PHOENIX: Crypto-Agile Hardware Sharing for ML-KEM and HQC

Antonio Ras[1], Antoine Loiseau[1], Mikael Carmona[1], Simon Pontié[1,3], Guénaël Renault[3,4], Benjamin Smith[3] and Emanuele Valea[5]

[1] Univ. Grenoble Alpes, CEA-Leti, Grenoble, France
{mikael.carmona,antoine.loiseau,simon.pontie,antonio.ras}@cea.fr
[2] CEA-Leti, Mines Saint-Étienne, Equipe Commune, Gardanne, France
[3] LIX, INRIA, CNRS, Ecole Polytechnique, Institut Polytechnique de Paris, France
smith@lix.polytechnique.fr
[4] ANSSI, Paris, France guenael.renault@ssi.gouv.fr
[5] Univ. Grenoble Alpes, CEA-List, Grenoble, France emanuele.valea@cea.fr

Abstract. The transition to quantum-safe public-key cryptography has begun: for key agreement, NIST has standardized ML-KEM and selected HQC for future standardization. The relative immaturity of these schemes encourages crypto-agile implementations, to facilitate easy transitions between them. Intelligent crypto-agility requires efficient sharing strategies to compute operations from different cryptosystems using the same resources. This is particularly challenging for cryptosystems with distinct mathematical foundations, like lattice-based ML-KEM and code-based HQC. We introduce PHOENIX, the first crypto-agile hardware coprocessor for lattice- and code-based cryptosystems—specifically, ML-KEM and HQC, at all three NIST security levels—with an effective agile sharing strategy. PHOENIX accelerates polynomial multiplication, which is the main operation in both cryptosystems, and the current bottleneck of HQC. To maximise sharing, we replace HQC's Karatsuba-based polynomial multiplication with the Frobenius Additive FFT (FAFFT), which is similar on an abstract level to ML-KEM's Number Theoretic Transform (NTT). We show that the FAFFT already brings substantial performance improvements in software. In hardware, our sharing strategy for the FAFFT and NTT is based on a new SuperButterfly unit that seamlessly switches between these two FFT variants over completely different rings. This is, to our knowledge, the first FAFFT hardware accelerator of any kind. We have integrated PHOENIX in a real System-on-Chip FPGA scenario, where our performance measurements show that efficient crypto-agility for lattice- and code-based KEMs can be achieved with low overhead.

Keywords: Post-Quantum Cryptography · Crypto-Agility · SW-HW FPGA codesign · Polynomial Multiplication sharing strategy · ML-KEM · NTT · HQC · FAFFT

1 Introduction

The transition to quantum-safe cryptography has begun [DRA+24], with the first international standards for post-quantum key encapsulation mechanisms (KEMs) and digital signatures appearing in 2024. But the relative immaturity—in security analysis and in implementation techniques—of schemes such as ML-KEM [Nat24b] (the first NIST standard post-quantum KEM) and HQC [AMAB+24] (selected by NIST for future standardization [AG25]) makes *crypto-agility* an important concern for long-term security [ASW+23].

Crypto-agility is promoted by NIST as a best practice to ensure the resilience of cryptographic infrastructure in the face of evolving threats [BCC+25]. Intuitively, crypto-agility means the ability to dynamically switch between different cryptographic algorithms

with minimal inconvenience. This switching might mean changing parameters within one system, or moving from one system to another completely different one, to maintain security in the face of advances in cryptanalysis.

For example: perhaps today we use ML-KEM 512. A significant advance in module-LWE cryptanalysis might mean revising ML-KEM's internal parameters to maintain an equivalent security level—or worse, it could force a migration to a KEM based on a different hard problem, such as code-based HQC.

Cryptographic migration is hard [LGH+24], and even harder when it must be done quickly. Crypto-agility anticipates and mitigates this difficulty starting at the design phase. This is especially challenging when specialized hardware is involved: development cycles are far longer, designed lifetimes of deployed systems are longer, and updates cannot be done by distributing a patch. Efficient hardware agility depends on the ability to execute common operations from different cryptosystems using shared hardware resources. Sometimes, this is trivial: for example, all new NIST standard cryptosystems use the Keccak-based hash functions and XOFs defined in FIPS 202 [Nat15], so a common core implementation of Keccak can serve many different cryptosystems. For another example, many lattice-based cryptosystems use the Number-Theoretic Transform (NTT) to perform polynomial multiplication. While different algorithms use different parameters, the nature of the function relies on the same mathematical operations, making it easy to integrate multiple cryptosystems using the same accelerator. But when we consider crypto-agility between cryptosystems from different paradigms—that is, with completely different underlying mathematical problems, different algorithms, and different implementation techniques in software and hardware—then finding ways to share computations is highly nontrivial.

