High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber

Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj

Abstract—Post-Quantum Cryptography (PQC) has emerged as a response of the cryptographic community to the danger of attacks performed using quantum computers. All PQC schemes can be implemented in software and hardware using conventional (non-quantum) computing systems. PQC is the biggest revolution in cryptography since the invention of public-key schemes in the mid-1970s. Lattice-based key exchange schemes have emerged as leading candidates in the NIST PQC standardization process due to their relatively short public keys and ciphertexts. This paper presents novel high-speed hardware architectures for four lattice-based Key Encapsulation Mechanisms (KEMs) representing three NIST PQC finalists: NTRU (with two distinct variants, NTRU-HPS and NTRU-HRSS), CRYSTALS-Kyber, and Saber. We benchmark these candidates in terms of their performance and resource utilization in today's FPGAs. Our best architectures outperform the best designs from other groups reported to date in terms of the area-time product by factors ranging from 1.01 to 2.88, depending on the algorithm and security level. Additionally, our study demonstrates that CRYSTALS-Kyber and Saber have very similar hardware performance. Both outperform NTRU in terms of execution time by a factor 36-62 for key generation and 3-7 for decapsulation, assuming the same security level.

Index Terms—Post-Quantum Cryptography, Lattice-based, Key Encapsulation Mechanism, Hardware, FPGA, High-speed.

1 Introduction

Post-Quantum Cryptography (PQC) refers to a class of cryptographic algorithms that are resistant against all known attacks using quantum computers and can be implemented on traditional non-quantum computing platforms. These platforms include microprocessors, microcontrollers, graphics processing units (GPUs), Field Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), and many others. The main goal of PQC is to replace the existing public-key cryptography standards based on RSA and Elliptic Curve Cryptography. These standards seem to be most vulnerable to quantum computing and impossible to defend using traditional approaches such as *gradually* increasing key sizes [1], [2].

To initiate a timely transition to a new class of cryptographic schemes, in December 2016, NIST launched its PQC standardization process with the release of a call for public-key post-quantum cryptographic algorithms [3]. Sixty-nine submissions were judged complete and accepted for Round 1, and 26 of them qualified for Round 2.

On July 22, 2020, NIST announced 15 candidates qualified for Round 3 of the standardization process, including 7 finalists and 8 alternate candidates. There have been only four KEM PQC finalists. Since the NIST announcement, it has become urgent to compare them against each other. The excellent implementation of Classic McEliece was reported in 2017-2018 [4]. Thus, the efficient implementations of the remaining three KEM finalists have been of utmost importance. Consequently, in this work, we aimed at evaluating and contrasting the hardware efficiency of NTRU, CRYSTALS-Kyber, and Saber.

The preliminary results of our study were presented at the Third PQC Standardization Conference in June 2021. The extended version of this paper was posted at the Cryptology ePrint Archive in November 2021. Consequently, this work has already contributed to the comprehensive evaluation of the NIST Round 3 finalists in the category of KEMs, leading to the NIST end-of-Round-3 announcement and the corresponding report, published on July 5, 2022. In relation to this study, NIST selected only one KEM, CRYSTALS-Kyber, to be standardized in the near future. This paper serves as an archival version of our earlier informal reports. Additionally, NIST, in its report, indicated that it is still in the process of negotiating agreements with several third parties to overcome potential adoption challenges posed by third-party patents. If these agreements are not executed by the end of 2022, NIST may still consider selecting NTRU instead of Kyber. Our study clearly quantifies the disadvantages of such a decision from the point of view of the performance of the future KEM standard in hardware.

In terms of the exact algorithm types, we focus on KEMs with indistinguishability under a chosen-ciphertext attack (IND-CCA). Our primary goal was to implement all lattice-based IND-CCA secure KEMs described in the specifications of PQC finalists. The submission package of NTRU describes two substantially different KEMs: NTRU-HRSS and NTRU-HPS. As a result, we have implemented four KEMs representing three PQC finalists. For each implemented KEM, we generated results for all supported security levels.

Our Contributions. The main contributions of this paper are summarized below:

a) We have proposed, documented, and developed the first complete hardware architectures of two variants of NTRU KEM (NTRU-HRSS and NTRU-HPS), as defined in the submissions to Rounds 2 and 3 of the NIST PQC standardization process.

b) We have developed a new hardware architecture of

CRYSTALS-Kyber that achieves the best product latency × #LUTs for all security levels.
c) We have developed three new hardware architectures of Saber. For security level 3, two of them outperform the best previous design in terms of speed and one in terms of resource utilization. Our Saber implementations achieve the best product latency × #LUTs for all security levels.
d) We benchmarked all mentioned above designs using two FPGA families, Zynq UltraScale+ and Artix-7, and compared them with the fastest earlier reported FPGA implementations of CRYSTALS-Kyber and Saber.
e) To the best of our knowledge, this is the first paper that describes pure hardware Register-Transfer Level implementations (rather than software/hardware or High-Level Synthesis-based implementations) of more than one PQC candidate by members of the same team. As a result, the underlying assumptions, designer skills, and optimization effort have been very comparable for all designs. All designs reported in this paper are fully reproducible, and, for the purpose of full transparency, their source code will be released as open-source after the acceptance of this paper.

2 Previous Work

We are unaware of any complete hardware implementations of either the NTRU-HPS or NTRU-HRSS KEMs as defined during Rounds 2 and 3 of the PQC standardization process. Ref. [5] presents a hardware implementation of the public-key encryption and decryption in NTRU-HPS using the schoolbook polynomial multiplier. This implementation does not support key generation and major functions of encapsulation and decapsulation, such as ternary sampling, SHA-3, etc. Earlier versions of NTRU were significantly different. Therefore, all major building blocks, a top-level block diagram, a scheduling scheme, and the corresponding control unit had to be designed from scratch.

The most relevant hardware implementations of CRYSTALS-Kyber are described in [6] and [7].

The third pure hardware implementation, reported in [8], supports only encapsulation and decapsulation and is about an order of magnitude less efficient. An instruction-set coprocessor for Kyber was proposed in [9]. Earlier implementations of Kyber were of the software/hardware type, and many of them concerned a substantially different Round 1 version of this candidate.

The most relevant FPGA implementations of Saber are described in [10] and [11]. Both designs follow a unified architecture approach that supports selecting parameter sets at run time. The first design, [10], employs a schoolbook-based multiplier. Meanwhile, [11] proposes to use a hierarchical 8-level Karatsuba multiplier. The later paper, [12], improves area consumption of the high-speed multiplier from [10] and introduces a new lightweight architecture.

There are multiple implementations of Kyber and Saber in ASICs, including [13], [14], [15]. Developing side-channel-resistant implementations of PQC algorithms is an interesting field of research. However, it is outside of the scope of this work.

3 Background

Selected features of all implemented KEMs and their parameter sets are summarized in Tables 1 and 2, respectively. From the implementation point of view, Kyber is designed specifically with the idea of performing all multiplications using the Number-Theoretic Transform (NTT). Saber and NTRU are not. NTRU and Saber avoid using modular reduction after each partial multiplication, as all operations are performed modulo a power of 2. Kyber requires a reduction modulo a prime. NTRU uses polynomial inversion, which is typically much more complex than polynomial multiplication. Kyber and Saber do not.

3.1 NTRU

From the implementation point of view, all major operations in NTRU are polynomial operations over the quotient rings R_q, S_q and S_p where $R_q : \mathbb{Z}_q[x]/\Phi_1\Phi_n$, $S_q : \mathbb{Z}_q[x]/\Phi_n$, and $S_p : \mathbb{Z}_p[x]/\Phi_n$. Φ_1 is $(x-1)$. Φ_n is $(x^n-1)/\Phi_1 = x^{n-1} + x^{n-2} + \ldots + x + 1$. Parameter p is fixed to 3 in all parameter sets of NTRU. Thus, polynomials in S_p are in ternary form, i.e., have their coefficients in $\{-1, 0, 1\}$. In this paper, for NTRU, we use the notation S_p and S_3 interchangeably. Coefficients of polynomials in R_q and S_q have bit-widths of $\epsilon_q = log_2 q$ and those of polynomials in S_p have bit-widths of $\epsilon_p = \lceil log_2 p \rceil$.

In NTRU-HRSS, polynomial f, which is a part of the secret key is required to have non-negative correlation property, $\sum_i f_i f_{i+1} \geq 0$. Intermediate polynomial g used in key generation also has the same property. In NTRU-HPS, polynomial m in S_p has the fixed-weight property, consisting of $d/2$ coefficients equal to 1 and $d/2$ coefficients equal to -1, with $d = q/8 - 2$. Having the fixed-weight property of m ensures that the ciphertext $c \equiv 0 \pmod{(q, \Phi_1)}$ in NTRU-HPS. In NTRU-HRSS, in order to achieve the same property of c, m is lifted from S_3 to R_q by the map $m \mapsto \Phi_1 \cdot S_3(m/\Phi_1)$.

3.2 CRYSTALS-Kyber

A basic operation of CRYSTALS-Kyber is the multiplication of two polynomials. In Kyber the polynomials are elements of $R_q = \mathbb{Z}_q[x]/(x^n + 1)$.

Thus, for all security levels, polynomials are of the same degree $n = 256$, and their coefficients are members of the base prime field $\mathbb{Z}_q$, where $q = 3329$. However, a different number of polynomials is required for each security level. These polynomials are treated as a vector. The size of this vector is specified using the parameter k. k is 2, 3, and 4 for security levels 1, 3, and 5, respectively. Secret noise polynomials are sampled from a Centered Binomial Distribution (CBD), where η is either 2 or 3.

An efficient method for polynomial multiplication in R_q is through the use of the Number-Theoretic Transform (NTT) [16], which is a generalization of the Discrete Fourier Transform (DFT) to the finite ring $\mathbb{Z}_q$. In Rounds 2 and 3 of the NIST PQC standardization process, Kyber uses $n = 256$ and $q = 3329 = 13 \cdot 2^8 + 1$, where $2n \nmid (q-1)$. To make efficient NTT multiplication possible, a new definition of NTT was provided, which transforms a polynomial of degree 256 to a polynomial of degree 128 made up of degree one polynomials as its coefficients.

