine-grained parallelism approaches for NTT on NVIDIA RTX2060. Wan et al. [20] used the Tensor Core to accelerate polynomial multiplication and obtain a better performance improvement.

B. Our Contribution

Based on the same CPU platform (Intel Core), the performance of Kyber [22] has been found to be within the same order of magnitude (in tens of thousands per second) as traditional elliptic curve cryptography (e.g., Curve25519 [23]). However, on the GPU platform, the optimal implementation [21] throughput based on universal fixed-point computing capability is 473 kops/s, while the state-of-art one [20] based on AI accelerated computing capability (Tensor Core) is 820 kops/s. These implementations exhibit a significant gap compared to the optimal elliptic curve performance (7,200 kops/s) [18]. Therefore, there is still considerable potential for performance improvements in GPU-based implementations of Kyber. The main innovation points are as follows:

- We design a novel and complete High-performance Implementation scheme of CRYSTALS-Kyber based on GPU (HI-Kyber), including complete public-key encryption (key generation, encryption, and decryption). Compared to [21] which also relies on the integer computing power of GPU, we propose a more efficient kernel fusion scheme adapted to Kyber, including the underlying functions such as NTT, INTT, SHA3, Encode, Decode, etc. It greatly reduces the number of global memory accesses during computation and improves the overall performance of the algorithm.
- We optimize the core operations of CRYSTALS-Kyber, including NTT, INTT, pointwise multiplication, etc. Especially for the calculation bottleneck NTT operation, we propose three novel schemes that are universal on other platforms: SLM, SDFS-NTT and EDFS-NTT, which achieve a speedup of 7.5%, 28.5%, and 41.6% respectively compared to the native NTT implementation. For pointwise multiplication, our approach involves leveraging two theoretical knowledge to minimize the overhead of multiplication with modular reduction, resulting in better lazy reduction. We also compare two modular reduction suites as the underlying implementation baseline for better optimization.
- We conduct comprehensive performance experiments with different parallel dimensions based on the above optimization. By testing extensive comparative experiments on GPUs, we aim to find the optimal parallel parameters and achieve optimal peak performance. Finally, our key exchange performance of HI-Kyber reaches 1,664 kops/s. Notably, based on the same platform, our HI-Kyber is 3.52× that of Gupta et al. [21] implementation based on the same instruction set and 1.78× that of Wan et al. [20] implementation based on AI-accelerated tensor core.

The rest of the paper is organized as follows. Section 2 introduces some basic preparatory knowledge. Section 3 presents our specific optimization implementation scheme on GPU. Section 4 shows the experimental results. Section 5 concludes.

II. Preliminary knowledge

This section introduces the preparatory knowledge related to our work. Firstly, we briefly introduce the GPU architecture. Then, we describe the lattice-related difficult problem. Finally, the relevant theoretical concepts of modular reduction and NTT are given.

A. GPU Architecture

GPUs have thousands of cores. Usually, dozens of cores form a group, and the group corresponds to the block, which has its own shared memory. Within a block, many threads can be launched. Each thread corresponds to a core. In the hardware, there are registers that can only be used by blocks. Threads in the same block will share registers. So the more threads are launched, the fewer registers can be used, which constrains the scale of parallelism. By CUDA programming, the kernel function can call GPU resources. Due to the parallelism principle in hardware design, the smallest execution unit in SM is called a warp. All threads in a warp execute the same instruction, which maximizes GPU efficiency when the instructions executed by all threads remain consistent.

B. Lattices and Related difficult problem

$\mathbf{Lattice}$. A lattice is the entirety of a vector consisting of all integer linear combinations of a set of linearly independent vectors. If the vectors in the lattice are all integer vectors, such a lattice is said to be an integer lattice. Any lattice can be composed by a base $\mathbf{V} = \{\mathbf{v_1}, \mathbf{v_2}, \ldots\ldots, \mathbf{v_n}\}$, as

$$\mathcal{L} = \{a_1\mathbf{v_1} + a_2\mathbf{v_2} + \ldots\ldots + a_n\mathbf{v_n} \mid a_i \in \mathbb{Z}\}$$

$\mathbf{Module-learning\ with\ errors(MLWE)}$ [24]. The assumption states that for the given m samples where $\mathbf{A}$ is a uniformly-random matrix in $\mathbb{Z}_q^{k\times k\times m}$, q is the modulus, $\mathbf{s}$ is a uniformly-random vector in $\mathbb{Z}_q^k$ and the coefficients of $\mathbf{e}$ are chosen from the distribution $\mathcal{X}$, it is hard to distinguish whether all of them are from $\mathbf{A}$ or $\mathbf{As}+\mathbf{e}$.

Kyber uses MLWE as the foundation for its key encapsulation mechanism, where a shared secret key is generated by both the sender and receiver using public and private keys, and the private key is shared securely. This shared secret key can then be utilized for symmetric encryption and decryption.

C. Key Encapsulation Mechanism of Kyber

Kyber is an IND-CCA2-secure KEM whose security is based on the hardness of learning with errors over modules on a lattice. The Kyber family consists of three different algorithms: Kyber512, Kyber768, and Kyber1024, which differ in their security levels of NIST Security Levels 1, 3, and 5. Their security levels correspond to AES128, AES192, and AES256, respectively. Kyber uses parameter k to fix the lattice dimension as a multiple of n. It is set to 256 so that each bit of the message can be encoded into a single coefficient of the polynomial. Changing k is the main mechanism to scale security.

The definition of key generation, encryption, and decryption of the Kyber.CPAPKE public-key encryption scheme are described in Algorithm 1, 2 and 3. In CPAPKE.KeyGen, ρ and σ are random variables generated by a random number d through the hash function $SHA3\_512$. The matrix $\hat{\mathbf{A}}$ is generated in NTT domain to compute with parameters $\hat{\mathbf{s}}$ and $\hat{\mathbf{e}}$ which are sampling from a binomial distribution. The public and private key pairs require encoding. In CPAPKE.Enc, after decoding pk, parameters $\mathbf{r}$, $\mathbf{e_1}$ and e_2 are sampling from two centered binomial distribution. $\mathbf{u}$ and v form the ciphertext c. In CPAPKE.Dec, by decoding the ciphertext c and sk to recovery message m.

Algorithm 1: Kyber.CPAPKE.KeyGen()

```
Output:
    Secret key $sk$, Public key $pk$.
1: $d \leftarrow RandomBytes()$
2: $(\rho, \sigma) := SHA3\_512(d)$
3: $\hat{\mathbf{A}} \in R_q^{k*k}$ in NTT domain $\leftarrow Gen\_matrix(\rho)$
4: $(\mathbf{s}, \mathbf{e}) \in R_q^k \leftarrow CBD\_Sample_{\eta_1}(\sigma)$
5: $(\hat{\mathbf{s}}, \hat{\mathbf{e}}) \leftarrow (NTT(\mathbf{s}), NTT(\mathbf{e}))$
6: $\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
7: $(pk, sk) := (Encode(\hat{\mathbf{t}}||\rho), Encode(\hat{\mathbf{s}}))$
8: return $(pk, sk)$
```

Algorithm 2: Kyber.CPAPKE.Enc()

```
Input:
    Random coins $r$, Public key $pk$, Message $m$.
Output:
    Ciphertext $c$.
1: $(\hat{\mathbf{t}}, \rho) \leftarrow Decode(pk)$
2: $\hat{\mathbf{A}}^T \in R_q^{k*k}$ in NTT domain $\leftarrow Gen\_matrix(\rho)$
3: $\mathbf{r} \in R_q^k \leftarrow CBD\_Sample_{\eta_1}(r)$
4: $\mathbf{e_1} \in R_q^k \leftarrow CBD\_Sample_{\eta_2}(r)$
5: $e_2 \in R_q \leftarrow CBD\_Sample_{\eta_2}(r)$
6: $\hat{\mathbf{r}} := NTT(\mathbf{r})$
7: $\mathbf{u} := NTT^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
8: $v := NTT^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + Decompress(m)$
9: $(c_1, c_2) := (Encode(\mathbf{u}), Encode(v))$
10: return $(c_1||c_2)$
```

Algorithm 3: Kyber.CPAPKE.Dec()

```
Input:
    Ciphertext $c$, Secret key $sk$.
Output:
    Message $m$.
1: $(\mathbf{u}, v) \leftarrow Decode(c)$
2: $\hat{\mathbf{s}} \leftarrow Decode(sk)$
3: $m := Compress(v - NTT^{-1}(\hat{\mathbf{s}} \circ NTT(\mathbf{u})))$
4: return $m$
```

D. Modular Reduction

It is time-consuming for a computer to perform a native module operation. Modular reduction helps prevent arithmetic overflow and improves the efficiency of mathematical computations by reducing the size of intermediate values. In our paper, we not only implement the native modular reduction (Barrett reduction and Montgomery reduction) in Kyber but apply the recent arithmetic techniques (Improved Plantard multiplication and Improved Plantard reduction), which are more suitable for word size moduli.