1.1 Prior work

Lattice-based cryptosystems have seen plenty of agile hardware designs. Sapphire [BUC19] presented a pioneering configurable and efficient ASIC design for cryptosystems based on the hardness of various Learning With Errors (LWE) problems [Reg09]. Focusing on more recent work targeting the NIST standards ML-KEM [Nat24b] and ML-DSA [Nat24a], we find full hardware ASIC implementations including [KMK+24, AMI+22, KML+24]. Other works follow a hardware-software co-design approach, offloading data to be processed on separate FPGA-based hardware accelerators [NKDNPH24, LZL+24, YSZ+24], RISC-V processors [DMSS24, MBB+23, WZZ+24] (which can be directly implemented and integrated as configurable logic in FPGAs), and ASICs [KSFS24].

The situation is different for code-based cryptosystems, and HQC in particular. We know of only three full-hardware implementations of HQC [AMDD+22, DXN+23, ABPS24], though other HW/SW co-design proposals accelerate the polynomial multiplication [LST+23, THKX23, HTB+24], the Reed–Muller encoder [THCX24], the Reed–Solomon decoder [LST+23], and constant-time sampling [SFW23, HTX23]. Early reference implementations of HQC used a sparse-dense method for polynomial multiplication; but since the last update [AMAB+24], this has been replaced by 2-way Karatsuba to mitigate side-channel attacks, and this has become the main bottleneck of the algorithm. (The sparse-dense implementations in [ABPS24, HTB+24] were claimed to be constant-time and thus secure, but the shift to 2-way Karatsuba makes this work obsolete.)

For crypto-agile hardware supporting lattice-based *and* code-based cryptosystems, we find three full-hardware proposals [ZZL+22, ZZO+24, SKN+24], but none of these has an efficient hardware-sharing strategy. Koleci, Mazzetti, Martina, and Masera [KMMM23] proposed using the NTT to accelerate binary multiplication operations in code-based cryptosystems, but unfortunately this method is incompatible with HQC because of the internal construction of the algorithm.

1.2 Contributions

This article introduces PHOENIX, an agile hardware coprocessor to accelerate lattice-based ML-KEM *and* code-based HQC. PHOENIX is a loosely-coupled accelerator:[1] it can be used as a memory-mapped peripheral and integrated in any CPU-based System-on-Chip.

PHOENIX simultaneously tackles the polynomial-multiplication bottlenecks in ML-KEM and HQC. It includes a digital design that speeds up both the Number-Theoretic Transform (NTT) required for ML-KEM and the Frobenius Additive FFT (FAFFT), which we use to implement efficient polynomial multiplication in HQC for the first time in software and in hardware. This also represents first hardware acceleration of HQC mitigating earlier side-channel attacks on its original sparse-dense multiplication algorithm.

The NTT and FAFFT algorithms are accelerated using an innovative and efficient hardware-sharing strategy—indeed, the first hardware-sharing strategy for simultaneous acceleration of lattice- and code-based cryptosystems. Our strategy is based on a new SUPERBUTTERFLY circuit which computes various "butterfly" structures (with different arithmetic foundations) that appear in the NTT and FAFFT. The result is competitive hardware acceleration for the three security levels of both ML-KEM and HQC, while reducing the resource consumption that would be required by separate accelerators.

We have implemented PHOENIX and SUPERBUTTERFLY in a Xilinx Zynq-7020 FPGA SoC, with an efficient memory-access scheme to minimize the impact of slow data transfer on this platform. As a loosely-coupled accelerator, the PHOENIX architecture can easily be ported to other platforms, and extended to accelerate other lattice- and code-based cryptosystems such as ML-DSA.

Our benchmarks on this SoC show important gains in efficiency for ML-KEM and HQC (and also for a new, faster HQC variant where keys are stored in the FFT domain). They also show an encouragingly low overhead for crypto-agility.

2 Preliminaries

We begin with a minimal treatment of the KEMs we implemented, ML-KEM and HQC. We give only on the details needed to discuss implementation of the schemes; the underlying theory, hard problems, and security arguments are defined in the specifications.

Recall that a general KEM comprises three algorithms: Key Generation (`KeyGen`), Encapsulation (`Encaps`), and Decapsulation (`Decaps`). For Alice and Bob to establish a shared secret key SK over a public channel,

1. Alice runs `KeyGen`, which outputs her **public encapsulation key** EK and her **private decapsulation key** DK.
2. Bob obtains Alice's encapsulation key EK. He runs `Encaps` on EK to obtain a **ciphertext** CT and a **shared secret key** SK; he sends CT to Alice.
3. Alice receives CT from Bob, and runs `Decaps` on CT and DK to obtain SK.