TABLE 1: Features of lattice-based NIST Round 3 finalists in the category of KEMs

Feature	CRYSTALS-Kyber	Saber	NTRU-HPS	NTRU-HRSS
Underlying problem	Mod-LWE: Module Learning with Errors	Mod-LWR: Module Learning with Rounding	SVP: Shortest Vector Problem	SVP: Shortest Vector Problem
Sampling	Integers are sampled from a centered binomial distribution (CBD)	Integers are sampled from a centered binomial distribution (CBD)	Fixed-weight and variable -weight polynomials are sampled from a uniform distribution	Variable-weight polynomials are sampled from a uniform distribution
Degree n	Power of 2	Power of 2	Prime	Prime
Modulus q	Prime	Power of 2	Power of 2 with $q/8 - 2 \leq 2n/3$	Power of 2 with $q > 8\sqrt{2}(n+1)$
Other major parameters	k: number of polynomials per vector, η: parameter of CBD	p, T: other moduli, l: number of polynomials per vector, μ: parameter of CBD	d: Fixed weight for g and m Lift(m): Identity $m \mapsto m$	f, g: Non-negative correlation Lift(m): $m \mapsto \Phi_1 \cdot S_3(m/\Phi_1)$
Hash-based functions	SHA3-256, SHA3-512, SHAKE128, SHAKE256	SHA3-256, SHA3-512, SHAKE128	SHA3-256, SHAKE128	SHA3-256, SHAKE128
Polynomial Rings	$\mathbb{Z}_q[x]/(x^n+1)$	$\mathbb{Z}_q[x]/(x^n+1)$	R_q: $Z_q[x]/(x^n-1)$ S_q: $Z_q[x]/(\Phi_n)^*$ S_3: $Z_3[x]/(\Phi_n)^*$	R_q: $Z_q[x]/(x^n-1)$ S_3: $Z_3[x]/(\Phi_n)^*$

* $\Phi_n = (x^n-1)/(x-1)$ irreducible in $Z_q[x]$

TABLE 2: Parameter sets of investigated algorithms.

Algorithm	Parameter Set	Security Level	Degree n	Modulus q
Kyber	Kyber512	1	256	3329
NTRU-HPS	ntruhps2048677	1*	677	2^{11}
NTRU-HRSS	ntruhrss701	1*	701	2^{13}
Saber	LightSaber-KEM	1	256	2^{13}
Kyber	Kyber768	3	256	3329
NTRU-HPS	ntruhps4096821	3*	821	2^{12}
Saber	Saber-KEM	3	256	2^{13}
Kyber	Kyber1024	5	256	3329
Saber	FireSaber-KEM	5	256	2^{13}

* assuming non-local computational models, implicitly used by submitters of other investigated algorithms.

3.3 Saber

A distinctive feature of Saber is that rounding operations are used to avoid the noise addition step and reduce the amount of randomness required. Additionally, by using only moduli that are powers of 2, modular reduction does not require any hardware resources, and rejection sampling is eliminated. Apart from q, other moduli in Saber include fixed $p = 2^{10}$ and $T = 2^3$, 2^4, and 2^6 in parameter sets corresponding to Level 1, 3, and 5, respectively.

Saber involves operations on matrices and vectors of polynomials over the quotient rings $R_q : \mathbb{Z}_q[x]/(x^n+1)$ with fixed $n = 256$. Polynomials in Saber are sampled from the uniform distribution or centered binomial distribution. β_μ denotes a centered binomial distribution with the parameter μ and the values of samples in the range $[-\mu/2; \mu/2]$. The module dimension l defines the size of vectors and matrices of polynomials as $l \times 1$ and $l \times l$, respectively. We denote $R_q^{l \times l}$ and $R_q^{l \times 1}$ as a matrix and vector of polynomials in R_q. The rounding operation includes coefficient-wise addition of a constant factor followed by bit shifting.

4 Hardware Designs

Our primary design goal is the minimum latency of three major operations – Key Generation, Encapsulation, and Decapsulation – expressed in time units. However, parallelization is pursued only until it gives a substantial reduction in latency as compared to the area increase in LUTs. For our target platforms, we chose representative devices of two different FPGA families: Zynq UltraScale+ and Artix-7. Specifically, we selected the largest devices of both families supported by free versions of Xilinx tools, Zynq UltraScale+ ZU7EV-3 and Artix-7 XC7A200T-3. Both FPGA families have been widely adopted in previous implementations, and using the same platforms enables fair comparison with state-of-the-art. For these devices, the percentage utilization of BRAMs and DSP units remains well below the percentage utilization of LUTs in all our designs. Therefore, for the purpose of design-space exploration, we use the number of LUTs as a measure of the circuit $Area$. In the Zynq devices, all computations are performed using Programmable Logic (FPGA fabric), without any contributions from the Processing System.

4.1 Choice of a Multiplier Type

Major features of investigated algorithms and their parameter sets affecting the choice of a multiplier type are summarized in Table 3. All four candidates involve multiplication of a polynomial with so-called "small" coefficients, belonging to the range listed in the second column of Table 3, by a polynomial with "large" coefficients, in the range [0..q-1], where q is given in Table 2. Out of several major polynomial multiplier types, only the Schoolbook multiplier can take full advantage of the feature that "small" coefficients have significantly fewer bits than "large" coefficients. This multiplier has a very regular structure and, in each clock cycle, allows the multiplication of u coefficients of one operand by all coefficients of the second operand. The parameter u is an unrolling factor, typically set to 1, 2, 4, etc. Consequently, the execution time of this multiplier is approximately equal to n/u clock cycles, and its area in LUTs is proportional to $n \cdot u$. Since "small" coefficients have from 2 to 4 bits, multiplication by them can be accomplished using ANDs, shifts, and additions. These operations can

TABLE 3: Features of algorithms and parameter sets affecting the choice of a multiplier type

	Small Coefficient Range	Number of coefficients	NTT friendly ring	One Operand in NTT domain	"Small" × "Large" Polynomial Multiplication in KeyGen/Encaps/Decaps	"Large" × "Large" Polynomial Multiplication in KeyGen/Encaps/Decaps
ntruhrss701		701			5/1/3	8*/–/1
ntruhps2048677	[-1..1]	677	N	N	5/1/3	8*/–/1
ntruhps4096821		821			5/1/3	8*/–/1
Kyber512	[-3..3], [-2..2]				4/6/8	–/–/–
Kyber768	[-2..2]	256	Y	Y	9/12/15	–/–/–
Kyber1024	[-2..2]				16/20/24	–/–/–
LightSaber-KEM	[-5..5]				4/6/8	–/–/–
Saber-KEM	[-4..4]	256	N	N	9/12/15	–/–/–
FireSaber-KEM	[-3..3]				16/20/24	–/–/–

* Performed in polynomial inversion in S_q

be efficiently implemented using LUTs and a special fast carry logic of Xilinx FPGAs, without the need for DSP units. Consequently, all polynomial multiplications in Saber and the majority of multiplications in NTRU can be efficiently implemented using the Schoolbook multiplier. The disadvantage is a relatively large area, even for the smallest value of the unrolling factor u=1.

An NTT-based multiplier has a much smaller area, independent of n, and the execution time proportional to $n \cdot lg_2(n)$. It can be sped up using a small number of DSP units. As a result, it is practical to instantiate several such multipliers within the same design without reaching the area threshold. The improvement in execution time depends on data dependencies and the relative speed of units producing inputs to the multipliers. An additional speed-up can be accomplished by defining and/or storing some inputs in the NTT domain. This way, the conversion from the regular to NTT domain may be skipped for one operand. Among the investigated algorithms, only Kyber is defined this way. NTT-based multipliers do not offer any advantage in terms of execution time for the case when one operand has small coefficients. They also impose specific requirements on the dependence between the number of coefficients in each operand, n, and the modulus q. If these dependencies do not hold, NTT may still be possible, but it requires extra computations, increasing the multiplier's area and possibly complicating control. Taking all these features into account, an NTT-based multiplier is an obvious choice only for CRYSTALS-Kyber.

As shown in Table 3, NTRU-HPS and NTRU-HRSS are the only investigated candidates that require multiplying two polynomials with "large" coefficients. Values of n and q do not fulfill the requirements of NTT. None of the operands is stored in the NTT domain. As a result, using a Toom-Cook multiplier appears to be the best choice. These multipliers have an area smaller than the Schoolbook and larger than NTT types. Similar to the NTT-based multipliers, the Toom-Cook multiplier requires using DSP units for integer multiplications. In NTRU, these multiplications can be sped up using a relatively moderate number of DSP units. Consequently, they appear to be the natural choice for the implementation of the "large" by "large" polynomial multiplication in the key generation and decapsulation operations of NTRU.

4.2 NTRU

The top-level diagram of NTRU is shown in Fig. 1. Below we describe the way of performing major operations of NTRU-HRSS and NTRU-HPS using this circuit.

4.2.1 Ternary Sampling

For NTRU-HRSS, the generation of polynomials f and g is performed in S_3 during key generation. Random bytes coming from SHAKE128 are reduced modulo 3 to obtain the ternary coefficients stored in a first-in, first-out (FIFO) unit. The sum of products of consecutive coefficients $s = \sum_i f_i f_{i+1}$ is computed at the same time. After finishing generating all coefficients, if $s < 0$, coefficients at even indices are signed-flipped before being transferred to the next computational stage. Thus, the non-negative correlation properties of f and g are satisfied. g is later multiplied by $x - 1$, which can be carried out trivially during the transfer. During encryption, r and m do not have either the non-negative correlation property or fixed-weight. They can be computed by simply reducing random data modulo 3.

For NTRU-HPS, f and r have arbitrary weight and can be sampled in a straightforward manner. However, m and g have fixed weight and are sampled by creating a random permutation of a list with a fixed number of values -1, 0, and 1. One can simply perform Fisher-Yates shuffle to have a random non-biased permutation of such a list. However, Fisher-Yates shuffle is not constant-time and creates a risk of potential timing attacks. Given that, we adopt a constant-time merge sorting approach for the permutation. The merge-sort module requires n random elements. Each element includes 30 random bits concatenated with "01" for the first $w/2$ elements, "10" for the next $d/2$ elements, and "00" for the rest. To get a 30-bit block, a 64-bit input is passed through a PISO, to be divided into two 32-bit blocks. Each 32-bit block is then processed using a buffer register and a variable shifter to get a 30-bit block. The leftover bits are stored in the buffer register to be concatenated with the subsequent output of PISO. After sorting, the upper 30 bits are discarded, and the lower 2 bits are converted from $\{0, 1, 2\}$ to $\{0, 1, -1\}$.