Barrett Reduction : It was proposed by P. D. Barrett [25] in 1986. The required division is converted into multiplication by multiplying by the inverse of the modulus q, which reduces the integer f to h, $0 \leq h < q$. The input of Barrett Reduction is a 16-bit signed integer $f \in [-2^{15}q, 2^{15}q)$, and the output of Barrett Reduction is a 16-bit signed integer $h \in [0, q]$.

$$h = f \mod q = f - \lfloor fq^{-1} \rfloor q = f - \lfloor f/2^k \rfloor \lfloor 2^k/q \rfloor q$$

Here $\lfloor \ \rfloor$ refers to the largest integer that is not greater than f, where $\lfloor 2^k/q \rfloor$ can be computed in advance, and $\lfloor f/2^k \rfloor$ is just a shift. k is 26 in Kyber.

Montgomery Reduction : Montgomery reduction [26] is one of the most used reduction algorithms and very efficient for large modulus. It needs to transfer the data f from the normal domain to the MONT domain. It converts the required division by adding k times the modulus q to the data on the MONT field. The input of Montgomery Reduction is a 32-bit signed integer $f \in [-2^{15}q, 2^{15}q)$, and the output of Montgomery Reduction is a 16-bit signed integer $h \in (-q, q)$.

$$k = -q^{-1} \mod R$$
$$h = [f \ + \ (fk \mod R)q] \ / \ R$$

Where k is a constant, R is a power of 2. Note that constant can be precomputed and both modulo R and division by R can be implemented with shifts.

Improved Plantard Multiplication : Thomas Plantard [27] came up with an efficient word size modular arithmetic to fit the small moduli of PQC, which only supported unsigned input range. Shortly after, improved Plantard reduction [28] is proposed to back up signed input of the existing lattice-based cryptography. The $\mod^{\pm} q$ maps integers to $(-\frac{q}{2}, \frac{q}{2})$, and $\gg$ represents the right shift operation. The input of Improved Plantard Multiplication is a 16-bit signed integer f, $g \in [-2^3q, 2^3q]$, and the output of Improved Plantard Multiplication is a 16-bit signed integer $h \in (-\frac{q}{2}, \frac{q}{2})$.

$$\begin{aligned} h &= fg(-2^{-2l}) \mod^{\pm} q \\ &= ((fgq^{'} \mod 2l) \gg l + 2^{\alpha})q \gg l \end{aligned}$$

Where $q^{'} = q^{-1} \mod^{\pm} 2^{2l}$ is precomputed, l = 16, and α = 3. When g is a constant, we can precompute $g(-2^{-2l})$ to save one multiplication.

Improved Plantard Reduction : When the number in the normal domain needs to be shortened, improved Plantard reduction can reduce the normal number by multiplying a constant in the Plantard domain. It ensures that the number remains in the normal domain after Plantard reduction. The input of Improved Plantard Reduction is a 32-bit signed

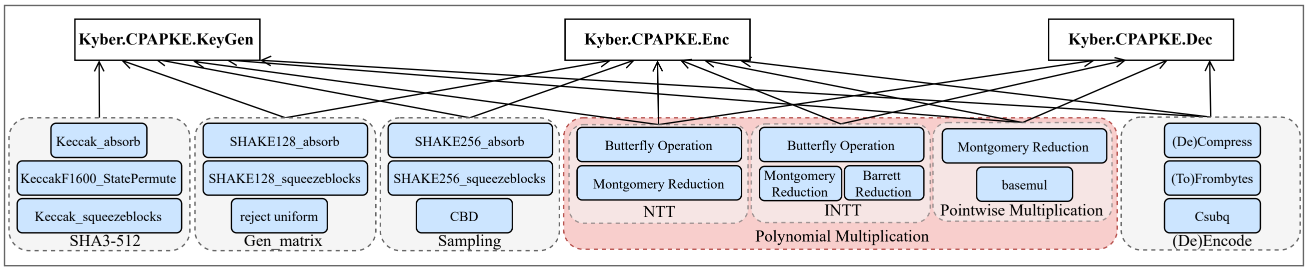


Fig. 1. The overall algorithmic framework of Kyber

integer $f \in [-2^6 q^2, 2^6 q^2]$, and the output of Improved Plantard Reduction is a 16-bit signed integer $h \in (-\frac{q}{2}, \frac{q}{2})$.

$$\begin{aligned} h &= f(-2^{-2l}) \mod{}^{\pm} q \\ &= ((fq' \mod 2l) \gg l + 2^{\alpha})q \gg l \end{aligned}$$

Where $q' = q^{-1} \mod {}^{\pm}2^{2l}$ is precomputed, l = 16, and α = 3.

E. NTT

NTT [29] is the transformation of a sequence of n elements $\mathbf{v} = [v_0, v_1, \ldots\ldots, v_{n-1}]$ into another sequence $\mathbf{V} = [V_0, V_1, \ldots\ldots, V_{n-1}]$ using the formula:

$$\mathbf{V_i} = \sum_{j=0}^{n-1} v_j \cdot \zeta^{ij}$$

where ζ denotes a primitive n-th root of unity. The powers of ζ are called twiddle factors, which need to satisfy that $\zeta^n \equiv 1 \bmod q$ does not exist $\zeta^i \equiv 1 \bmod q$ $(0 \le i < n)$. The twiddle factor is a complex exponential that depends on the position of the element in the sequence. INTT can utilize a similar formula to restore the original n sequences.

$$v_j = n^{-1} \sum_{i=0}^{n-1} \mathbf{V_i} \cdot (\zeta^{ij})^{-1}$$

Considering that polynomials have coefficient representation and point value representation, Cooley and Tukey [30] introduced a butterfly transform in $O(nlogn)$ time, which is based on the idea of divide and conquer, to speed up the calculation of polynomial multiplication.

According to the Chinese Remainder Theorem (CRT) [31], R_q and $\prod_i \mathbb{Z}_q[X]/(X - \zeta^i)$ are isomorphic. Using this idea, the polynomial $X^{256} + 1$ can be written as

$$X^{256} + 1 = \prod_{i=0}^{127}(X^2 - \zeta^{2i+1}) = \prod_{i=0}^{127}(X^2 - \zeta^{2br_7(i)+1})$$

where $br_7(i)$ is the bit reversal of the unsigned 7-bit integer i. In that way, a polynomial of $f \in R_q$ is given by

$$(f \bmod X^2 - \zeta^{2br_7(0)+1}, \ldots\ldots, f \bmod X^2 - \zeta^{2br_7(127)+1})$$

Hence, the NTT of f can be further written as

$$NTT(f) = \hat{f}_0 + \hat{f}_1 X^1 + \ldots\ldots + \hat{f}_{255} X^{255}$$

with

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j} w^{(2br_7(i)+1)j}$$

$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1} w^{(2br_7(i)+1)j}$$

III. Optimized Implementation of Kyber on GPU

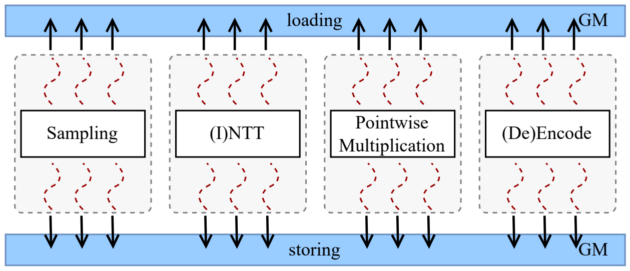
The detailed optimization strategies of Kyber on GPU are discussed in the section. Firstly, we show the overall framework of Kyber and adopt the kernel fusion implementation as a baseline. Then, the design framework for polynomial multiplication is given. Based on the idea of the framework, we introduce the optimization methods of NTT, INTT, pointwise multiplication, and lazy reduction one by one.

A. The overall framework of Kyber Implementation

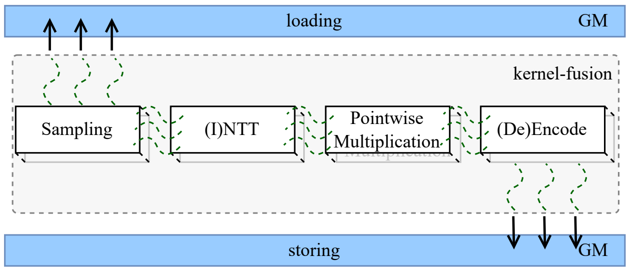
Fig. 1 shows the overall algorithm composition framework of Kyber.PKE.KeyGen, Kyber.PKE.Enc, and Kyber.PKE.Dec. When implementing Kyber on GPU, we are committed to reducing the memory access from global memory as off-chip memory has higher latency and lower access speed. In the traditional multi-kernel scheme, which is used in Gupta et al. [21] implementation, each kernel assigned with computing tasks stores the data in global memory and the data will be loaded before the next kernel executes. Such extensive access to global memory inevitably consumes a significant amount of memory-access time. To some extent, the penalty cannot be compensated by optimizing thread parallelism.

Accordingly, as shown in Fig. 2, taking the partial core algorithms in Enc as an example, we adopt the kernel fusion technique to improve the performance of parallel computations by combining multiple kernel functions into a single kernel on GPU. When multiple kernel functions are combined into a single kernel, the number of accessing global memory is reduced from 4 to 1. That means it reduces the amount of time that is spent transferring data between the CPU and GPU. And, the overhead associated with launching and managing multiple kernels is reduced.