2.1 ML-KEM

ML-KEM [Nat24b] is an IND-CCA2-secure lattice-based KEM, selected by NIST as the first PQC KEM standard. Table 1 lists ML-KEM parameter sets targeting NIST security levels I, III, and V. The polynomial length n and prime modulus q (the same in each parameter set) define a ring

$$\mathcal{R}_q := \mathbb{Z}_q[X]/(X^n + 1)\,,$$

[1] *Loosely-coupled hardware accelerators* operate independently of the central processing unit (CPU), with a minimum of synchronisation and interdependency, communicating with the CPU and on-chip memory through communication interfaces such as buses.

where $\mathbb{Z}_q$ is the finite field of integers modulo q, with values represented in the range $[0, q)$.

ML-KEM is constructed in two stages. The first defines K-PKE, an IND-CPA-secure public-key encryption scheme whose security relies on the hardness of the Module Learning With Errors problem (M-LWE) [LS15]. The second derives ML-KEM from K-PKE using a tweaked Fujisaki-Okamoto (FO) transform [FO99] to provide IND-CCA2 security.

K-PKE consists of three algorithms: key generation, encryption, and decryption.

- `K-PKE.KeyGen`: A matrix $\mathbf{A}$ is sampled uniformly from $\mathcal{R}_q^{k\times k}$, and secret vectors $\mathbf{s}, \mathbf{e}$ are sampled from binomial distributions on $\mathcal{R}_q^k$. We compute $\mathbf{t} := \mathbf{A}\mathbf{s} + \mathbf{e} \in \mathcal{R}_q^k$. The public key is $(\mathbf{A}, \mathbf{t})$, and the private key is $\mathbf{s}, \mathbf{e}$.
- `K-PKE.Encrypt`: To encrypt a 32-byte message m, we sample $\mathbf{r}, \mathbf{e}_1 \in \mathcal{R}_q^k$, $\mathbf{e}_2 \in \mathcal{R}_q$ from binomial distributions, then compute $\mathbf{u} := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1 \in \mathcal{R}_q^k$ and $v := \mathbf{t}^T\mathbf{r} + e_2 + \texttt{Decompress}_q(m, 1) \in \mathcal{R}_q$ (the function $\texttt{Decompress}_q(m, 1)$ encodes m as an element in $\mathcal{R}_q$). The ciphertext is $(\mathbf{u}, v)$.
- `K-PKE.Decrypt`: To decrypt a ciphertext $(\mathbf{u}, v)$ using the private key $\mathbf{s}$, we compute $m := \texttt{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1)$, where the function $\texttt{Compress}_q$ inverts $\texttt{Decompress}_q$.

The real computational "work" here is in the sampling (dominated by the cost of hashing with Keccak), and in the costly multiplications in $\mathcal{R}_q$ (which is what we will accelerate).

Table 1: ML-KEM Parameter Sets.

KEM	NIST Level	n	q	k	η_1	η_2	(d_u, d_v)
ML-KEM-512	1	256	3329	2	3	2	(10, 4)
ML-KEM-768	3	256	3329	3	2	2	(10, 4)
ML-KEM-1024	5	256	3329	4	2	2	(11, 5)

2.1.1 The Number-Theoretic Transform (NTT)

On the surface, multiplication in $\mathcal{R}_q$ means multiplying polynomials of degree $< n$ over $\mathbb{Z}_q$, then reducing the result modulo $X^n + 1$. But $X^n + 1$ factors over $\mathbb{Z}_q$ into a product of 128 irreducible quadratics:

$$X^n + 1 = \prod_{i=0}^{127} (X^2 - \zeta^{2\texttt{BitRev7}(i)+1})$$

where ζ is a primitive 256-th root of unity in $\mathbb{Z}_q$ (the ML-KEM standard takes $\zeta = 17$), and `BitRev7(`i`)` is defined to be the integer represented by bit-reversing the unsigned 7-bit value $0 \le i < 128$ (so $\texttt{BitRev7}(1) = 64$, $\texttt{BitRev7}(2) = 32$, $\texttt{BitRev7}(3) = 96$, etc.; note that $\{2\texttt{BitRev7}(i) + 1 : 0 \le i < 128\} = \{1, 3, 5, 7, \ldots, 253, 255\}$). This gives a ring isomorphism

$$\begin{aligned} \texttt{NTT} : \mathcal{R}_q &\longmapsto \mathcal{T}_q := \prod_{i=0}^{127} \mathbb{Z}_q[X]/(X^2 - \zeta^{2\texttt{BitRev7}(i)+1}) \\ f &\longmapsto \hat{f} := (f \bmod X^2 - \zeta^{2\texttt{BitRev7}(i)+1} : 0 \le i < 128) \in \mathcal{T}_q\,; \end{aligned}$$

the product ring $\mathcal{T}_q$ is called the *NTT domain*.