Related work: Wang et al. [4] proposed a fully pipelined constant-time merge sort module to generate random permutation in the Key Generation operation of Classic McEliece. To sort a random list of n elements, the module

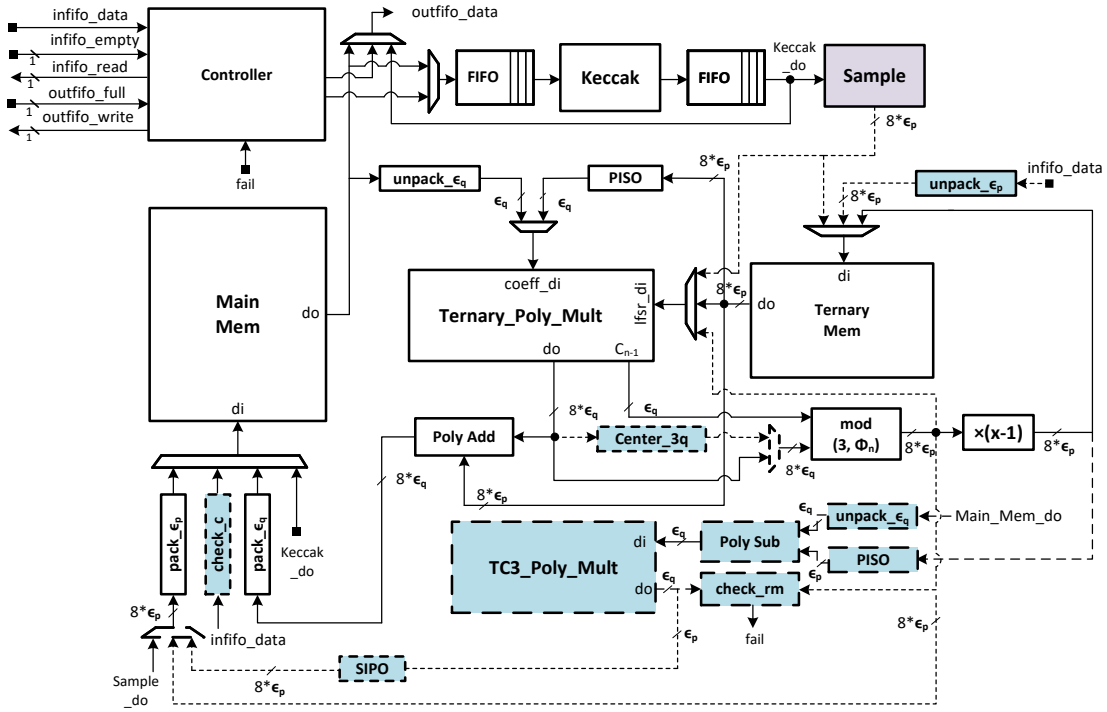

Fig. 1: Top-level block diagrams of the Encapsulation and Decapsulation modules of NTRU. The purple and blue modules are used only in Encapsulation and Decapsulation, respectively. All bus widths are 64 bits unless specified otherwise.

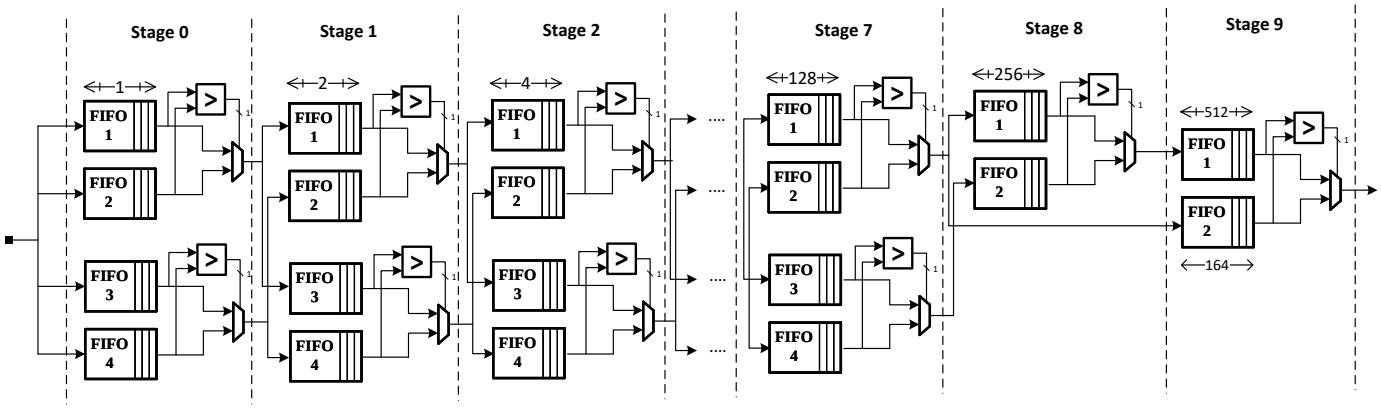

Fig. 2: FIFO-based merge sort module for NTRUHPS2048677.

needs $log_2(n)$ iterations, where each step requires $O(n)$ comparison operations. Therefore, the total cycle count is approximately equal to $nlog_2(n)$ cycles. Marotzke [17] implemented an iterative Batcher's merge exchange sort module for a very similar sampling function in the Streamlined NTRU Prime. Its operation also have asymptotic complexity of $O(nlog_2(n))$.

To speed up this operation, we use a merge-sort module consisting of $log_2(n)$ cascaded Sort Stages to sort the random sequences. The FIFO-based merge-sort module for NTRU-HPS677 is shown in Fig. 2. The inputs to each Sort Stage are two sorted lists, and the output is a sorted list of double input length, including all elements from the two input lists. Each input list is stored in a separate segment of memory. While the lower stages can be implemented by registers, the higher stages are implemented in dual-port memory. This approach can reduce the number of LUTs and FFs used to construct a large FIFO in higher stages at the cost of a small number of BRAMs. By using the dual-port memory, the controller in each stage can write out the sorted list to the next stage and receive other input lists from the previous stage simultaneously. By pipelining the operation of multiple Sort Stages, we can achieve a highly optimized latency for sorting. Our merge-sort module requires n clock cycles for reading n elements, roughly n cycles for sorting, and another n cycles to write out a sorted sequence. In particular, sampling m or g takes 2,678 and 3,343 cycles for NTRU-HPS677 and NTRU-HPS821, respectively.

4.2.2 *Polynomial Multiplication*

The schoolbook multiplication has quadratic-complexity but enables simple, parallel, easy-to-parameterize, and very fast architecture for polynomial multiplication in NTRU. It also can take advantage of "small" coefficients of one of the operands. However, for the multiplication of two polynomi-

als with "large" coefficients, this multiplier choice becomes sub-optimal, as explained in Section 4.1.

Toom-Cook and Karatsuba are multiplication algorithms with better asymptotic complexity than the schoolbok method. Toom-Cook k-way is a generalization of Karatsuba with $k = 2$. Both algorithms generally follow five steps: splitting, evaluation, pointwise multiplication, interpolation, and recomposition. The input polynomials are split into $2k - 1$ polynomials with n/k coefficients. These polynomials are then evaluated at $2k - 1$ points. The evaluated polynomials are multiplied in the pointwise-multiplication steps. The results are interpolated as the opposite of the evaluation step. The output polynomials of the interpolation step are finally recomposed into the final product.

Therefore, in this work, for a multiplication not involving ternary polynomial, we implement a Toom-Cook 3-way polynomial multiplier, which splits an n-coefficient polynomial multiplication into five multiplications with $n/3$ coefficients. The five multiplications are performed in parallel using five Odd-Even Karatsuba multipliers.

Our design was inspired by the software/hardware co-design from [18], in which Toom-Cook 4-way was applied to divide polynomial multiplication of 256 coefficients into seven multiplications with 64 coefficients. These seven multiplications were then run in parallel using seven schoolbook polynomial multipliers.

Our improvements over [18] include:

a) Our hardware implementation supports splitting input polynomials into three smaller polynomials before the Evaluation step. The Toom-Cook core in [18] relies on software to do this operation.

b) Using the Odd-Even Karatsuba method significantly improves the latency of the multiplication step.

c) Our core supports Recomposition, which has the output polynomial in the ring R_q. In [18], 5 output polynomials are transferred to software and are then recomposed into a single polynomial.

Toom-Cook 3-way splits input polynomial $A(x)$ into three polynomial a_0, a_1 and a_2 such that $A(y) = a_0 + a_1 y + a_2 y$, where $y = x^{\lceil n/3 \rceil}$. a_0, a_1 and a_2 are then evaluated at five points $\{0, 1, -1, 2, \infty\}$. The pointwise multiplications are performed by Odd-Even Karatsuba modules. We adopt the optimal sequence for evaluation and interpolation in the Toom-Cook 3-way from Bodrato et al. [19]. We would like to highlight that during evaluation, there is a division by 2, which becomes a one-bit shift right. Therefore, the pointwise multiplication and interpolation steps require one extra bit for each coefficient (a bit with the weight 2^{-1}). After interpolation steps, we have 5 output polynomials $c_0, c_1, \ldots c_4$ with $2 \cdot \lceil n/3 \rceil$ coefficients that need to be recomposed and reduced modulo Φ_n.

The overlap-free Karatsuba splits input polynomial $A(x)$ into two polynomials a_0 and a_1 such that $A(y) = a_0 + a_1 y$ where $y = x$. It means that a_0 consists of all even coefficients of A(x). Meanwhile, a_1 consists of all odd coefficients of A(x). The overlap-free Karatsuba scheme enables a more efficient alignment of product coefficients compared to the classic Karatsuba scheme.

We note that the splitting step is merged into evaluation. The interpolation and recomposition units work concurrently. Each splitting/evaluation or recomposition/interpolation takes $\lceil 2n/3 \rceil$ cycles. The pointwise-multiplication latency is given by $(\frac{n}{6\times 3} + 1) \times (\frac{n}{6} + 1)$. Our Toom-Cook multiplier finishes one polynomial multiplication in R_q or S_q in 5507, 5098 and 7274 cycles for $n = 701, 677$ and 821, respectively.