In our work, we realize the kernel fusion of Kyber on GPU. Table I compares the additional global memory-access times caused by calling kernels on the multi-kernel scheme with the kernel-fusion scheme. It is evident that the kernel-fusion scheme has greatly reduced the number of global memory



(a) Multi-kernel scheme



(b) Kernel-fusion scheme

Fig. 2. Multi-kernel VS. Kernel-fusion

accesses triggered by calling kernels. Our work tests the kernel fusion implementation as a baseline. The throughput of PKE.KeyGen, PKE.Enc and PKE.Dec can reach 1,348 kops/s, 1,373 kops/s, and 5,064 kops/s respectively, which can achieve a speed-up of 1.25× that of Gupta et al. [21] implementation. The detailed calculation formula is described in Section IV-D.

TABLE I
ADDITIONAL GLOBAL MEMORY-ACCESS TIMES CAUSED BY DIFFERENT PARALLEL METHODS

Parallelization	KeyGen	Enc	Dec
multi-kernel	12	17	8
kernel-fusion	1	1	1

It can be seen that three parts of polynomial multiplication: NTT, INTT, and pointwise multiplication, have appeared multiple times, which is also the most time-consuming function part in Kyber. The next focus is to introduce the improvement optimization of polynomial multiplication in this article.

B. *Design for polynomial multiplication on GPU*

Fig. 3 depicts the design framework of polynomial multiplication. In the underlying implementation, we choose two modular reduction suites: one is the native implementation of Kyber and the other is the improved Plantard arithmetic (see Section II-D) which is customized for LBC schemes. Then, we present three novel algorithms to explore the optimization methods of NTT and subsequently implement INTT. Besides, two excellent theoretical methods are applied to accelerate pointwise multiplication. Details are discussed in Section II-E, III-D, and III-E respectively. By regulating these underlying algorithms, the efficiency of polynomial multiplication can be exploited to the fullest.

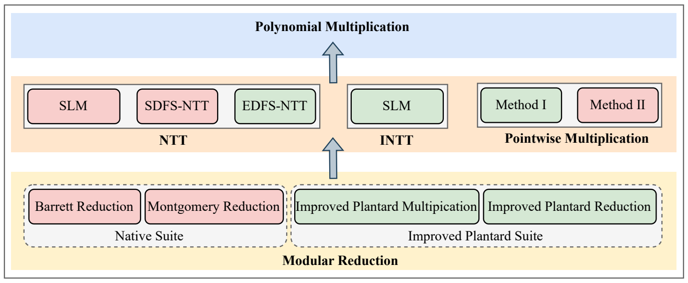


Fig. 3. Design scheme of polynomial multiplication

Before delving into the topic, it is important to understand that if loading all the temporary variables into the stack in the actual experiment, an error segmentation fault (core dumped) is encountered, which means that the memory accessed exceeds the memory space allocated to the program by the system. Hence, we must store all the data in the global memory, then load it into the register. Whether for the global memory or the cache, reducing the time of memory access is imperative. Registers are a scarce resource on GPU, and therefore, their utilization must be maximized as much as possible. In the specific experimental scheme, as the scheme designer of the GPU, it is crucial to reuse registers as much as possible, although we cannot accurately control the specific data saved by each register.

C. *NTT*

In this section, we introduce the SLM and provide pseudocode for a generic implementation for the first time. Then, we innovatively propose the concept of an NTT tree and a novel traversal mode: depth-first search. Finally, Two designed implementations of NTT are presented: SDFS-NTT and EDFS-NTT.

1) A sliced layer merging scheme: Kyber used a variant of NTT, so the total number of NTT layers is $logN-1$. For the original NTT, N NTT coefficients are loaded from global memory and then stored back in global memory after each layer of the CT butterfly. The native NTT results in a total cost of $(logN-1)\times N$ times memory loading and storing. To optimize this, we adopt the layer merging technology [32] to decompose NTT into different sub-layers, namely, $L_1+L_2+\cdots+L_i+\cdots+L_n = logN-1$. L_i means the number of sub-layer that may contain two or three NTT layers. The maximum value of n can be taken to $logN-1$, but it is meaningless as we can achieve the same result using the precompiler instruction #program unroll. In general, n is less than $\frac{logN}{2}$.

The main idea of [32] is to reuse multiple layers of NTT coefficients, thus reducing memory-access times to global memory. For $n=2$, $L_1=4$, and $L_2=3$, we only need to load and store N points before and after L_1 and L_2, instead of loading and storing $(logN-1)N$ from global memory, thus reducing the number of memory-access times. [32] used 8 registers processing 8 coefficients on Cortex-m4 while [10] processed 16 coefficients. It should be noted that the reused data must be loaded in registers to avoid being put back into global memory. The scale of coefficients that needed to be processed is a vital factor. Usually, the amount of data loaded in a loop is determined based on the number of registers on Cortex-m4 or other platforms.

In our work, we present a sliced layer merging scheme. It divides N NTT coefficients into several slices to adapt to the dynamic scheduling of registers. When L_i is ensured, the maximum number of slices that N NTT coefficients will be divided depends on the distance of butterfly units in the last layer of L_i. Fetch coefficients from global memory in batches. Fig. 4 shows that when N NTT coefficients are divided into equal length slices f_{i_j}, where j presents the j-th slice. Every slice has $\frac{N}{j}$ coefficients. The k-th coefficient of all slices i_j which can be called $Group_k$ performs the same CT butterfly units. What should be understood is that loading all N NTT coefficients sequentially into the registers (the number of slices is 1) is not an option, because it is obvious that it is impossible to load N NTT coefficients for the scarce resource registers.

In the beginning, it is no different from the original NTT, which needs to load data from global memory. However, during the computation process of the merged layer L_i, it is no longer necessary to access the global memory. Then, the grouped data can be loaded into the register to perform butterfly operations by slicing the N-dimensional vector. It is obvious that processing the grouped data in the register file directly is a smart option.

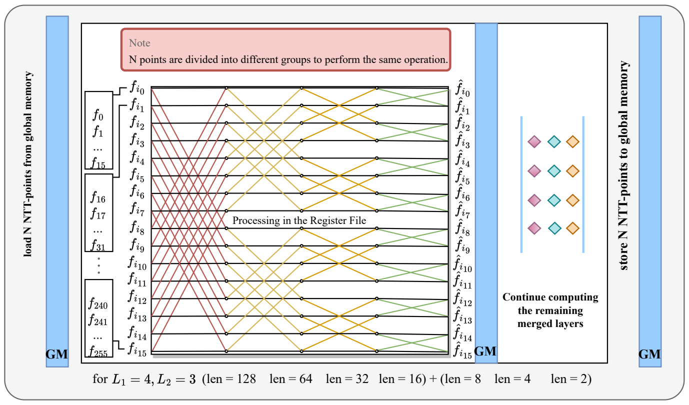

load N NTT-points from global memory
GM
Note
N points are divided into different groups to perform the same operation.
f_0 f_1 ... f_{15}
f_{16} f_{17} ... f_{31}
f_{240} f_{241} ... f_{255}
f_{i_0} … $f_{i_{15}}$
Processing in the Register File
$\hat{f}_{i_0}$ … $\hat{f}_{i_{15}}$
GM
Continue computing the remaining merged layers
store N NTT-points to global memory
GM
for $L_1 = 4, L_2 = 3$ (len = 128 len = 64 len = 32 len = 16) + (len = 8 len = 4 len = 2)

Fig. 4. A sliced layer merging scheme for NTT

When combining NTT layers $1, 2, 3$, and 4, the distance of butterfly units in the last layer L_1 is 16. Hence, the $Group_0$, $\{f_0, f_{16}, f_{32}, f_{48}, f_{64}, f_{80}, f_{96}, f_{112}, f_{128}, f_{144}, f_{160}, f_{176}, f_{192}, f_{208}, f_{224}, f_{240}\}$, a set of a N-dimensional vector which contains the first coefficient of all slices can accomplish independent butterfly operations, i.e., these 16 numbers in the merged four layers have no external dependencies. Therefore, we should iterate the butterfly operation 16 times by grouping $\frac{N}{16}$ NTT coefficients. N NTT coefficients will group the minimum pair of butterfly operations that can process completely unrelated coefficients where $slice = 2^{L_{finished}}$, $Group = len = N \gg L_{finished}$, and $L_{finished}$ presents the NTT layers which have been merged. This will save $(logN - 1 - n)N$ loading and storage.

Algorithm 4: A sliced layer merging scheme of L_i for NTT

```
input  : $f(x) \in \mathbb{Z}_q[X]/(X^n+1), \zeta^n \in \mathbb{Z}_q$
output: $\hat{f}(x) \in \mathbb{Z}_q[X]/(X^n+1)$, after $L_i$ Layer
        Merging
$L_{finished} \leftarrow \sum_{j=1}^{i-1} L_j$
$MAX\_Group \leftarrow N \gg (L_{finished} + L_i)$
for $Group \leftarrow 0$ to $MAX\_Group$ do
    $k \leftarrow \frac{N}{N \gg (L_{finished})}$
    for $len \leftarrow N \gg (L_{finished}+1)$ to
     $N \gg (L_{finished}+L_i)$ do
            /* shift right one layer */
        for $start \leftarrow Group$ to $N$ do /* step by
         $j + len$ */
            $zeta \leftarrow \zeta^{k++}$
            for $j \leftarrow start$ to $start + len$ do
             /* step by $MAX\_Group$ */
                Butterfly Unit($f[j+len], fj$)
```

In the classic layer merging implementation, most of the ARM underlying instruction sets are used to complete the loading and storage of data. Even though performance is improved, the code is very verbose, which increases the compilation burden. We present the pseudocode of merging the L_i-th layer in Algorithm 4. In essence, it is actually a method of performing more fine-grained grouped divide-and-conquer inside the layer merging NTT.