As we will see, the NTT is a highly efficient way of computing this isomorphism (rather than just as a series of slow mods). The NTT is the key to fast arithmetic in $\mathcal{R}_q$, since

$$f \times_{\mathcal{R}_q} g = \texttt{NTT}^{-1}(\texttt{NTT}(f) \times_{\mathcal{T}_q} \texttt{NTT}(g))\,;$$

the cost of a multiplication in $\mathcal{T}_q$ (which is n multiplications modulo quadratics) is much less than a multiplication in $\mathcal{R}_q$ (multiplication modulo a polynomial of degree $2n$), even without parallelization and vectorization. In fact, with this kind of multiplication the cost of $\texttt{NTT}$ and $\texttt{NTT}^{-1}$ becomes dominant.

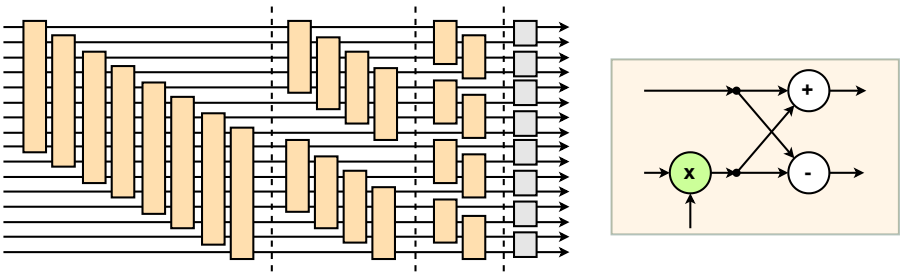
ML-KEM employs an incomplete $\texttt{NTT}$ to define the $\texttt{NTT}$ domain. That is,

$$\hat{f} = \texttt{NTT}(f) = (\hat{f}_0 + \hat{f}_1 X, \ldots, \hat{f}_{254} + \hat{f}_{255} X)$$

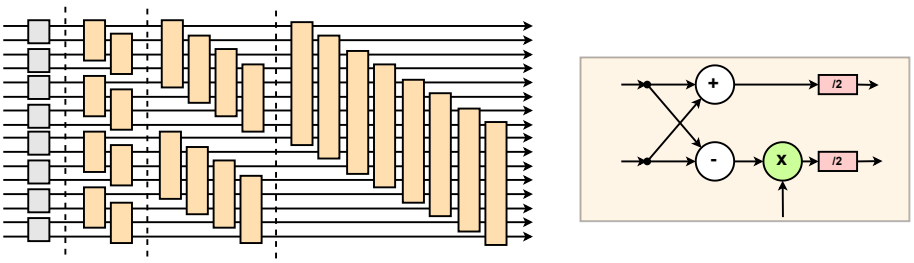
where

$$\hat{f}_{2i} := \sum_{j=0}^{127} f_{2j} \zeta^{(2\texttt{BitRev7}(i)+1)j} \qquad \text{and} \qquad \hat{f}_{2i+1} := \sum_{j=0}^{127} f_{2j+1} \zeta^{(2\texttt{BitRev7}(i)+1)j} .$$

Figure 1 illustrates the divide-and-conquer scheme of the NTT operation, and its inverse INTT. The highlighted process elements are called Cooley–Tukey (CT) and Gentleman–Sande (GS) **butterflies**.



(a) Incomplete $\texttt{NTT}_{\text{CT}}$ process using the Cooley–Tukey butterfly



(b) Incomplete $\texttt{INTT}_{\text{GS}}$ process using the Gentleman–Sande butterfly

Figure 1: Examples of NTT and INTT networks with their elementary process elements. The grey boxes represent unnecessary operations in the incomplete NTT-based processes.

The NTT defines a *Point-Wise Multiplication* ($\texttt{PWM}$) operation, denoted by $\hat{h} = \hat{f} \circ \hat{g}$, with components defined by

$$\hat{h}_{2i} = \hat{f}_{2i}\hat{g}_{2i} + \zeta^{2br(i)+1}\hat{f}_{2i+1}\hat{g}_{2i+1} \quad \text{and} \quad \hat{h}_{2i+1} = \hat{f}_{2i}\hat{f}_{2i+1} + \hat{g}_{2i}\hat{g}_{2i+1} . \tag{1}$$

This can be computed using five multiplication in $\mathbb{Z}_q$, but this can be reduced to four using an optimization based on Karatsuba's algorithm [XL21]. The calculation is done in two steps, each with two multiplications:

$\texttt{PWM0}$ computes $s_0 := \hat{f}_{2i} + \hat{f}_{2i+1}$, $s_1 := \hat{g}_{2i} + \hat{g}_{2i+1}$, $m_0 := \hat{f}_{2i} \cdot \hat{g}_{2i}$, $m_1 := \hat{f}_{2i+1} \cdot \hat{g}_{2i+1}$;

$\texttt{PWM1}$ computes $s_2 = m_0 + m_1$, $m_2 = s_0 \cdot s_1$, $m_3 = m_1 \cdot \zeta^{2\texttt{BitRev7}(i)+1}$, and then the final result $h_{2i} = m_0 + m_3$, $h_{2i+1} = m_2 - s_2$.