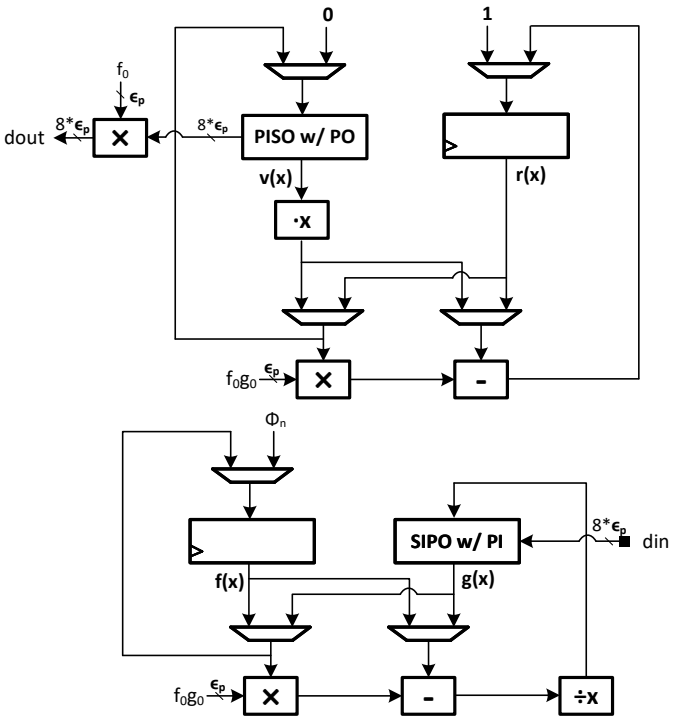

Fig. 3: S_2/S_3 Inversion Module. All bus widths are $n\epsilon_p$ bits unless specified otherwise ($\epsilon_p = \lceil log_2 p \rceil$).

For multiplications involving polynomial in the ternary form $\{-1, 0, 1\}$, we use the constant-time LFSR-based polynomial multiplier, proposed in [20], which has the latency of n clock cycles.

4.2.3 Inversion in S_3 and S_q

The inverse of polynomials in S_q and S_3 plays an important role in key generation. We need to compute f_p, which is an inverse of f in S_3 for the secret key. Computation of v_1, which is an inverse of v_0 in S_q, must be completed before any later operations could proceed [21].

Inversion in S_3: Inversion in S_3 is done using the constant-time extended Greatest Common Divisor (GCD) unit proposed in [22]. The top-level diagram of our `S3_inverse` module is shown in Fig. 3. At first, $g(x)$ is initialized with an input polynomial in reverse order, i.e., with the zeroth coefficient entering the corresponding Serial-In/Parallel-Out (SIPO) unit first. This unit has an additional parallel input and therefore is denoted as SIPO w/ PI. $f(x)$ and $r(x)$ are initialized with Φ_n and 1, respectively. $v(x)$ is stored in a Parallel-In/Serial-Out unit with Parallel Output (PISO w/ PO) and initialized with 0. The module runs in exactly $2(n-1)$ cycles. All coefficients of four polynomials are updated simultaneously during each iteration according to the value of δ and g_0. All operations, including addition, subtraction, and multiplication, have coefficients of resulting polynomials reduced modulo 3. Multiply and divide by x are performed by simple coefficient shifting. Lastly, the inverse of input polynomial is $f_0 \times v(x)$. We note

TABLE 4: Implementation results of the Extended GCD module and comparison with related work for Streamlined NTRU Prime in Zynq UltraScale+.

	Freq.	LUT	FF	BRAM	DSP	Cycles
$n = 761$ [17]	271	518	216	0	0	1,168,899
$n = 821$ **[TW]**	250	8,534	5,479	0	0	1,846

Algorithm 1 Polynomial Inversion in S_q [23]

Input: Polynomial a in S_q
Output: Polynomial b in S_q such that $a \cdot b = 1 \bmod (q, \Phi_n)$

1: $v_0 \leftarrow a^{-1} \bmod (2, \Phi_n)$
2: $i \leftarrow 1$
3: **while** $i < log_2 q$ **do**
4: $\quad v_0 \leftarrow v_0 \cdot (2 - a \cdot v_0)$
5: $\quad i \leftarrow 2i$
6: **end while**
7: $b \leftarrow v_0$

that the inverse polynomials are also stored in the reverse order. Our module also supports inversion in S_2, which is used in inversion in S_q. In Table 4, we compare our results for NTRU-HPS821 with $n = 821$ with the Reciprocal in $R/3$ module in the implementation of Streamlined NTRU Prime in [17]. These results demonstrate that the extended GCD can be implemented in an unrolled fashion, achieving highly optimized latency.

Inversion in S_q: To compute the inverse of h in S_q, we perform $h^{-1} \bmod (2, \Phi_n)$ and then apply a variant of the Newton iteration in S_q to obtain $h_q \equiv h^{-1} \bmod (q, \Phi_n)$ [23]. The pseudocode of inversion in S_q is given in Algorithm 1. A similar approach is presented in [23], which finds an inverse mod $(2, \Phi_n)$ using $h^{-1} \equiv h^{2^{n-1}-2} \bmod (2, \Phi_n)$. Given that squaring operation in $\mathbb{Z}_2[x]$ is particularly very efficient in software, this approach is suitable for software implementation. In our case, we can re-use our `S3_inverse` module to compute inversion in S_2. All arithmetic operations are now reduced modulo 2 instead of 3 as in inversion in S_3. Operations from lines 3 to 6 in Algorithm 1 are equivalent to 8 polynomial multiplications, which are performed by the Toom-Cook multiplier. Due to the long latency of the polynomial multiplication, inversion in S_q is the most time-consuming operation in Key Generation of NTRU.

4.2.4 `Lift` function

`Lift` function in NTRU-HPS applies a simple map to ternary coefficients of m, converting $\{0, 1, -1\}$ to $\{0, 1, q - 1\}$. This can done on-the-fly by sign extending all the coefficients from $\epsilon_p = 3$ bits to ϵ_q bits.

In NTRU-HRSS, the `Lift` function maps m from S_3 to R_q by doing $m \mapsto \Phi_1 \cdot S_3(m/\Phi_1)$ [23]. It can be performed by one multiplication with $z = 1/\Phi_n$, followed by reduction modulo $(3, \Phi_n)$ and multiplication by Φ_1. Since z is a constant ternary polynomial, it is stored in the memory, and the multiplication can be performed by the `Ternary_Poly_Mult` in n cycles. Reduction modulo $(3, \Phi_n)$ and multiplication by $\Phi_1 = x - 1$ can be performed on the fly while transferring the result back to the memory.

4.2.5 Operation scheduling

Almost all `pack` and `unpack` operations [21] are hidden by overlapping them with Polynomial Multiplications and/or Sampling. In Decapsulation, the checks for the validity of r and m are executed in parallel with packing them. The majority of the execution time of Decapsulation is taken by the "large" $\times$ "large" polynomial multiplication. Meanwhile, in Key Generation, the polynomial inversion in R_q takes up to 90% of total latency.

4.3 CRYSTALS-Kyber

The proposed hardware architecture for Round 3 Kyber supports the following variants and operations: a) CPA-PKE: Key Generation, Encryption, and Decryption, and b) CCA-KEM: Key Generation, Encapsulation, and Decapsulation. The top-level unit is shown in Fig. 4.

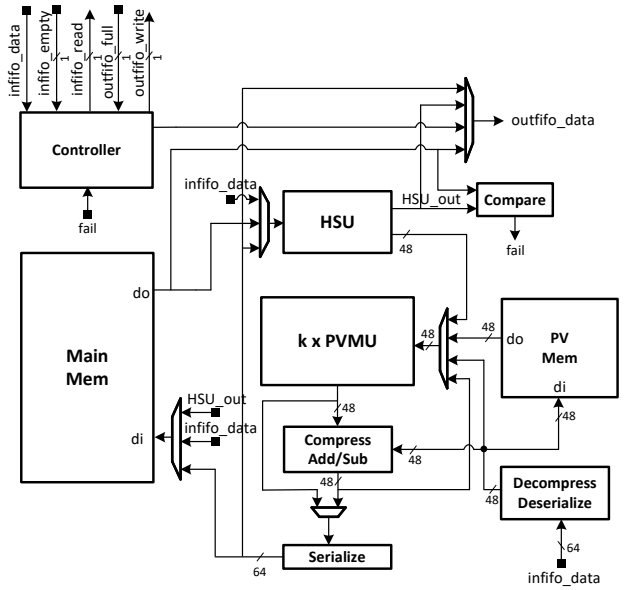

Fig. 4: Block diagram of the Kyber top-level datapath. All bus widths are 64 bits unless specified otherwise.

4.3.1 Polynomial NTT and Multiplication Unit

The Polynomial-Vector Multiplication Unit (PVMU) can perform forward and inverse NTT operations concurrently on up to k polynomials, where k is the security level parameter. This unit also performs polynomial point-wise multiplication (PWM) and accumulation to compute vector-vector and matrix-vector multiplications. The top-level block diagram of PVMU is shown in Fig. 5. At the security level k, the PVMU module consists of k DoubleButterfly pipelines and k memory banks (NTT RAM), each with a single read port and a single write port and datawidth of 4×12 bits (4 coefficients). On the input path, k FIFOs exist, which allow receiving up to k polynomials while a previous operation is underway and the main memory bank port is busy.

A DoubleButterfly pipeline consists of two merged parallel configurable radix-2 butterflies, which can operate in three modes of operation: DIT (Decimation in Time) NTT, DIF (Decimation in Frequency) iNTT, and point-wise multiplication (PWM). During the NTT/iNTT operations, each

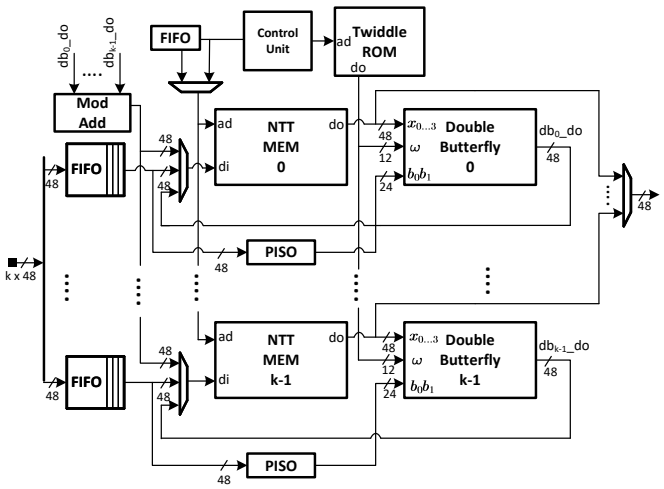

Fig. 5: Block diagram of the Kyber Polynomial-Vector Multiplication Unit (PVMU)

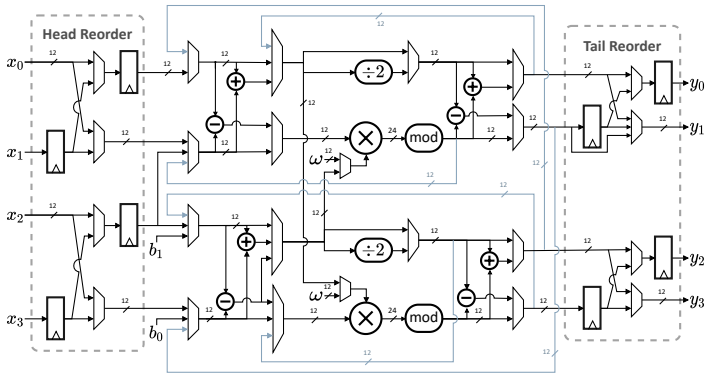

Fig. 6: Kyber DoubleButterfly and Reordering units

DoubleButterfly pipeline carries out two radix-2 butterfly operations in parallel for odd and even coefficients.