Taking a deeper consideration, if there are too few slices, the NTT coefficients to be processed within a group may exceed the number of available registers. This situation will influence performance because it requires coefficients to be stored temporarily in local or global memory and then fetched again. Therefore, in general, we usually choose the maximum number of slices to process the minimum pair of butterfly operations. The details will be discussed in Section IV-B.

2) NTT Tree: As we know, NTT adopts the idea of recursive divide-and-conquer, dividing the original problem into several similar and smaller subproblems. The original problem is solved by the solutions of the subproblems. Similarly, NTT of the N-dimensional vector can be seen as a full binary tree with a depth of $logN - 1$. A CT butterfly with the same twiddle factor is a node, then, NTT can be considered a breadth-first search (BFS) for a full binary tree. Based on the order of traversing the root node, Left-subtree, and Right-subtree, the subtrees solve the same subproblem of each layer, finally merging the solution of the subproblems to get the result of the original problem. The BFS means traversing all N NTT coefficients, whereas there are not enough registers to manipulate these coefficients. In this paper, we propose an innovative approach that uses the depth-first search of NTT. Each element in a node is accessed in the order of preorder traversal (perform the CT butterfly in the node).

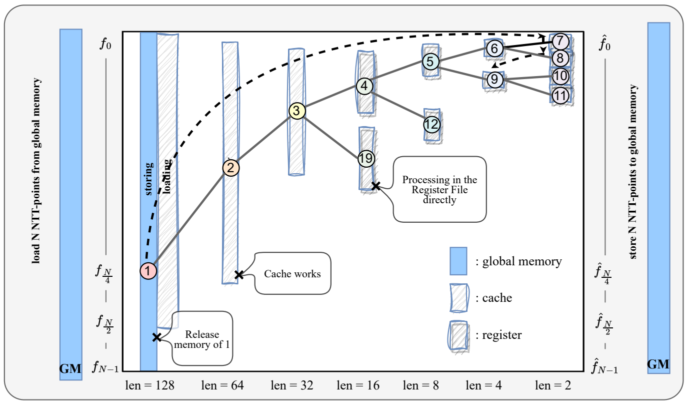


Fig. 5. A depth-first search implementation scheme for NTT

Fig. 5 reveals the DFS implementation scheme for NTT. We have simplified the NTT coefficients of the same twiddle factor into a node. Due to the space limitations of the drawing board, only a part of the depth-first traversal process is shown. Starting from the root node, follow the direction of the left subtree longitudinally until the leaf node is found. Then go back to the previous node and traverse the right subtree until all reachable nodes are traversed. That is, the NTT tree of the depth-first search will first complete all butterflies of the first layer $[f_0, f_1, \cdots, f_{N-1}]_{len=128}$. When the traversal of the root node is finished, we need to go to the left node of the root node and locate the ζ of the root node in the Left-subtree, namely, the butterfly of the second layer $[f_0, f_1, \cdots, f_{\frac{N}{2}-1}]_{len=64}$ will be performed. In the longitudinal traversal of the Left-subtree, until the butterfly of the last layer $[f_0, f_1, f_2, f_3]_{len=2}$ is completed, the Left-subtree traversal stops. Then backtrace the node, and locate the ζ of the father node. We can go to the right node in the Right-subtree, and calculate the butterfly of $[f_4, f_5, f_6, f_7]_{len=2}$. By constantly backtracking until the full NTT Tree is traversed, the DFS is executed.

In DFS, the number of coefficients loaded in a node is always half of the last node. However, due to the scarcity of registers, the memory resources of node 1 will be released when the root node finishes the CT butterfly. At this point, the coefficients loaded to node 2 may come from the global memory or from the cache (depending on where the value will be stored after node 1 finishes the computation). If there are insufficient registers, the return value of the node is more possible to be stored in the cache. For example, when computing node 19, the coefficients which are left from its father node 3 may well be loaded from the cache.

Based on the idea of layer merging and depth-first search, taking memory-access-friendly and register-friendly as the starting point, we explore two novel schemes of traversing NTT: SDFS-NTT and EDFS-NTT.

A sliced DFS scheme : In this paper, we design a sliced depth-first search scheme for NTT based on SLM. SLM reduces the number of memory accesses, and on the basis of data slicing, the N-dimensional vectors are divided into disjoint slices to maximize the utilization of registers. When the last layer of L_i is fixed, the maximum number of slices is ensured, and the minimum is 2. Each slice has $\frac{N}{len(slice)}$ coefficients which can be organized into a group to do CT butterfly.

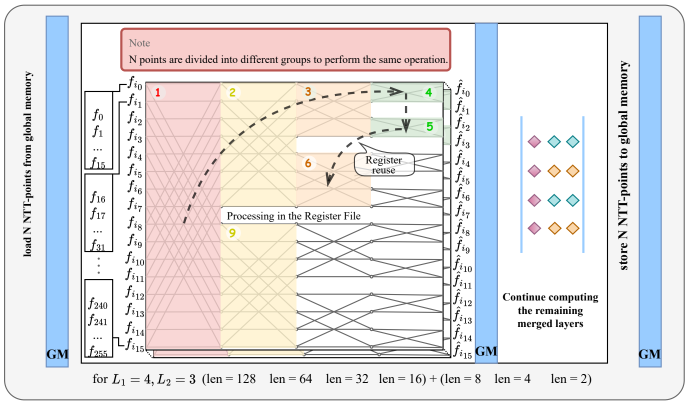

load N NTT-points from global memory
Note
N points are divided into different groups to perform the same operation.
Register reuse
Processing in the Register File
Continue computing the remaining merged layers
store N NTT-points to global memory
GM
for $L_1 = 4, L_2 = 3$ (len = 128 len = 64 len = 32 len = 16) + (len = 8 len = 4 len = 2)

Fig. 6. A sliced depth-first search scheme for NTT

Unlike Fig. 4, the butterfly operations for each group in Fig. 6 are not executed sequentially, but in the order of preorder traversal. Step 1 performs all the butterfly units in each group. Then, Step 2 performs half of the previous step, and so on. When L_1 equals 4, the N coefficients are split into sixteen slices, the procedure is as follows: Firstly, we split N-dimensional vectors into sixteen slices where the size of the slice depends on the distance of the butterfly in the last layer of L_i. The size of the first merged layers L_1 is 4, and the len of the last layer is 16. So the slice equals 2^{L_1}, and the number of groups is $N \gg L_1$. Then, the k-th ($0 \le k \le len(slice) - 1$) coefficient of each slice performs the first butterfly units. Followed by the first butterfly units, the k-th ($0 \le k \le \frac{len(slice)}{2} - 1$) coefficient performs the second butterfly units. Besides, as the depth increases, the scale continues to halve. Until visiting the whole tree in preorder traversal, sub-NTT completes all computations. Repeat this process sixteen times to complete the first four layers of data conversion.