2.2 HQC

Hamming Quasi-Cyclic (HQC) [AMAB+24] is an IND-CCA2-secure *code-based* KEM, recently selected by NIST as a future standard. HQC uses concatenated Reed–Muller (RM) and Reed–Solomon (RS) codes to add errors to guarantee message confidentiality. Table 2 shows the parameters sets for the three HQC instances. The parameters n and w define a ring

$$\mathcal{R} := \mathbb{F}_2[X]/(X^n + 1)\,,$$

with a special subset $\mathcal{R}_w = \{v \in \mathcal{R} \mid \omega(v) = w\}$ where $\omega(\cdot)$ is the Hamming weight.

Table 2: HQC Parameter Sets

HQC	NIST Level	n_1	n_2	n	w	$w_{\mathbf{r}} = w_{\mathbf{e}}$
HQC-128	1	46	384	17,669	66	75
HQC-192	3	56	640	35,851	100	114
HQC-256	5	90	640	57,637	131	149

Like ML-KEM, the KEM is constructed in two stages: first defines `HQC.PKE`, an IND-CPA-secure public-key encryption scheme whose security relies on the hardness of the Quasi-Cyclic Syndrome Decoding problem (QCSD) [AMAB+24]. The second applies the Fujisaki–Okamoto-like Hofheinz–Hövelmanns–Kiltz (HHK) transform [HHK17] to `HQC.PKE` to derive the IND-CCA2-secure `HQC.KEM`.

`HQC.PKE` consists of three algorithms: key generation, encryption, and decryption.

- `HQC.PKE.KeyGen`: We uniformly sample a vector $\mathbf{h}$ from $\mathcal{R}$, and secret vectors $\mathbf{x}, \mathbf{y}$ from $\mathcal{R}_w$. The public key is $(\mathbf{h}, s := \mathbf{x} + \mathbf{h} \cdot \mathbf{y})$, and the private key is $(\mathbf{x}, \mathbf{y})$.
- `HQC.PKE.Encrypt`: To encrypt a 32-byte message m, we uniformly sample $\mathbf{r}_1, \mathbf{r}_2 \in \mathcal{R}_{w_r}$, $\mathbf{e} \in \mathcal{R}_{w_e}$, then compute $\mathbf{u} = \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2 \in \mathcal{R}$ and $\mathbf{v} = m\mathbf{G} + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{h} \in \mathcal{R}$, where $m\mathbf{G} = \mathcal{C}.\texttt{Encode}(m)$ is the encoding of m in the RMRS concatenated code. The ciphertext is $(\mathbf{u}, \mathbf{v})$.
- `HQC.PKE.Decrypt`: To decrypt a ciphertext $(\mathbf{u}, \mathbf{v})$ using the private key $\mathbf{y}$, we compute $m := \mathcal{C}.\texttt{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y})$, where $\mathcal{C}.\texttt{Decode}(\cdot)$ is the decoding process for the RMRS concatenated code.

Again, the real computational "work" here is in hashing with Keccak for the sampling and in the costly multiplications in $\mathcal{R}$ (which is what we will accelerate).

3 Binary Polynomial Multiplications using FAFFT

We now present the Frobenius AFFT (FAFFT), a state-of-the-art binary polynomial multiplication. We start by reviewing classic FFT multiplication in §3.1, then Cantor bases in §3.2. In §3.3 we present the Lin–Chung–Han Additive FFT (AFFT) [CCK+17] for binary multiplication, explaining how to improve its efficiency using the Frobenius partition technique of [vDHL17, LCK+18], and analyze the efficiency of the resulting FAFFT multiplication [CCK+18, CCK21]. Finally, in §3.4 we propose the parameters set to properly use this technique in HQC.

3.1 Binary Multiplication using Kronecker Segmentation

We briefly recall the now-standard technique of FFT polynomial multiplication using *Kronecker segmentation* (see [CLRS22, VZGG03] for further detail and background).