Depending on the butterfly operation (DIF/DIT/PWM), a reordering of the input is performed by the Head Reorder unit. A similar reordering of the outputs is performed by the Tail Reorder unit. During DIF/DIT operations, the reorder units operate as multi-path delay commutators (MDC) [24], ensuring the correct ordering of the stored coefficients and preventing the need for any subsequent reordering steps.

Kyber's point-wise multiplication of polynomials a and b is on degree 1 polynomials in the form of $a_{2i} + a_{2i+1}X$. While straightforward formulation requires 5 modular multiplications for each pair of output coefficients, by using the Karatsuba method, as demonstrated by Xing et al. [6], the number of multiplications can be reduced to 4. Through pipelining, a single DoubleButterfly unit is able to generate 2 PWM result coefficients every 2 cycles.

Design of the DoubleButterfly module along with Head and Tail Reordering units is depicted in Figure 6. The butterfly datapath is deeply pipelined (configurable, up to 12 stages), but pipeline registers are not shown in Figure 6 to keep the diagram uncluttered. The feedback paths, depicted in gray, are activated in PWM mode of operation, circulating each set of coefficients twice through the pipeline and performing the required multiplications, additions, and subtractions.

The inverse NTT operation involves scaling all coefficients by 256^{-1}. In many software and hardware implementations, the scaling step is performed separately, requiring 256 additional field multiplications for each polynomial. By performing a division by 2 $(\mathrm{mod}\ q)$ at each layer of inverse NTT, the scaling step can be entirely avoided. This observation was also used by Zhang et al. [25]. In that implementation, two divide-by-2 hardware units are utilized to scale both outputs of the radix-2 iNTT butterfly. In our implementation, we use a single divide-by-2 unit for each butterfly, and the other output of each butterfly is scaled by using a scaled copy of the twiddle factors during the inverse transform. The twiddle factors are stored in a single ROM shared by all butterfly pipelines and are mapped to BRAM-based memory during the FPGA synthesis.

4.3.2 Barrett reduction with support for division

Coefficients of polynomials are elements of a finite field (or ring) $\mathbf{Z}_q$, where q is a small constant modulus. In Kyber q is a prime. This choice requires a modular reduction step after most arithmetic operations to keep the bit width of the data bounded. Variants of Barrett [26], Montgomery [27], K-RED [28], and SAMS2 [29] reduction algorithms have been used in software and hardware implementations of R-LWE schemes.

We use an optimized variant of the Barrett reduction algorithm. As shown by Knezevic et al. [30], by careful selection of parameters α and β, only one level of conditional subtraction will be required. The hardware generator code creates optimized single constant multipliers (SCM) from shift-adder trees and ternary adders based on [31].

4.3.3 Hash and Sampling Unit

Kyber uses the SHA3-256 and SHA3-512 hash functions as well as SHAKE128 and SHAKE256 extendable-output functions. All of them are based on the Keccak permutation. Hash and Sampling Unit (HSU) integrates Keccak core with centered-binomial (CBD) and uniform rejection-based samplers, performing hashing operations in the FO transform as well as generation of noise polynomials and expansion of the public matrix $\mathbf{A}$. Our Keccak implementation takes advantage of the full-width, basic iterative architecture, which performs 24 rounds in 24 clock cycles. The data input and output are 64 bits wide with the valid-ready (decoupled) interface. In Kyber, all hashed data and base seed values (without the "nonce" bytes) are of lengths that are multiples of 64 bits. Based on this observation, we efficiently generate a padding word and append it to the input in a single cycle. The padding word includes specific SHA3/SHAKE padding bytes as well as one nonce byte when generating noise polynomials (CBD sampling) or two nonce bytes during the expansion of matrix $\mathbf{A}$. Keccak output is transferred from the state registers to a PISO to allow the next permutations to be performed while the output is consumed.

The MultiwidthCoverter module converts 64 bits of data from the output PISO of the Keccak code to the number of bits required for the selected sampling operation. This module is an improvement to the "Bus Width Converter" design introduced by Farahmand et al. in [32] with support for multiple output widths.

4.3.4 Centered Binomial Sampler

The CBD module in Kyber is responsible for performing binomial sampling. Kyber requires 12 bits of random data generated by the SHAKE module to generate four coefficients per clock cycle. Two CBD parameters η_1 and η_2 are used. $\eta_2 = 2$ for all security levels, $\eta_1 = 3$ for security Level 1 and $\eta_1 = 2$ for the other security levels. The samples are calculated from formula 1.

$$B_\eta = \sum_{i=1}^{\eta}(a_i - b_i) \tag{1}$$

Hamming weights of the input chunks of the size η are calculated, and negative results are mapped to equivalent $mod^+ q$ positive values.

4.3.5 Rejection-based Sampler

In order to minimize the size of the public key, the public matrix A (or its transpose A^T) is generated through the rejection-based sampling of a deterministic random source. The uniform random is generated using SHAKE128 from the public key seed. The output from SHAKE is partitioned into groups of 12 bits, and the resulting unsigned value is only accepted as a valid coefficient if it is less than $q = 3329$. This gives a probability of 81.27% for a sample to be valid. As k^2 sampled polynomials need to be generated through multiple invocations of the Keccak permutation and filtering of coefficients, this step is one of the bottlenecks in Kyber hardware scheduling. The rejection-based sampling of A is inherently not constant time, but any timing variation entirely depends on the public key seed and therefore would not expose any secrets. Our fast and efficient implementation of the uniform rejection sampling can process five coefficients (60 bits) in every clock cycle, requiring ≈ 106 cycles on average for the expansion of each polynomial in the public matrix (total of $\approx 106 \times k^2$ cycles for the matrix).

4.3.6 Improvements over Previous Work

TABLE 5: Implementation results of our NTT core (TW) and comparison with previous work on Artix-7, based on [9], Table VI. ($n = 256, q = 3,329$)

Work	Freq [MHz]	LUT	FF	DSP	BRAM	Cycles		
						NTT	INTT	PWM
[7]	222	801	717	4	2.0	324	324	–
[9]	115	360	145	3	2.0	940	1,203	1,289
[9]	115	737	290	6	4.0	474	602	1,289
[6]	161	1,579	1,058	2	3.0	448	448	256
TW	229	880	999	2	1.5	448	448	256

A state-of-the-art compact hardware implementation of Kyber is reported in [6]. Our design has been conducted independently. Both designs employ all relevant optimization techniques reported before.

Our high-level architecture and scheduling are based on using k DoubleButterfly units. In [6], only one pair of butterflies is used. Our DoubleButterfly datapath has a low area (around 726 LUTs), which allows efficient exploitation of Kyber's algorithm-level parallelism by employing k instances of DoubleButterfly, with k set to 2, 3, and 4 for the security levels 1, 3, and 5, respectively.

We utilize an efficient memory access scheme, reducing the memory requirement of each DoubleButterfly unit to a 1-read 1-write (1R1W) 64x48-bit RAM. In Xilinx FPGAs, this memory is mapped to a single BRAM tile (36 Kb block RAM) in the simple dual-port (SDP) mode of operation. Efficient "Head/Tail Re-order" units of the double-butterfly structure perform online re-ordering of coefficients entering/exiting the butterfly pipeline in NTT/iNTT (as a Multi-path Delay Commutator) as well as the re-ordering required for PWM/MAC. The double-butterfly structure computes the point-wise multiplication through the interleaved reiteration of the pipeline.

Our deeply pipelined butterfly implementation, including 12 stages, results in a higher maximum clock frequency. The optimized control circuit can skip pipeline flushing stalls whenever possible.

We have developed an optimized reduction unit based on a tweaked version of Barrett's algorithm. This unit has been shown to be faster and more efficient than the other implementations of modular reduction suggested in the literature. It also computes the division by q required for a fast and efficient implementation of the compression step. As a bonus, our hardware generation code works perfectly for any value of q, including the value used in CRYSTALS-Dilithium.

Finally, unlike [6], our design is technology-independent and does not employ any vendor-specific IPs. These features allow for easy deployment on FPGA platforms other than Xilinx, use of synthesis flows other than Vivado (including open-source FPGA flows), as well as porting to ASICs.

Bisheh-Niasar et al. in [7] present an accelerator for the NTT-based polynomial multiplication targeting Kyber. They utilize a 2×2 configurable butterfly architecture, which requires 29% fewer cycles for performing the NTT and iNTT operations compared to our 2×1 DoubleButterfly. They use a variation of the K-RED [28] modular reduction.

The comparison among the NTT units reported in [7], [9], [6], and this work is summarized in Table 5. For this work, the contribution of operations other than polynomial multiplication to the execution time of decapsulation at security level 3 is summarized in the rightmost column of Table 7. The contribution of all remaining operations is limited to about 26% of the total clock cycles due to their overlap with the NTT-unit operations for polynomial multiplication.

4.4 Saber

The top-level block diagram is shown in Fig. 7. Below we describe the way of performing major operations of Saber using this circuit.

4.4.1 Centered Binomial Sampler

For security levels 1, 3, and 5, the values of coefficients sampled from CBD are in the range [-5..5], [-4..4], or [-3..3], corresponding to the bit-widths w = 4, 4, 3. The 64-bit inputs are buffered in the dual-step shift register. After the shift register is full, chunks of data are read out and fed through a pure combinational logic to generate the coefficients. The output width of sampling modules is equal to $8 \cdot w$. Therefore, we have 8 samples generated per clock cycle.