A detailed observation reveals that in the SDFS-NTT, the combination of SLM and DFS not only reduces the number of accesses to the global memory but also greatly increases the reuse of registers. When calculating Step 6, coefficients would be loaded from the cache which is left from Step 2. And, when the grouped data in the register file reach the leaf node in batches, computational resources are released. With idle registers in advance, the compiler will schedule other available work. This is an ideal situation when you have a suitable amount of data. In cases where registers are overloaded or redundant, we have alternative solutions. On the one hand, when a register is overloaded, it is the compiler's job to load the remaining unloadable coefficients into the cache for the next access. In our experiment, we choose the maximum number of slices, namely, the most coefficients that can be grouped. On the other hand, when the register is redundant, we conduct experiments to determine the appropriate amount of loaded data to saturate the register. However, as mentioned

```
Algorithm 5: SDFS-NTT scheme
input : $f(x) \in \mathbb{Z}_q[X]/(X^n+1), \zeta^n \in \mathbb{Z}_q$
output: $\hat{f}(x) \in \mathbb{Z}_q[X]/(X^n+1)$, after $L_i$ Layer
        Merging
 $L_{finished} \leftarrow \sum_{j=1}^{i-1} L_j$
 $MAX\_Group \leftarrow N \gg (L_{finished}+L_i)$
for $Group \leftarrow 0$ to $MAX\_Group$ do
    $a \leftarrow Group$    /* the location where
     the butterfly begins */
    $Leafnode\_num \leftarrow 0$   /* the number of
     leaf nodes traversed */
    $k \leftarrow \frac{N}{N \gg (L_{finished}+L_i-2)}$
         /* locate the $\zeta$ of the father
    node */
    $len \leftarrow N \gg (L_{finished}+L_i-1)$   /* visit
     the root node */
    while $Leafnode\_num \neq \frac{N}{N \gg (L_{finished}+L_i-2)}$
     do
                  /* Jump nodes in layer
    $(L_{finished}+L_i)$ are not fully
    traversed */
        if $len \neq N \gg (L_{finished}+L_i+1)$ then
         /* Non-empty node */
            $zeta \leftarrow \zeta^k$
            for $start \leftarrow a$ to $a+\frac{len}{2}$ do  /* step
             by $j+len$ */
                for $j \leftarrow start$ to $start+len$ do
                 /* step by $MAX\_Group$ */
                    Butterfly Unit($f[j+len], fj]$)
            $len \leftarrow len \ll 2$ /* go to the left
             node */
            $k \leftarrow k \gg 2$
        else
            $len \leftarrow len \ll 2$  /* backtrace the
             node */
            $k \leftarrow k \gg 2$
            $zeta \leftarrow \zeta^{k+1}$  /* go to the right
             node in Right-subtree */
            $a = j + len$
               /* butterfly begins in the
            leaf node */
            for $j \leftarrow a$ to $a+\frac{len}{2}$ do  /* step by
             $MAX\_Group$ */
                Butterfly Unit($f[j+len], fj]$)
            $a = j + len$
               /* butterfly begins in the
            backtracing node */
            $k = \frac{k+2}{pow(2, jump_i)}$   /* locate the $\zeta$
             of the backtracing node */
            $len = pow(2, jump_i + 1)$
             /* backtrace the node */
            $Leafnode\_num++$    /* add the
             number of visited node */
```

```
Algorithm 6: EDFS-NTT scheme
input : $f(x) \in \mathbb{Z}_q[X]/(X^n+1), \zeta^n \in \mathbb{Z}_q$
output: $\hat{f}(x) \in \mathbb{Z}_q[X]/(X^n+1)$, after $L_i$ Layer
        Merging
 $L_{finished} \leftarrow \sum_{j=1}^{i-1} L_j$
 $Leafnode\_num \leftarrow 0$     /* the number of
 leaf nodes traversed */
 $a \leftarrow 0$       /* the location where the
 butterfly begins */
while $Leafnode\_num \neq \frac{N}{N \gg (L_{finished}+L_i-2)}$ do
          /* Leaf nodes are not fully
traversed */
    if $len \neq N \gg (L_{finished}+L_i+1)$ then
     /* Non-empty node */
        $zeta \leftarrow \zeta^k$
        for $start \leftarrow a$ to $a+\frac{len}{2}$ do  /* step by
         $j+len$ */
            for $j \leftarrow start$ to $start+len$ do
                Butterfly Unit($f[j+len], fj]$)
        $len \leftarrow len \ll 2$    /* go to the left
         node */
        $k \leftarrow k \gg 2$
    else
        $len \leftarrow len \ll 2$     /* backtrace the
         node */
        $k \leftarrow k \gg 2$ $zeta \leftarrow \zeta^{k+1}$  /* go to the
         right node in Right-subtree */
        $a = j + len$
          /* CT butterfly begins in the
        leaf node */
        for $j \leftarrow a$ to $a+len$ do
            Butterfly Unit($f[j+len], fj]$)
        $a = j + len$
          /* CT butterfly begins in the
        next node */
        $k = \frac{k+2}{pow(2, jump_i)}$  /* locate the $\zeta$ of
         the backtracing node */
        $len = pow(2, jump_i + 1)$
        $Leafnode\_num++$       /* add the
         number of visited node */
```

before, we cannot accurately control the specific data saved by each register. We can only reuse registers as much as possible, and reduce the offset of memory access addresses.

We show the pseudocode of the SDFS-NTT with layer merging L_i in Algorithm 5. Frequent recursive will consume a lot of stack space, so we opt to use an iterative method to expand the search. The maximum number of slices is chosen as the termination condition for the number of each group. When the N coefficients are split into an abundance of slices, the grouped data is much smaller than the number of registers, which may lead to register redundancy. We will discuss the situation in Section IV-C.

An entire DFS scheme : Specifically, when the number of slices is 1 (uncut original N NTT coefficients), we present another thought: the entire depth-first search scheme for NTT. Unlike the SLM, it is just dividing a bid loop into two or three small loops, which can not change the truth that the register resources are so scarce that can not load all N coefficients and access global memory frequently.

However, different from the batch mode, the entire DFS in NTT will load all of the N coefficients at first. Even though the register resource is not enough for the native N coefficients, it is no doubt that when the halved coefficients reach a certain point, these coefficients can be processed directly in the register. Another beneficial point is that the probability of cache-hit will be greatly raised, as the amount of loaded coefficients that need to be loaded is ranging in $N \dashrightarrow \frac{N}{2} \dashrightarrow \frac{N}{4} \cdots$. This is the choice we can guide the compiler to make.

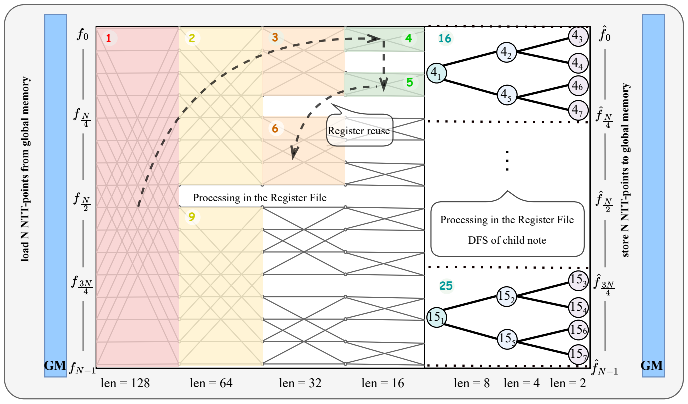


Fig. 7. An entire depth-first search scheme for NTT

The process of the EDFS-NTT is shown in Fig. 7, where the first four layers are merged. The trait of DFS of the entire NTT will load all of the N coefficients in the first layer, which is different from the batch mode. When the DFS reaches the leaf node in L_i, the register can be reused immediately owing to the enormous data scale. Moreover, regardless of the number of sub-layer, the depth-first search of the entire tree always accesses consecutive memory locations. It is friendly for threads to reduce the memory-access time whether from global memory or cache. For this reason, when processing the later sub-layer, the coefficients of the subtree must be a consecutive segment of N coefficients. Such as the node 16 which is the left subtree of node 4, has only 16 intersecting coefficients in the next three layers. That means the coefficients are loaded from consecutive memory locations, and mathematical operations of the latter three layers can be performed directly in registers without putting them back.

At the same time, another important merit of layer merging in EDFS-NTT is reducing the depth of the NTT tree, thereby allowing early reaching of the leaf node. The frequent reuse of registers ensures the efficient utilization of the GPU's computational resources, resulting in a high acceleration ratio.

Algorithm 6 shows the pseudocode of the entire DFS scheme for NTT when layer merging L_i.

D. INTT

What is different from the CT butterfly is that the distance of the GS butterfly is increased layer by layer. The distance of the butterfly is changing from ranging in [128,64,32,16,8,4,2] to [2,4,8,16,32,64,128], which can be understood as the back-tracking process of the NTT tree. Fig. 8 shows that in the backtracking process, the upper layer of the INTT depends on the lower layers. For instance, in INTT, node 4 depends on the values of the underlying node 8 and the underlying node 9, and the final node 1 depends on the values of node 2 and node 3. If we get the value of 4 by calculating the underlying nodes 8 and 9, we cannot free the memory of it in advance until node 5 is calculated. The computation of node 2 needs its left and right child nodes. Similarly, if node 3 is not calculated, the memory of node 3 cannot be released. In this case, the depth-first search cannot be used because the value of the parent node is determined by its left and right child nodes. However, this does not affect the sliced layer merging. Think in another way, the sliced layer merging essentially performs the breadth-first search, that is, sequentially calculating the GS butterfly of all coefficients layer-by-layer.

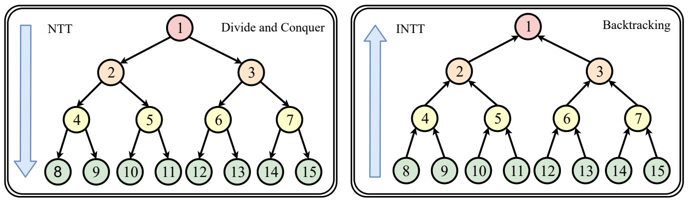


Fig. 8. NTT and INTT

E. Pointwise Multiplication

We can compute the product of two polynomial $f \cdot g$ in a short time using NTT and NTT^{-1} as $h = f \circ g$ equals to $h = NTT^{-1}(\hat{h}) = NTT^{-1}(\hat{f} \circ \hat{g}) = NTT^{-1}(NTT(f) \circ NTT(g))$. At this point, the product between polynomial vectors are transformed as $\hat{h}_{2i} + \hat{h}_{2i+1}X = (\hat{f}_{2i} + \hat{f}_{2i+1}X)(\hat{g}_{2i} + \hat{g}_{2i+1}X) \bmod X^2 - w^{2br_7(i)+1}$, with

$$\begin{aligned}\hat{h}_{2i} &= \hat{f}_{2i}\hat{g}_{2i} + \hat{f}_{2i+1}\hat{g}_{2i+1}\zeta^{2br_7(i)+1} \\ \hat{h}_{2i+1} &= \hat{f}_{2i}\hat{g}_{2i+1} + \hat{f}_{2i+1}\hat{g}_{2i}\end{aligned} \tag{1}$$