Suppose we want to compute the product $C(x)$ of two polynomials $A(x) = a_0 + \cdots + a_d x^d$ and $B(x) = b_0 + \cdots + b_d x^d \in \mathbb{F}_2[x]_{\leq d}$, each represented as a bit sequence of length $d+1$. Filling high-degree coefficients with 0 as required, we can assume $n = 2(d+1)$ is a power of 2 (note that the coefficients of C fit into a bit sequence of length n). Fix a parameter $0 < w \leq d$ and set $\ell := \lceil d/w \rceil$. Choose an irreducible polynomial g in $\mathbb{F}_2[x]$ of degree $m := 2w$, and let $\mathbb{F}_{2^m} = \mathbb{F}_2[z]/g(z)$. Now, to compute C we proceed as follows:

1. Partition the coefficients of A and B into ℓ blocks of w bits:
$$A(x) = \sum_{i=0}^{\ell-1} A_i(x) x^{iw} \quad \text{where } A_i(x) := a_{iw} + a_{iw+1}x + \cdots + a_{(i+1)w-1}x^{w-1},$$
$$B(x) = \sum_{i=0}^{\ell-1} B_i(x) x^{iw} \quad \text{where } B_i(x) := b_{iw} + b_{iw+1}x + \cdots + b_{(i+1)w-1}x^{w-1}.$$

2. Now, applying the map $x \mapsto z \in \mathbb{F}_{2^m}$ to each coefficient block, we set
$$A(x) \longmapsto A'(y) := a'_0 + a'_1 y + \cdots + a'_{l-1} y^{l-1} \in \mathbb{F}_{2^m}[y] \qquad \text{where } a'_i := A_i(z),$$
$$B(x) \longmapsto B'(y) := b'_0 + b'_1 y + \cdots + b'_{l-1} y^{l-1} \in \mathbb{F}_{2^m}[y] \qquad \text{where } b'_i := B_i(z).$$

3. Using the FFT, evaluate $A'(y)$ and $B'(y)$ at ℓ points $\alpha_0, \ldots, \alpha_{\ell-1}$ in $\mathbb{F}_{2^m}[y]_{\leq \ell}$ to get
$$(\hat{a}_0, \ldots, \hat{a}_{\ell-1}) := (A'(\alpha_0), \ldots, A'(\alpha_{\ell-1})),$$
$$(\hat{b}_0, \ldots, \hat{b}_{\ell-1}) := (B'(\alpha_0), \ldots, B'(\alpha_{\ell-1})).$$

4. Perform point-wise multiplication (PWM) on the ℓ evaluations:
$$(\hat{c}_0, \ldots, \hat{c}_{\ell-1}) := (\hat{a}_0 \cdot \hat{b}_0, \ldots, \hat{a}_{\ell-1} \cdot \hat{b}_{\ell-1}).$$

5. Use the IFFT to interpolate the $C'(y)$ such that $C'(\alpha_i) = \hat{c}_i$ for $0 \leq i < \ell$.

6. Map $C'(y)$ to $C(x) \in \mathbb{F}_2[x]_{\leq n}$ via $z \mapsto x$ and $y \mapsto x^w$, collecting terms and coefficients using the *InterleavedCombine* operation.

3.2 Cantor Basis Representations of Binary Fields

The representation of elements of $\mathbb{F}_{2^m}$ as vectors in $\mathbb{F}_2^m$ always depends on a choice of $\mathbb{F}_2$-basis $(\beta_0, \ldots, \beta_{m-1})$ of $\mathbb{F}_{2^m}$. Cantor [Can89] defined useful bases when m is a power of 2, i.e. $m = 2^{\ell_m}$ for some $\ell_m > 0$. Gao and Mateer [GM10] give an explicit recursive construction:

$$\beta_0 = 1 \qquad \text{and} \qquad \beta_i^2 + \beta_i = \beta_{i-1} \quad \text{for } 0 < i < m. \tag{2}$$

(Cantor bases are not unique: there are two choices for β_i at each step, differing by 1.)

Given a Cantor basis $(\beta_i)_{i=0}^{m-1}$ of $\mathbb{F}_{2^m}$, we have a sequence of subspaces

$$V_k := \langle \beta_0, \beta_1, ..., \beta_{k-1} \rangle \quad \text{for } 0 \leq k \leq m.$$

Note that $V_0 = \{0\}$, and V_k is the subfield $\mathbb{F}_{2^k} \subset \mathbb{F}_{2^m}$ if and only if k is a power of 2. Each of the V_k defines a *subspace vanishing polynomial*:

$$s_k(x) := \prod_{a \in V_k} (x - a) \quad \text{for } 0 \leq k \leq m.$$

These polynomials satisfy three important properties (see [Can89, GM10]):

- **Linearity**: $s_k(x+y) = s_k(x) + s_k(y)$ for all $0 \leq k \leq m$;
- **Two terms *for fields***: $s_k(x) = x^{2^k} + x \iff k$ is a power of 2, i.e., V_k is a field;
- **Recursivity**: $s_{k+1}(x) = s_k^2(x) + s_k(x) = s_1(s_k(x))$ for $0 \leq k < m$, and more generally $s_{k+i}(x) = s_k(s_i(x)) = s_i(s_k(x))$ for all $k \geq 0$ and $i \geq 0$ with $k + i \leq m$.