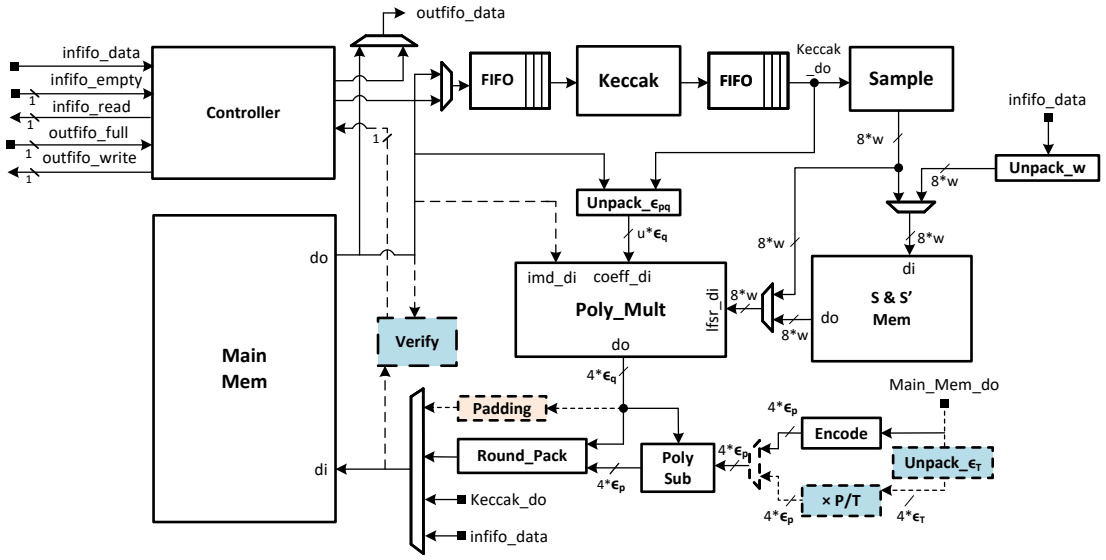

Fig. 7: Top-level block diagrams of Saber. All bus widths are 64 bits unless specified otherwise. The orange, blue modules are used only in key generation and decapsulation, respectively ($\epsilon_q = log_2 q$, $\epsilon_p = log_2 p$, and $\epsilon_T = log_2 T$).

TABLE 6: Implementation results of the Optimized Polynomial Multiplier using optimized integer multipliers vs. the centralized multiplier architecture in [12]

	Optimized Multiplier	Centralized Multiplier
LightSaber	**12,492 LUTs, 8,727 FFs**	13,658 LUTs, 8,727 FFs
Saber	12,492 LUTs, 8,727 FFs	**11,426 LUTs, 8,727 FFs**
FireSaber	**8,726 LUTs, 8,215 FFs**	8,734 LUTs, 8,215 FFs

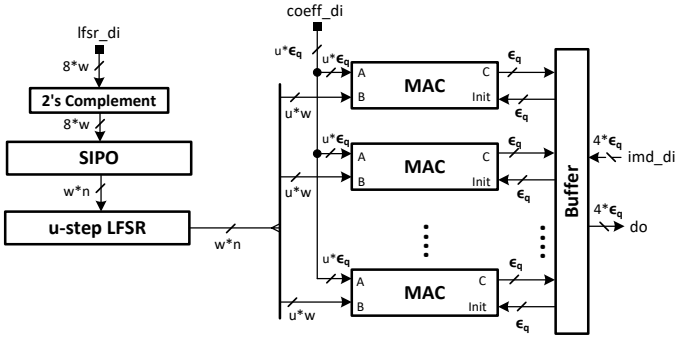

Fig. 8: Schoolbook-based polynomial multiplier with unroll factor $u = 1, 2, 4$.

4.4.2 Polynomial Multiplication

The block diagram of the schoolbook-based polynomial multiplier for Saber is shown in Fig. 8. Since there are multiple multiplications involved in vector-by-vector and matrix-by-vector multiplications, we improve the latency of multiplication by adding input and output buffers. The buffers are capable of pre-loading the next input polynomial and unloading the previous product polynomial at the same time as the current multiplication is performed. The `S&S'MEM` stores all small coefficients of secret polynomials in their unpacked form. Thus, it can provide one polynomial in 32 cycles. The latencies of loading and unloading polynomials are hidden in the multiplication latency. The multiplier can also be unrolled by a factor u = 1, 2, or 4, leading to the polynomial multiplication latency of 256, 128, or 64 cycles, respectively. Instead of having simple integer coefficient-wise multipliers, which are based on shift-add operations, as in [33], we generated optimized integer multipliers using an open-source tool FloPoCo [34]. We also tried the centralized coefficient-wise multiplier approach proposed in [12]. We report the results of the two approaches in Table 6. The centralized multipliers approach has better area consumption in the case of security level 3 (Saber). Thus, we use this approach for this specific parameter set. For security levels 1 and 5 (LightSaber and FireSaber), the optimized integer multipliers are used.

4.4.3 Operations Scheduling

During encapsulation and decapsulation, the generation of a vector of polynomials with "small" coefficients in CBD takes uniform inputs from the Keccak module. Whenever a polynomial is generated, it is then loaded into the polynomial multiplier and, at the same time, stored in `S&S'MEM` for later use. Multiplication can start as soon as a polynomial with "small" coefficients is fully loaded. For each cycle during multiplication, 1, 2, and 4 coefficients of a polynomial in A are fetched to the multiplier with the unrolling factor u = 1, 2, and 4, respectively. The `unpack_`ϵ_{pq} module serializes 64-bit data blocks into 13-bit coefficients. Since the multiplier consumes data at a slower rate than the `Keccak` module, `Keccak` module works intermittently when generating A. It stops its operation when there is still data left to be read by the multiplier. The next small noise polynomial is loaded in parallel with the multiplication of the current polynomials. Thus, the multiplier is always busy during vector-by-vector and matrix-by-vector multiplication. The result polynomials are rounded, packed, and stored into `Main_Mem`. During key generation, matrix A is generated in column-major order. The intermediate results of polynomials in vector b have to be stored in `Main_Mem`.

4.4.4 Improvements over Previous Work

The Karatsuba-based multiplier in [11] can execute polynomial multiplications in a very low number of clock cy-

TABLE 7: Comparison to Saber implementation in [10] for Level 3 Decapsulation based on [10], Table 1. In our designs, some operations have their execution time overlapping with others in the rows above. Hence, only their non-overlapping cycles, contributing to the total cycle count, are reported. Notation: TW - This work.

Operations	Decapsulation - CCAKEM L3		
	Saber [10]	Saber [TW]	Kyber [TW]
Polynomial multiplications	4,484	3,873	3,744
SHA3-256	303	294	449
SHA3-512	62	47	39
SHAKE128/256	1,403	51	428
CBD/Uniform Sampling	176	89	43
Remaining operations	1,606	328	330
Total Cycles	8,034	4,682	5,033

cles. By employing pipelined 8-level Karatsuba unit with efficient scheduling, one polynomial multiplication can be completed in 81 cycles. However, its area consumption is much higher than that of our multiplier in terms of DSP units (85 vs. 0) and BRAMs (6.0 vs. 1.5). Due to the recursive nature of the algorithm, the critical path through the multiplier is longer. Consequently, the circuit can only operate at low frequencies, 100 MHz and 160 MHz, for the unified and Saber Level 3 architecture, respectively. By unrolling our schoolbook-based multiplier, with the unrolling factors equal to 2 and 4, we obtain the designs Saber x2 and Saber x4. These designs can achieve comparable cycle counts while having a much higher maximum clock frequency and hence also better latency in μs. We note that the design in [11] targets an ASIC. Thus, both the area consumption and maximum frequency on FPGA might still be improved.

The high-speed instruction-set coprocessor in [10] offers flexibility in supporting multiple parameter sets in a unified architecture. The polynomial multiplier is then improved in [12]. Area consumption is significantly reduced while keeping the same latency in clock cycles. The coprocessor design, however, limits exploiting parallelism between non-data-dependent operations. In our design, many operations can be executed in parallel with polynomial multiplications. In Table 7, we show in detail how our efficient scheduling of operations for Saber Level 3 can improve the overall latency of decapsulation. The latency of polynomial multiplications is improved by 14% using buffers, allowing loading input and unloading output in parallel with multiplications. SHAKE128 and CBD sampling are almost fully overlapped. The remaining operations only include loading random input and ciphertext, rounding, and packing the final polynomial in the result vectors. Consequently, the total cycle count for decapsulation at Level 3 is reduced by more than 40%.

5 Results

All results reported in this section have been obtained after placing and routing using Xilinx Vivado 2020.2. In Tables 8, 9, and 10, we report Area-Time trade-off as a product of Total Time (Key Gen. + Encaps. + Decaps.) $\times$ number of LUTs. The improvements over previous work are also included. Notation: TW - This work.

5.1 NTRU

The results of our implementations of two variants of NTRU are summarized in Table 8. At security level 1, NTRU-HRSS outperforms NTRU-HPS for all operations in terms of the execution time in microseconds. NTRU-HRSS operates at a higher clock frequency but requires more LUTs than NTRU-HPS. With the increase in the security level, NTRU-HPS requires more FPGA resources, except for DSP units, the number of which remains the same.

5.2 CRYSTALS-Kyber

In Table 9, we report our results for CRYSTALS-Kyber and compare them with previous hardware-only implementations. We omit software/hardware implementations, as they are clearly inferior in terms of both the latency and the product of the latency and the number of LUTs.

The implementation of Kyber presented in this work slightly trails the implementation reported in [7] in terms of the execution time in clock cycles and time units at security level 1. At the same time, our design outperforms the design reported in [7] in terms of resource utilization and the area-time product for all security levels. The number of DSP units in our design is consistently two times smaller. Our implementation outperforms the design reported in [6], by a factor greater than two in terms of the execution time in microseconds for all major operations (key generation, encapsulation, and decapsulation). The comparison with [6] in terms of resource utilization is less obvious, considering that all operations are allowed to share the same resources in this work. In [6], the resource utilization for the server side (executing key generation and decapsulation) and the client side (executing encapsulation) are reported separately. However, based on our design, extending the coverage of operations from the server side to include encapsulation has negligible influence on the circuit area. Thus, it seems fair to compare our resource utilization numbers with the corresponding numbers for the server unit in [6]. The implementations reported in [9] and [8] are significantly less efficient. Ref. [8] also does not support key generation.

5.3 Saber

The comparison between our implementations of Saber and the best previous designs reported in [11] and [10] are shown in Table 10. The designs with the terms x1, x2, and x4 in the name are obtained by unrolling the schoolbook polynomial multiplication unit by 1, 2, and 4 times, respectively. Based on Table 10, Saber x4 and Saber x2 are the fastest, Saber x1 is the smallest among the compared high-speed implementations.

5.4 Comparison of Round 3 finalists

In Figs. 9, 10, and 11, we illustrate the dependence between the latency (in μs) of designs implemented using Zynq UltraScale+ SoC FPGAs and their resource utilization in LUTs. All other components of resource utilization, such as the number of BRAMs or DSP units, are omitted for simplicity. In terms of the percentage of the total amount of FPGA resources, the utilization of LUTs is the highest for all considered designs.