A polynomial of degree 1 requires five multiplications and five reductions. Based on the special feature between the four variables, we adopt two well-known algorithms to ameliorate equation 1. The first algorithm we use is the Karatsuba algorithm [33], which reduces the scale of the multiplication by reusing the results of the products $\hat{f}_{2i}\hat{g}_{2i}$ and $\hat{f}_{2i+1}\hat{g}_{2i+1}$ twice. It only takes four multiplications and four reductions. We call it method I. The formula is

$$\begin{aligned}\hat{h}_{2i} &= (\hat{f}_{2i}\hat{g}_{2i})_q + ((\hat{f}_{2i+1}\hat{g}_{2i+1})_q\zeta^{2br_7(i)+1})_q \\ \hat{h}_{2i+1} &= ((\hat{f}_{2i}+\hat{f}_{2i+1})(\hat{g}_{2i}+\hat{g}_{2i+1}))_q \\ &\quad - (\hat{f}_{2i}\hat{g}_{2i})_q - (\hat{f}_{2i+1}\hat{g}_{2i+1})_q\end{aligned} \tag{2}$$

Method II is the Proposition proposed by Alkim et al. [10]. If we perform one reduction followed by one multiplication, the resulting coefficients are much smaller than the range that the reduction function can handle. Therefore, adding the results of several multiplications within the processing range of the reduction function is an alternative. It can decrease two times of the reduction, as

$$\begin{aligned}\hat{h}_{2i} &= (\hat{f}_{2i}\hat{g}_{2i} + (\hat{f}_{2i+1}\hat{g}_{2i+1})_q\zeta^{2br_7(i)+1})_q \\ \hat{h}_{2i+1} &= (\hat{f}_{2i}\hat{g}_{2i+1} + \hat{f}_{2i+1}\hat{g}_{2i})_q\end{aligned} \tag{3}$$

We designed a thread-friendly program using the characteristics of the GPU to minimize the address offsets. The important intention is that we hope threads can access consecutive memory as much as possible. Fig. 9 points out the design ideas. In Method I, two 32-bit temporary variables a, b and two 16-bit temporary variables c, d are required to save the value of $\hat{f}_{2i}\hat{g}_{2i}$, $\hat{f}_{2i+1}\hat{g}_{2i+1}$, $\hat{f}_{2i}+\hat{f}_{2i+1}$, and $\hat{g}_{2i}+\hat{g}_{2i+1}$ respectively. We try to place $\hat{f}_{2i}+\hat{f}_{2i+1}$ earlier after $\hat{g}_{2i}\hat{f}_{2i}$ in order to reduce the thread access distance. Considering the different types of stored variables after the two arithmetic operations, the addresses of a and c are most likely discontinuous. So we run the two multiplications and additions consecutively with the memory contiguity of temporary variables as a priority.

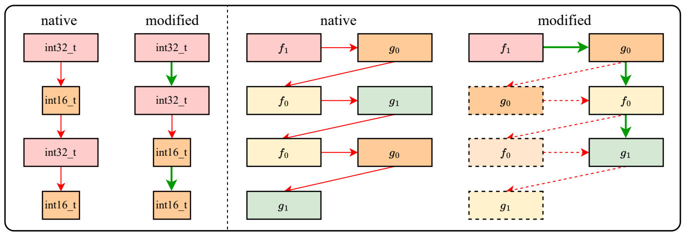


Fig. 9. Memory access optimization

Note that the Improved Plantard Arithmetic scheme uses 32-bit twiddle factors. In Method II, the output range after multiplying two variables $\hat{f}_{2i}\hat{g}_{2i}$ mod q is $(-\frac{q}{2}, \frac{q}{2})$. After multiplying with the 32-bit twiddle factor, the result needs to be stored in 64-bit, which means one extra register is needed here for storing the result. Hence, we selectively reduce the number of registers by adjusting the order of the product of variables and twiddle factors. Algorithm 7 shows the process in which only two 32-bit temporary variables are needed. First, variable f is multiplied with the twiddle factor first to perform the reduction operation and obtain the result in $(-\frac{q}{2}, \frac{q}{2})$. The variable g is multiplied to get a number resulting in 18-bit, which can be stored in the 32-bit.

In Method II, the native procedure is to compute $\hat{f}_{2i}\hat{g}_{2i+1}$ after $\hat{f}_{2i+1}\hat{g}_{2i}$. It is clear that these four variables are irrelevant.

Algorithm 7: Pointwise Multiplication with Erdem Alkim scheme

```
input : two 16-bit integer $f_0$, $f_1$ of polynomial $f$
        two 16-bit integer $g_0$, $g_1$ of polynomial $g$
output: two 16-bit integer $h_0$, $h_1$ of polynomial $h$
$tmp0 \leftarrow 0$;$tmp1 \leftarrow 0$
$h_0 \leftarrow$ Reduction module$(f_1, \zeta)$
$tmp0 \leftarrow f_1 g_0$
$tmp1 \leftarrow g_0 f_0$
$tmp0 \leftarrow f_0 g_1 + tmp0$
$tmp1 \leftarrow g_1 h_0 + tmp1$
$h_0 \leftarrow$ Reduction module$(tmp1)$
$h_1 \leftarrow$ Reduction module$(tmp0)$
```

The modified procedure is shown on the right side of Fig. 9. By adjusting the order of instructions, the modified path of threads brings reused variables closer together, which shortens the address offset between adjacent instructions.

F. Lazy Reduction

The purpose of lazy reduction is to skip unnecessary reductions without affecting correctness. In Kyber, reductions usually occur in the following three cases: firstly, after the product of two coefficients, the reduction is necessary to decrease the size of the product, such as the operation $g\zeta$ in CT butterfly. Secondly, when adding or subtracting two coefficients, the resulting double coefficient may exceed the storage type limit, such as $f + g$ in GS butterfly. Finally, reductions are applied to all coefficients, which appear at the end of the NTT and at the end of the pointwise multiplication.

It is worth noting that the range of inputs and outputs plays a crucial role in achieving better lazy reductions when using deterministic formulas. We adopt two suites of reduction functions: the native modular reduction and the improved Plantard modular reduction. A complete polynomial multiplication is divided into three parts: NTT, pointwise multiplication, and INTT. By detailed analysis, we can simplify the required number of reductions in the three parts above and compare the difference between two suites of reduction, i.e., Montgomery arithmetic and improved Plantard arithmetic.

CT Butterfly. The butterfly units in forward NTT perform a pair of $f \pm g\zeta$. Owing to the symmetry of the output range of Montgomery arithmetic and improved Plantard arithmetic, either in $(-q, q)$ or $(-\frac{q}{2}, \frac{q}{2})$, we can only discuss the upper limit of $f \pm g\zeta$. For Montgomery arithmetic, all of the N NTT coefficients grow by q after each layer and reach $q + 7q$ finally. However, a variable of $8q$ cannot enter base multiplication, which uses Montgomery multiplication in pointwise multiplication in that $8q > \sqrt{2^{15}q}$. Therefore, in the native modular reduction, all of the N NTT coefficients must undergo a one-time modular reduction after forward NTT for the purpose of entering base multiplication. For improved Plantard arithmetic, all of the N NTT coefficients grow by $\frac{q}{2}$ after each layer and reach $q + 3.5q$ finally. Thus, the coefficients can enter base multiplication directly without modular reduction due to the objective fact that the input upper range of improved Plantard multiplication is 2^3q.

Pointwise Multiplication. As is mentioned in III-E, two methods are used to reduce the number of multiplications and reductions required. The main requirement is to ensure that the limit values of the coefficients are within the admissible range of the function. In matrix-vector multiplication, all of the coefficients expand K times, which is 4. For Montgomery arithmetic, all of the N NTT coefficients are reduced to $[0, q]$. When using Method I, the odd coefficients of equation1 expand to $12q$ after matrix-vector multiplication, which exceeds the range of 16-bit signed integers. In Method II, all N NTT coefficients expand to $4q$ and none of them need to be reduced as the impact factor in the following INTT must be within 16-bit. For improved Plantard arithmetic, the odd coefficients grow to $6q$ in Method I after base multiplication, while the even coefficients are $4q$. Yet, all N NTT coefficients will be within $2q$ in Method II. Overall, except for using Method I in Montgomery arithmetic, which is not feasible, no reduction needs to be performed after pointwise multiplication.

GS Butterfly. The butterfly units in inverse NTT perform a pair of $f + g$ and $(f - g)\zeta$. That means we can ignore half of N coefficients since they are always q or $\frac{q}{2}$ in the second formula $(f - g)\zeta$. Noted that the distance of GS butterfly is doubled by the power of 2. The doubled coefficients $f + g$ are influenced by two sets. When the distance is 4, the set (f_0, f_1, f_2, f_3) will double but the extremes are not synchronized if the limit of (f_0, f_1) isn't equal to (f_2, f_3) in the first-layer butterflies. The challenge is to determine the most cost-effective modular reduction process. Even if the number of modular reductions can be cut (either for all coefficients or parts), it is not efficient to pause a loop to perform additional reductions. Therefore, we have been trying to find a balance between lazy reduction and extra overhead. For Montgomery arithmetic, the even coefficients grow to $4q$. After the first-layer butterflies, half of N coefficients grow to $8q$ and the other half are q. For $8q < 2^{15} - 1 < 10q$, one modular reduction is required to keep the coefficients in q. 3 Layers butterflies can be conducted over Barrett reduction. In the forth-layer butterflies, eight sets of coefficients $(f_i \sim f_{i+3})$, $i \in \{0, 32, 64, 96, 128, 160, 192, 224\}$ grow to $8q$. Considering the next four sets $(f_{i+4} \sim f_{i+7})$, $i \in \{0, 64, 128, 192\}$ will also grow to $8q$ in the fifth layer and two sets $(f_{i+8} \sim f_{i+15})$, $i \in \{0, 128\}$ in the sixth layer, i.e, one modular reduction of 256 coefficients is required in the forth-layer butterflies in order to continue the next third-layer butterflies without any more reductions.