Evaluating $s_i(x)$. Evaluating $s_i(x)$ is an important operation for us, and the properties above allow us to do this very efficiently. First, by definition, $s_i(\beta_j) = 0$ for $j < i$. Then, rewriting (2) as $s_1(\beta_j) = \beta_{j-1}$ for $0 < j < m$, recursivity gives $s_i(\beta_j) = \beta_{j-i}$ for $i \le j < m$; then, by linearity, if we identify elements of $\mathbb{F}_{2^m}$ with their coefficient vectors, we get

$$s_i((a_0, \ldots, a_{m-1})) = (a_i, \ldots, a_{m-1}, 0, \ldots, 0) \quad \text{for} \quad i \ge 0\,. \tag{3}$$

That is: evaluating s_i corresponds to a simple shift of the coefficient vector by i places. This suggests a particularly convenient encoding of elements of $\mathbb{F}_{2^m}$ using unsigned m-bit integers: let $\phi_\beta : [0, 2^m) \to \mathbb{F}_{2^m}$ be the bijection defined by

$$\phi_\beta : \sum_{j=0}^{m-1} c_j 2^j \longmapsto \sum_{j=0}^{m-1} c_j \beta_j \quad \text{where } c_j \in \{0,1\} \text{ for all } 0 \le j < m\,.$$

With this encoding, Equation (3) becomes

$$s_i(\phi_\beta(n)) = \phi_\beta(n \gg i)$$

where $n \gg i$ means n shifted right by i bits.

3.3 Optimized multiplication in $\mathbb{F}_2[x]$ using the Frobenius AFFT

We can now present a series of optimizations to the general FFT-based binary polynomial multiplication scheme above.

3.3.1 Basis Conversion for Polynomials

Evaluation and interpolation of a polynomial $f \in \mathbb{F}_2[x]_{<n}$ at $n = 2^{l_n}$ points in $\mathbb{F}_{2^m}$ expressed in a Cantor basis is particularly efficient if f is represented in a particular basis of $\mathbb{F}_2[x]$ called `novelpoly`.

Definition 1. Given a Cantor basis $(\beta_0, \ldots, \beta_{m-1})$ for $\mathbb{F}_{2^m}$, the associated `novelpoly` basis [CCK+17] of $\mathbb{F}_2[x]$ is the sequence of polynomials $X_0, X_1, \ldots$ where

$$X_k(x) := \prod_{i \ge 0} (s_i(x))^{b_i} \quad \text{where} \quad k = \sum_{i \ge 0} b_i 2^i \text{ with } b_i \in \{0,1\} \tag{4}$$

and the s_i are the vanishing polynomials for the Cantor basis. That is: X_k is the product of all the $s_i(x)$ where the i-th bit of k is set. We will use the variable x for polynomials in the monomial basis, and X for polynomials in the `novelpoly` basis.

We need to convert a polynomial $f(x)$ in the usual monomial basis to and from a polynomial $g(X)$ in the `novelpoly` basis. The algorithm proposed in [BC14] converts $f(x)$ to $g(X)$ recursively by finding the largest i such that $2^i < \deg(f)$, then dividing $f_i(x)$ by $s_i(x)$ to get $f(x) = f_0(x) + s_i(x) f_1(x)$; then f_0 and f_1 are divided again, and so on. Division by $s_i(x)$ is performed by `XOR` operations, with the number of `XOR`s depending on the number of *non-zero* terms in $s_i(x)$. To reduce the number of `XOR`s, [LANH16] finds the largest i such that $2^{2^i} = \deg(f)$ and then performs variable substitution to express f as a power series in s_{2^i}. By the **two terms for fields** property of the vanishing polynomials, this optimization selects only the $s_i(x)$ which contains two terms, consequently reducing the cost of each division of the basis conversion. See [CCK+17, §2.4] for more details.

3.3.2 FFT-like Operations and their Butterfly Structures

Given $f(x) \in \mathbb{F}_{2^m}[x]_{<n}$ converted to $g(X)$ in the `novelpoly` basis, we can efficiently evaluate f at the set $\alpha + V_{\ell_n} = \{\alpha + u \mid u \in V_{\ell_n}\}$ for $\alpha \in \mathbb{F}_{2^m}$ (where $n = 2^{\ell_n}$) using the $\mathtt{FFT}_{\mathrm{LCH}}$ algorithm [CCK+17, §2.3.2, Alg. 1].