TABLE 8: Implementation results of NTRU on Zynq UltraScale+. The Area-Time Product (ATP-LUT) is calculated by Total Time (Key Gen. + Encaps + Decaps. in s) × LUT (the number of LUTs).

Design	Key/Encaps/Decaps [K Cycles]	Freq [MHz]	Key/Encaps/Decap [us]	LUT	FF	Slices	DSP	BR AM	ATP-LUT
Security Level 1									
NTRU-HRSS701 [TW]	51.8/2.2/8.8	300	172.7/7.4/29.4	57,301	41,124	10,617	45	2.5	12.01 (1.15)
NTRU-HPS677 [TW]	48.2/3.7/7.5	250	192.7/14.7/30.1	43,901	42,003	9,023	45	6.0	10.43 (1.00)
Security Level 3									
NTRU-HPS821 [TW]	67.2/4.6/10.2	250	268.6/18.3/40.8	53,619	47,362	11,823	45	6.5	17.58 (1.00)

TABLE 9: Implementation results for different Kyber instances on various FPGAs. Notation: S/C - Server/Client, E/D - Encapsulation/Decapsulation, Dev. - Device. The Area-Time Product (ATP-LUT) is calculated as Total Time (Key Gen. + Encaps + Decaps. in s) × LUT. A value in the parentheses is a ratio compared to the best design at the same security level.

Design	Key/Encaps/Decaps [K Cycles]	Freq [MHz]	Key/Encaps/Decaps [us]	LUT	FF	DSP	BR AM	ATP-LUT	Dev.
Kyber-CCAKEM L1									
Kyber R3 [TW]	2.1/3.3/4.5	220	9.7/14.8/20.3	9,347	8,186	4	6.0	0.419 (1.00)	A7
Kyber R3 [7]	1.9/2.4/3.8	200	9.4/12.2/18.8	10,502	9,859	8	13.0	0.424 (1.01)	A7
Kyber R3 [6]	3.8/5.1/6.7	S/C 161/167	23.4/30.5/41.3	S/C 7412/6785	S/C 4644/3981	S/C 3/3	S/C 2.0/2.0	0.65 (1.54)	A7
Kyber R2 [9]	4.0/7.0/10.0	115	34.8/60.9/87.0	18,000	5,000	6	15.0	3.29 (7.84)	A7
Kyber R2 [8]	–/49.0/68.8	155	–/316.0/444.0	E/D 80,322/88,901	–	E/D 54/354	E/D 200.5/202.0	61.04 (145.57)	A7
Kyber R3 [TW]	2.1/3.3/4.5	450	4.8/7.2/9.9	9,435	8,605	4	6.0	0.21 (1.00)	ZU+
Kyber-CCAKEM L3									
Kyber R3 [TW]	2.7/3.9/5.0	220	12.3/17.7/22.9	10,434	9,473	6	8.5	0.55 (1.00)	A7
Kyber R3 [7]	2.7/3.3/4.8	200	13.3/16.3/24.0	11,783	10,424	12	14.0	0.63 (1.15)	A7
Kyber R3 [6]	6.3/7.9/10	S/C 161/167	39.2/47.6/62.3	S/C 7412/6785	S/C 4644/3981	S/C 3/3	S/C 2.0/2.0	1.01 (1.83)	A7
Kyber R2 [9]	7/10/14	115	60.9/87.0/121.7	16,000	6,000	9	16.0	4.31 (7.82)	A7
Kyber R2 [8]	–/77.5/102.1	155	–/500.0/659.0	E/D 97,085/110,260	–	E/D 36/292	E/D 200.5/202.0	112.52 (204.03)	A7
Kyber R3 [TW]	2.7/3.9/5.0	450	6.0/8.6/11.2	10,512	10,105	6	8.5	0.27 (1.00)	ZU+
Kyber-CCAKEM L5									
Kyber R3 [TW]	3.6/4.8/6.0	220	16.3/21.8/27.1	11,527	10,767	8	10.5	0.75 (1.00)	A7
Kyber R3 [7]	3.5/4.1/6.3	185	18.7/22.3/33.8	13,347	11,639	16	16.0	1.00 (1.33)	A7
Kyber R3 [6]	9.4/11.3/13.9	S/C 161/167	58.2/67.9/86.2	S/C 7412/6785	S/C 4644/3981	S/C 3/3	S/C 2.0/2.0	1.44 (1.92)	A7
Kyber R2 [9]	10/14/18	112	89.3/125.0/160.7	16,000	6,000	12	17.0	6.00 (7.99)	A7
Kyber R2 [8]	–/107.1/135.6	192	–/558.0/706.0	E/D 119,189/132,918	–	E/D 36/548	E/D 200.5/202.0	150.65 (200.65)	A7
Kyber R3 [TW]	3.6/4.8/6.0	450	8.0/10.6/13.2	11,598	11,606	8	10.5	0.37 (1.00)	ZU+

TABLE 10: Implementation results of Saber and comparison with related works on Zynq UltraScale+ platform. The Area-Time Product (ATP-LUT) is calculated as Total Time (Key Gen. + Encaps + Decaps. in s) × LUT (the number of LUTs). A value in the parentheses is a ratio compared to the best design at the same security level.

Design	Key/Encaps/Decaps [K Cycles]	Freq [MHz]	Key/Encaps/Decap [us]	LUT	FF	Slices	DSP	BR AM	ATP-LUT
Security Level 1									
LightSaber x4 [TW]	0.9/1/1.3	310	2.9/3.3/4.2	65,890	28,230	10,404	0	1.5	0.69 (1.57)
LightSaber x2 [TW]	1.1/1.4/1.8	345	3.2/4.1/5.2	39,423	21,467	6,610	0	1.5	0.49 (1.13)
LightSaber x1 [TW]	1.6/2.2/2.8	370	4.3/5.8/7.6	24,688	14,785	4,309	0	1.5	0.44 (1.00)
Unified Saber [11]	0.6/0.9/1.1	100	6.0/8.6/10.8	34,886	9,858	—	85	6.0	0.89 (2.03)
Unified Saber [10]	2.8/4.0/5.0	150	18.4/26.9/33.6	24,979	10,732	—	0	2.0	1.97 (4.51)
Security Level 3									
Saber x4 [TW]	1.3/1.5/1.9	310	4.3/4.8/6.0	48,895	27,715	7,726	0	1.5	0.74 (1.15)
Saber x2 [TW]	1.8/2.2/2.8	345	5.2/6.5/8.1	32,099	21,037	5,294	0	1.5	0.64 (1.00)
Saber [11]	1.1/1.5/1.7	160	6.7/9.1/10.6	28,169	9,504	—	85	6.0	0.74 (1.15)
Saber x1 [TW]	2.7/3.7/4.7	370	7.3/10.1/12.7	21,352	14,232	3,763	0	1.5	0.64 (1.00)
Unified Saber [11]	1.1/1.5/1.7	100	10.7/14.6/17	34,886	9,858	—	85	6.0	1.48 (2.30)
Saber [10]	5.5/6.6/8.0	250	21.8/26.5/32.1	25,079	10,750	—	0	2.0	2.02 (3.14)
Unified Saber [10]	5.5/6.6/8.0	150	36.4/44.1/53.6	24,979	10,732	—	0	2.0	3.35 (5.21)
Security Level 5									
FireSaber x4 [TW]	2.0/2.1/2.6	310	6.5/6.9/8.5	38,268	27,677	6,348	0	1.5	0.84 (1.08)
FireSaber x2 [TW]	2.9/3.4/4.1	345	8.4/9.8/11.9	25,760	21,035	4,239	0	1.5	0.78 (1.00)
FireSaber x1 [TW]	4.9/5.9/7.1	370	13.2/15.9/19.3	20,383	14,239	3,408	0	1.5	0.99 (1.27)
Unified Saber [11]	1.7/2.2/2.5	100	17.2/21.9/24.8	34,886	9,858	—	85	6.0	2.23 (2.88)
Unified Saber [10]	9.0/10.3/12.3	150	60.2/68.4/82	24,979	10,732	—	0	2.0	5.26 (6.78)

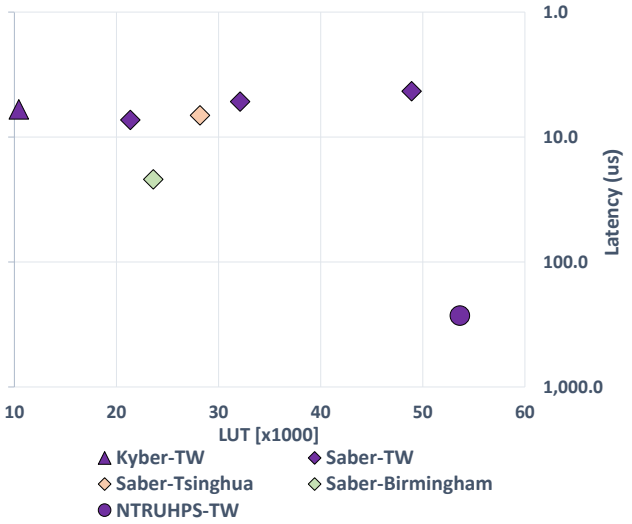

Fig. 9: L3, KeyGen, Zynq UltraScale+

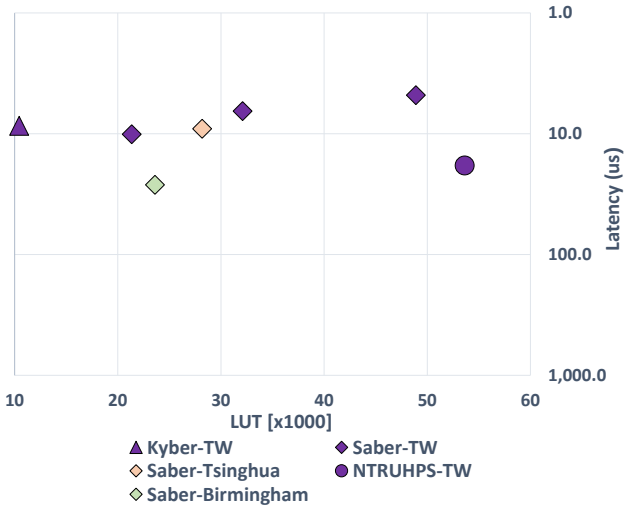

Fig. 10: L3, Encaps, Zynq UltraScale+

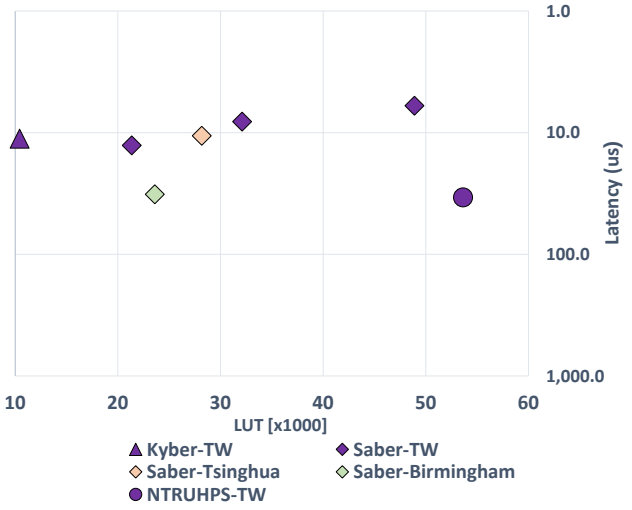

Fig. 11: L3, Decaps, Zynq UltraScale+

The exact correspondence between the names of designs given in these figures' legends and the related publications is as follows: <candidate_name>-TW: this work, Saber-Tsinghua: [11], Saber-Birmingham: [10]. Due to space constraints, we illustrate results only for security level 3 and Zynq UltraScale+. UltraScale+ was selected because of its use in previous work [10], [11]. Level 3 was selected because of the existence of the dedicated design, reported in [10], optimized specifically for this security level.