For improved Plantard arithmetic, the coefficients expand to $12q$ after the first-layer butterflies by using Method I. One modular reduction is required to keep the coefficients in $\frac{q}{2}$. Within the allowed range, four layers of reductions can be skipped. Four sets of coefficients $(f_i \sim f_{i+3})$, $i \in \{0, 64, 128, 192\}$ grow to $8q$. Same as Montgomery arithmetic, the wise choice is to perform one modular reduction of 256 coefficients after the fourth-layer butterflies, instead of 16, 8, and 8 reductions in the fifth-layer, sixth-layer, and seventh-layer, respectively. In Method II, [28] discussed the condition of initial value $2q$, choosing modular reduction of 128 coefficients after the second-layer butterflies and 8 coefficients after the sixth-layer butterflies. In our case, it is not worth suspending access to consecutive memory for threads, even though the number of modular reductions can be decreased. Detailed discussions will be displayed in Section IV-C.

IV. Performance Evaluation

In our work, the performance of Kyber implementation is evaluated in the following environment. The hardware environment of the CPU is a 36-core Intel (R) Xeon (R) CPU E5-2699 v3 @ 2.30GHz. The hardware environment of the GPU is Titan V with NVIDIA Volta architecture and 5120 CUDA cores, and the memory bandwidth is 653 GB/s. The software environment is Linux Ubuntu 20.04 operating system and NVCC compiler, the compilation parameters are –std=c++11 –relocatable-device-code=true, and the CUDA version number is 11.4.

A. Evaluation Criteria

- *Throughput*: The number of requests completed by the host and the device in a unit of time, such as 1 second.
- *Latency*: In cryptographic algorithms, the time it takes for the host and the device to complete the computation from the time the request is received, respectively.
- *Threads/Block*: the number of threads contained in a CUDA block.
- *Registers/Thread*: the number of registers assigned for each thread, the maximum number of registers in a block is 65537. The *Register/Thread* equals 65537 divided by *Threads/Block*.
- *Block Number*: the number of blocks contained in a CUDA kernel.

B. Performance of NTT

The metric we are most concerned with is the throughput of Kyber implementation on GPU. The performance of native NTT on GPU reaches 48,318 kops/s. As the reference, three proposed implementations of NTT achieve 7.5%, 28.5%, and 41.6% speed-ups for SLM, SDFS-NTT, and EDFS-NTT respectively.

First, we evaluate the performance of SLM. Fig. 10a shows the effectiveness of SLM by selecting the most suitable sub-layers of 7. The default number of data loaded into a group is N divided by the maximum number of slices. When n = 2, the loss from large address offsets where the distance of CT butterfly is 128 in the first layer, is greater than the gain from reducing the memory-access times of global memory. It is obvious that owing to the smaller address offsets, merging more layers in the first stage can get better performance. Compared with the native NTT, the best performance of 2+2+2+1 can reach 51,924 kops/s.

As stated above, we set the maximum number of slices as initial inputs. The amount of coefficients within a group is the

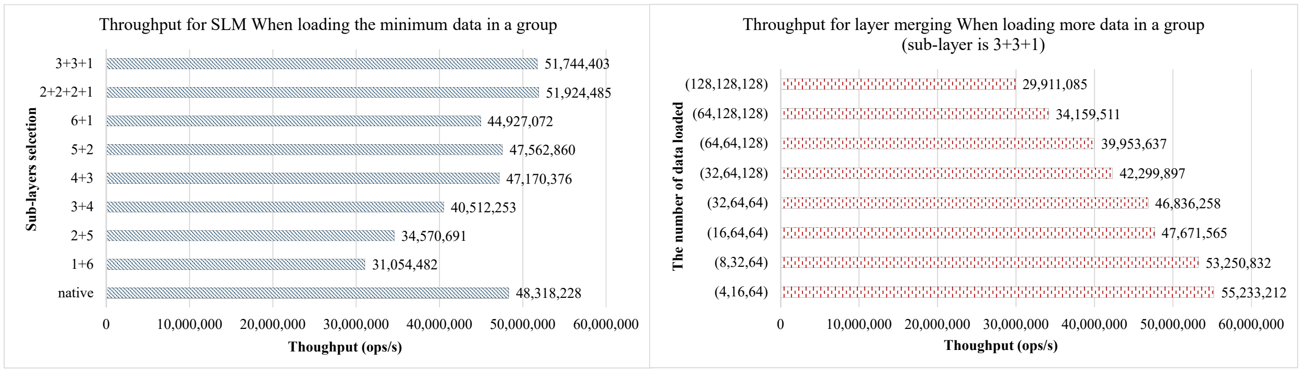


Fig. 10. Performance of SLM

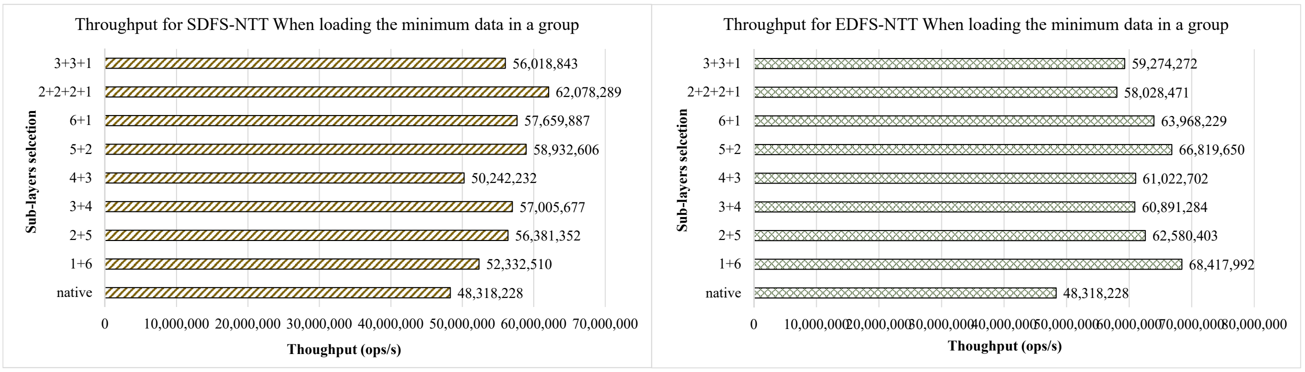


Fig. 11. Performance of SDFS-NTT and EDFS-NTT

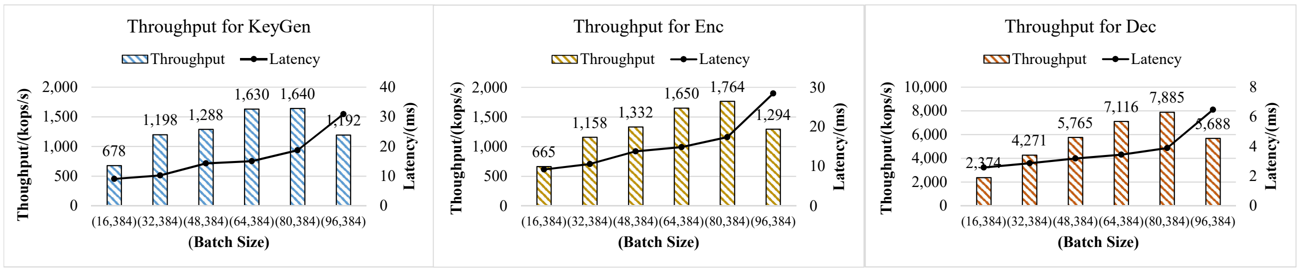


Fig. 12. Performance of Kyber implementation

minimum that can be processed in sub-layers. Take the sub-layer 3+3+1 as a test, for the first two layers, 4 coefficients within a group performing CT butterfly. For the subsequent two layers, the size increases to 16 and 64 coefficients depending on the last layer of L_i. However, we can decrease the number of slices to load more coefficients to a group. Hence, we test this condition to judge whether the registers are redundant. Fig. 10b reveals that when the number of loaded coefficients in a group increases, registers become unable to handle the overloaded data efficiently.

Significantly, both SDFS-NTT and EDFS-NTT outperform the native implementation of NTT, as shown in Fig. 11. Compared with the native NTT, all schemes achieve improvements by using the concept of the NTT tree and adjusting the traversal order into a depth-first search. For the sliced depth-first search NTT, we also test the throughput of processing more coefficients into a group and find no register redundancy. The total number of registers available per block in NVIDIA Titan V is 65536. When the thread size is fixed to 384, available registers per block are 168. After computing the first layer, the left-subtree and right-subtree of the root node will process 128 data which can be loaded in 168 registers. Benefit from access to consecutive memory locations and halved data layer by layer, ultimately, the peak performance reaches 68,418 kops/s in EDFS-NTT.

C. *Performance of Polynomial Arithmetic*

Compared with the native polynomial arithmetic, the throughput of the proposed polynomial arithmetic implementation achieves 35.8% speed-up. Except for the optimized NTT,

TABLE II
COMPARISON OF THROUGHPUT ON HI-KYBER WITH THE RELATED WORK

	Platform	KeyGen/(ops/s)	Enc/(ops/s)	Dec/(ops/s)	Perf/(KX/s)
Sanal et al. [34]	Apple A12 2-cores Vortex at 2.490 GHz and 4-cores	26,157	26,774	27,360	13
Xing et al. [35]	Xilinx Artix7	17,182	14,728	11,600	7
C-Ref [22]	Intel Core i7-4770K 3.5 GHz (Haswell), 4 Cores	11,390	10,101	8,826	5
AVX2-Ref [22]	Intel Core i7-4770K 3.5 GHz (Haswell), 4 Cores	47,591	35,962	44,232	23
Gupto et al. [21]	NVIDIA Volta V100 GV100, 5120 CUDA cores	-	-	-	473
L. Wan et al. [20]	NVIDIA GeForce RTX 3080	1,250,000	1,298,701	2,380,952	820
Our work	NVIDIA Titan V Volta GV100, 5120 CUDA cores	1,639,949	1,763,898	7,885,157	**1,358**
	NVIDIA Tesla V100 Volta GV100, 5120 CUDA cores	2,022,552	2,105,977	9,378,937	**1,664**
	NVIDIA GeForce RTX 3080	1,916,227	2,204,866	6,091,689	**1,458**

INTT using SLM has a speed-up of 2.5%. Since INTT is the process of backtracking, the distance of GS butterfly is from 2 to 128. Hence, we select three sub-layers of 1+2+2+2 and 1+3+3 because of smaller address offsets. The experiments show that the performance of Method I and Method II in pointwise multiplication has no big difference. However, after applying two schemes into NTT, pointwise multiplication, and INTT, combining lazy reduction strategy, Method I achieves a better performance of Kyber implementation. This means the overhead of multiplication is larger than modular reduction.

It should be noted that two underlying modular reduction suites have different input ranges and output ranges, where we can ignore the input range because of the limit of 16-bit signed integers. So, What plays a role in GS butterfly is the output range. In the naive suite, two modular reductions are required to execute in the first and fourth layers. When choosing sub-layers of INTT in 1+3+3, the modular reduction can be inserted in the first sub-layer. Considering the overhead of breaking down a loop to do other work, the modular reduction of the fourth layer can be inserted in the second sub-layer to maintain the original order of memory access. The experiments demonstrate that inserting in the original order of INTT in 1+3+3 outperforms breaking down the three-layer loop. According to the same principle, in improved Plantard arithmetic, we adjust the modular reduction to the second sub-layer, which is the most efficient scheme.

D. Comparison with related works

Fig. 12 shows that the throughput of Kyber increases with the launching of more batch sizes. Since NVIDIA Titan V has 80 stream processors, we test the number of blocks up to 96. Additionally, We calculate the number of key exchanges when the number of threads and registers is (384, 168) in that when we test the baseline experiment on GPU, (384, 168) always outperforms other groups. The number of key exchanges is obtained by the formula $\frac{ab}{a+b}$, where a and b refer to the throughput of KeyGen and Dec. At this point, we determine the optimal parallel parameters (80, 384, 168) for HI-Kyber. The number of key exchanges reaches $1,358$ kops per second.

Table II shows the performance comparison with the state-of-the-art implementation whether on different platforms or the same. It is obvious that our throughput of HI-Kyber is far superior to theirs. Compared with the state-of-the-art CPU implementation, our work achieves a speed-up of 34×, 49×, and 178×, respectively.

On the GPU platform, we test our work on V100 and GeForce RTX 3080 to illustrate the performance on different GPUs. Compared with state-of-the-art GPU implementations, the throughput of HI-Kyber is 3.52× and 1.78× faster than [21] and [20]. The speed-ups for Kyber mainly come from the efficient EDFS-NTT which is the first to propose on GPU and memory access optimization in the whole polynomial arithmetic. However, the parallel parameters used in V100 and GeForce RTX 3080 are the optimal chosen in NVIDIA Titan V. We believe better experimental results can be obtained on other GPUs by tuning the parallel parameters as well as the selected SDFS-NTT or EDFS-NTT.

V. CONCLUSION

In this paper, we explore three schemes for NTT implementation on GPU, SLM, SDFS-NTT, and EDFS-NTT. For these optimized schemes, We consider the valuable time of memory access and the efficient utilization of registers as starting points to explore the research direction. From the experimental results, our HI-Kyber performance obtains a new speed record compared to existing implementations. The throughput of HI-Kyber is 1.78× faster than all other works.

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[2] L. K. Grover, "Quantum computers can search rapidly by using almost any transformation," *Physical Review Letters*, vol. 80, no. 19, p. 4329, 1998.

[3] I. Company®. (2023) IBM Quantum Computing. [Online]. Available: https://www.ibm.com/quantum/

[4] NIST. (2023) Post-Quantum Cryptography Standardization. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[5] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber: a CCA-secure module-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 353–367.

[6] N. Aragon, P. S. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Guneysu, C. A. Melchor *et al.*, "Bike: bit flipping key encapsulation," 2017.

[7] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier *et al.*, "Classic McEliece: conservative code-based cryptography," *NIST submissions*, 2017.

[8] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, and I. Bourges, "Hamming quasi-cyclic (HQC)," *NIST PQC Round*, vol. 2, no. 4, p. 13, 2018.

[9] L. Botros, M. J. Kannwischer, and P. Schwabe, "Memory-efficient high-speed implementation of Kyber on Cortex-M4," in *Progress in Cryptology–AFRICACRYPT 2019: 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9–11, 2019, Proceedings 11*. Springer, 2019, pp. 209–228.

[10] E. Alkim, Y. A. Bilgin, M. Cenk, and F. Gérard, "Cortex-m4 optimizations for {R, M} LWE schemes," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 336–357, 2020.

[11] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and A. Sprenkels, "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*. Springer, 2022, pp. 853–871.

[12] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon NTT: faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1," *Cryptology ePrint Archive*, 2021.

[13] D. T. Nguyen and K. Gaj, "Optimized software implementations of CRYSTALS-Kyber, NTRU, and Saber using NEON-based special instructions of ARMv8," in *Proceedings of the NIST 3rd PQC Standardization Conference (NIST PQC 2021)*, 2021.

[14] V. Hwang, J. Liu, G. Seiler, X. Shi, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang, "Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 718–750, 2022.

[15] NVIDIA, "CUDA C programming guide 9.0," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2017.

[16] W. Pan, F. Zheng, W. Zhu, and J. Jing, "An efficient elliptic curve cryptography signature server with GPU acceleration," *IEEE Transactions on Information Forensics and Security*, 2017.

[17] J. Dong, F. Zheng, N. Emmart, J. Lin, and C. Weems, "sDPF-RSA: Utilizing floating-point computing power of GPUs for massive digital signature computations," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 599–609.

[18] L. Gao, F. Zheng, N. Emmart, J. Dong, J. Lin, and C. Weems, "DPF-ECC: Accelerating Elliptic Curve Cryptography with Floating-Point Computing Power of GPUs," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 494–504.

[19] W.-K. Lee and S. O. Hwang, "High throughput implementation of post-quantum key encapsulation and decapsulation on GPU for Internet of Things applications," *IEEE Transactions on Services Computing*, vol. 15, no. 6, pp. 3275–3288, 2021.

[20] L. Wan, F. Zheng, G. Fan, R. Wei, L. Gao, Y. Wang, J. Lin, and J. Dong, "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*. Springer, 2022, pp. 514–534.

[21] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: Frodokem, Newhope, and kyber," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 575–586, 2020.

[22] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, and D. Stehle, "CRYSTALS-Cryptographic Suite for Algebraic Lattices: Kyber," *NIST Round 3 Submissions*, 2020.

[23] O. S. Foundation, "OpenSSL Cryptography and SSL/TLS Toolkit," http://www.openssl.org/, 2016.

[24] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[25] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.

[26] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[27] T. Plantard, "Efficient word size modular arithmetic," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1506–1518, 2021.

[28] J. Huang, J. Zhang, H. Zhao, Z. Liu, R. C. Cheung, Ç. K. Koç, and D. Chen, "Improved Plantard Arithmetic for Lattice-based Cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 4, pp. 614–636, 2022.

[29] M. Bhattacharya, R. Creutzburg, and J. Astola, "Some historical notes on number theoretic transform," in *Proc. 2004 Int. TICS Workshop on Spectral Methods and Multirate Signal Processing*, vol. 2004, 2004.

[30] F. W. Vote, "Discrete Fourier Transform for 360/67 Computing System (Cooley-Tukey FFT Method)." MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB, Tech. Rep., 1972.

[31] D. Pei, A. Salomaa, and C. Ding, *Chinese remainder theorem: applications in computing, coding, cryptography*. World Scientific, 1996.

[32] E. Alkim, P. Jakubeit, and P. Schwabe, "Newhope on ARM Cortex-m," in *Security, Privacy, and Applied Cryptography Engineering: 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*. Springer, 2016, pp. 332–349.

[33] A. A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet physics. Doklady*, vol. 7, pp. 595–596, 1963.

[34] P. Sanal, E. Karagoz, H. Seo, R. Azarderakhsh, and M. Mozaffari-Kermani, "Kyber on ARM64: Compact Implementations of Kyber on 64-Bit ARM Cortex-A Processors," in *Security and Privacy in Communication Networks*, J. Garcia-Alfaro, S. Li, R. Poovendran, H. Debar, and M. Yung, Eds. Cham: Springer International Publishing, 2021, pp. 424–440.

[35] Y. Xing and S. Li, "A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, pp. 328–356, 2021.