Saber is the fastest for all three major operations. From left to right, Saber-TW is represented by three diamonds corresponding to the x1, x2, and x4 architectures. Saber-TW x1 clearly outperforms Saber-Birmingham [10] in terms of both latency and resource utilization. Saber-TW x2 outperforms Saber-Tsinghua [11] in terms of speed, but uses slightly greater area. Kyber-TW is slightly faster than Saber-TW x1 and slower than Saber-TW x2 and Saber-TW x4. However, it uses only about a half of the LUTs of Saber-TW x1, about one-third of LUTs of Saber-TW x2, and about one-fifth of Saber-TW x4. NTRU-HPS is about 37-62x slower for key generation, about 3-7x slower for decapsulation, and about 2-4x slower for encapsulation (considering designs described in this paper). In terms of resource utilization, NTRU-HPS takes the largest number of LUTs, exceeding the number of LUTs for Saber-TW x4.

The performance of these candidates at security levels 1 and 5 can be determined by the analysis of data shown in Tables 8 (NTRU), 9 (Kyber), and 10 (Saber). The primary difference at level 1 is that NTRU can be represented by NTRU-HRSS, which for encapsulation has a performance matching that of Kyber. However, for key generation and decapsulation, the difference in performance remains at the similar level, independently whether NTRU-HRSS or NTRU-HPS are considered.

For security level 5, Kyber-TW outperforms FireSaber x1-TW by at least 46% in terms of the execution time in time units. At the same time, its resource utilization in LUTs is smaller by a factor of 1.8.

6 Conclusions

In terms of latency in time units, Saber and CRYSTALS-Kyber outperform NTRU by a factor 36-62 for key generation and 3-7 for decapsulation at security levels 1 and 3. For encapsulation at level 1, the performance of NTRU-HRSS is comparable to that of Kyber. For encapsulation at level 3, NTRU-HRSS is not supported, and the performance of NTRU-HPS lags behind that of Saber and Kyber by approximately a factor of 2-4. The differences between the two top candidates, Saber and Kyber, are relatively minor and may change as a result of future optimizations.

References

[1] D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds., *Post-Quantum Cryptography*. Springer, 2009.

[2] D. J. Bernstein, N. Heninger, P. Lou, and L. Valenta, "Post-quantum RSA," in *8th International Workshop on Post-Quantum Cryptography, PQCrypto 2017*. Cham: Springer, Jun. 2017, pp. 312–329.

[3] NIST, "Post-Quantum Cryptography: Call for Proposals," https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals, 2016.

[4] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes," in *9th International Conference on Post-Quantum Cryptography, PQCrypto 2018*. Fort Lauderdale, Florida: Springer, Apr. 2018, pp. 77–98.

[5] Z. Qin, R. Tong, X. Wu, G. Bai, L. Wu, and L. Su, "A Compact Full Hardware Implementation of PQC Algorithm NTRU," in *2021 International Conference on Communications, Information System and Computer Engineering (CISCE)*. IEEE, pp. 792–797.
[6] Y. Xing and S. Li, "A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 2, pp. 328–356, Feb. 2021.
[7] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "High-Speed NTT-based Polynomial Multiplication Accelerator for Post-Quantum Cryptography," in *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, Jun. 2021, pp. 94–101.
[8] Y. Huang, M. Huang, Z. Lei, and J. Wu, "A Pure Hardware Implementation of CRYSTALS-KYBER PQC Algorithm through Resource Reuse," *IEICE Electronics Express*, vol. 17, 2020.
[9] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "Instruction-Set Accelerated Implementation of CRYSTALS-Kyber," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 11, pp. 4648–4659.
[10] S. Sinha Roy and A. Basso, "High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 443–466, Aug. 2020.
[11] Y. Zhu, M. Zhu, B. Yang, W. Zhu, C. Deng, C. Chen, S. Wei, and L. Liu, "LWRpro: An Energy-Efficient Configurable Crypto-Processor for Module-LWR," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 3, pp. 1146–1159, 2021.
[12] A. Basso and S. S. Roy, "Optimized Polynomial Multiplier Architectures for Post-Quantum KEM Saber," in *58th Design Automation Conference, DAC 2021*, San Francisco, Dec. 2021.
[13] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "A Monolithic Hardware Implementation of Kyber: Comparing Apples to Apples in PQC Candidates," in *Progress in Cryptology – LATINCRYPT 2021*. Springer, pp. 108–126.
[14] M. Imran, F. Almeida, J. Raik, A. Basso, S. S. Roy, and S. Pagliarini, "Design Space Exploration of SABER in 65nm ASIC," in *5th Workshop on Attacks and Solutions in Hardware Security, ASHES 2021*. ACM, pp. 85–90.
[15] Aikata, A. C. Mert, D. Jacquemin, A. Das, D. Matthews, S. Ghosh, and S. S. Roy, "A Unified Cryptoprocessor for Lattice-based Signature and Key-exchange." [Online]. Available: https://eprint.iacr.org/2021/1461
[16] Dubois and Venetsanopoulos, "The Discrete Fourier Transform Over Finite Rings with Application to Fast Convolution," vol. C-27, no. 7, pp. 586–593.
[17] A. Marotzke, "A Constant Time Full Hardware Implementation of Streamlined NTRU Prime," in *19th International Conference on Smart Card Research and Advanced Applications, CARDIS 2020*. Springer, pp. 3–17.
[18] J. M. B. Mera, F. Turan, A. Karmakar, S. Sinha Roy, and I. Verbauwhede, "Compact domain-specific co-processor for accelerating module lattice-based KEM," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE, Jul. 2020, pp. 1–6.
[19] M. Bodrato and A. Zanoni, "Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices," in *International Symposium on Symbolic and Algebraic Computation, ISSAC 2007*, Jul. 2007, pp. 17–24.
[20] F. Farahmand, V. B. Dang, D. T. Nguyen, and K. Gaj, "Evaluating the Potential for Hardware Acceleration of Four NTRU-Based Key Encapsulation Mechanisms Using Software/Hardware Codesign," in *10th International Conference on Post-Quantum Cryptography, PQCrypto 2019*, ser. LNCS, vol. 11505. Chongqing, China: Springer, May 2019, pp. 23–43.
[21] C. Chen, O. Danba, J. Rijneveld, J. M. Schanck, T. Saito, P. Schwabe, W. Whyte, K. Xagawa, T. Yamakawa, and Z. Zhang, "NTRU: Algorithm Specifications And Supporting Documentation," Tech. Rep., Sep. 2020.
[22] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 3, pp. 340–398, May 2019.
[23] A. Hülsing, J. Rijneveld, J. Schanck, and P. Schwabe, "High-Speed Key Encapsulation from NTRU," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, vol. 10529, pp. 232–252.
[24] J.-H. Ye and M.-D. Shieh, "High-performance NTT architecture for large integer multiplication," in *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, Apr. 2018, pp. 1–4.
[25] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 49–72, Mar. 2020.
[26] P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," in *Advances in Cryptology — CRYPTO' 86*, ser. Lecture Notes in Computer Science, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer, 1987, pp. 311–323.
[27] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.
[28] P. Longa, M. Naehrig, P. Longa, and M. Naehrig, "Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography," in *Cryptology and Network Security - CANS 2016*, vol. 10052. Cham: Springer International Publishing, 2016, pp. 124–139.
[29] B. Liu and H. Wu, "Efficient architecture and implementation for NTRUEncrypt system," in *2015 IEEE 58th International Midwest Symposium on Circuits and Systems, MWSCAS 2015*, Fort Collins, CO, USA, Aug. 2015.
[30] M. Knezevic, F. Vercauteren, and I. Verbauwhede, "Faster Interleaved Modular Multiplication Based on Barrett and Montgomery Reduction Methods," *IEEE Transactions on Computers*, vol. 59, no. 12, pp. 1715–1721, Dec. 2010.
[31] M. Kumm, O. Gustafsson, M. Garrido, and P. Zipf, "Optimal Single Constant Multiplication Using Ternary Adders," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 7, pp. 928–932, Jul. 2018.
[32] F. Farahmand, M. U. Sharif, K. Briggs, and K. Gaj, "A High-Speed Constant-Time Hardware Implementation of NTRUEncrypt SVES," in *2018 International Conference on Field-Programmable Technology, FPT 2018*. IEEE, pp. 190–197.
[33] S. Sinha Roy and I. Verbauwhede, *Lattice-Based Public-Key Cryptography in Hardware*, ser. Computer Architecture and Design Methodologies. Singapore: Springer Singapore, 2020.
[34] F. de Dinechin, "Reflections on 10 Years of FloPoCo," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. Kyoto, Japan: IEEE, Jun. 2019, pp. 187–189.

Viet Ba Dang is a Graduate Research Assistant in the ECE Department at George Mason University. He received his BS degree in Computer Engineering from Danang University of Science and Technology in 2016.

Kamyar Mohajerani is a Graduate Research Assistant in the ECE Department at George Mason University. He received his BSc degree in Computer Engineering from Isfahan University of Technology in 2012 and his MSc degree in Computer Architecture from University of Tehran in 2016.

Kris Gaj is a professor in the ECE Department at George Mason University. He is a co-director of the Cryptographic Engineering Research Group (CERG). He has been involved in most previous and current cryptographic competitions, from AES to PQC.