The algorithm follows a typical divide-and-conquer process: we rewrite $f(x) = g(X) = p_0(X) + X_{2^{i_n-1}}(x) \cdot p_1(X)$, with p_0 and p_1 half the length of g. The general idea of evaluating at all points of V_{ℓ_n} is to divide the subspace in two sets, V_{ℓ_n-1} and $V_{\ell_n}/V_{\ell_n-1} = V_{\ell_n-1} + \beta_{\ell_n-1} := \{x + \beta_{\ell_n-1} : x \in V_{\ell_n-1}\}$. Since $s_i(x) = X_{2^i}(x)$ is linear, evaluations at $V_i + \beta_i$ share common computations with the evaluation at V_i.

In practice, given $\alpha + V_{\ell_n}$ as the starting evaluation point set, we first find $\max(i)$ such that $2^i < n$. Then we divide by $s_i(x)$ to get $g(X) = p_0(X) + s_i(x) \cdot p_1(X)$. Then, we use the polynomials $p_i(X)$ to perform an elementary butterfly structure, here called an $\texttt{FFT}_{\text{LCH}}$ butterfly:

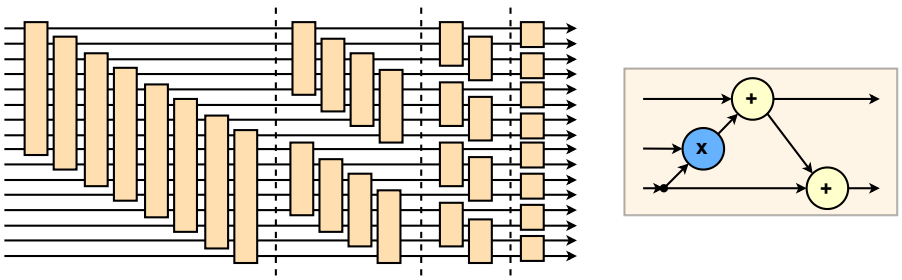
$$h_0(X) \leftarrow p_0(X) + s_i(\alpha) \cdot p_1(X)\,,$$
$$h_1(X) \leftarrow h_0(X) + p_1(X)\,.$$

At this point, the algorithm continue recursively its execution of the new half-sized polynomials. To do that, it recalls the $\texttt{FFT}_{\text{LCH}}$ function passing as scalar input α and $\alpha + \beta_i$, for the respectively $h_0(X)$ and $h_1(X)$ polynomials.

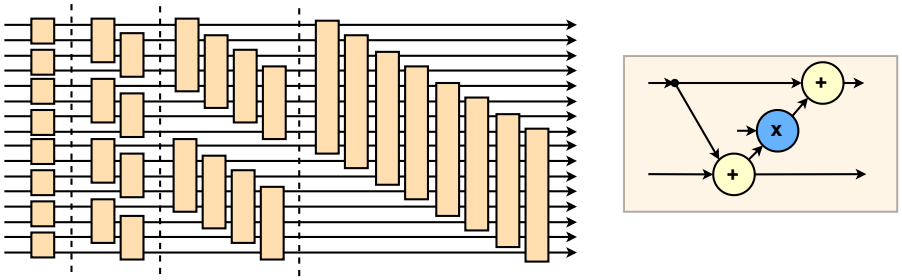
The inverse process $\texttt{IFFT}_{\text{LCH}}$ follows a *unit-and-build* approach: starting from the known $h_0(X)$ and $h_1(X)$, it computes the original value $p_0(X)$, $p_1(X)$ using the $\texttt{IFFT}_{\text{LCH}}$ butterfly structure

$$p_0(X) \leftarrow h_0(X) + s_i(\alpha) \cdot p_1(X)\,,$$
$$p_1(X) \leftarrow h_0(X) + h_1(X)\,.$$

The whole process is composed of ℓ_n layers, each of $n/2$ butterfly operations. Figure 2 illustrates the $\texttt{FFT}_{\text{LCH}}$ and an $\texttt{IFFT}_{\text{LCH}}$ processes, highlighting the butterfly structures above.



(a) The HQC $\texttt{FFT}_{\text{LCH}}$ process and its elementary processing element, the $\texttt{FFT}_{\text{LCH}}$ butterfly.



(b) HQC $\texttt{IFFT}_{\text{LCH}}$ process and its elementary processing element, the $\texttt{IFFT}_{\text{LCH}}$ butterfly.

Figure 2: Example of a $\texttt{FFT}_{\text{LCH}}$ and $\texttt{IFFT}_{\text{LCH}}$ network with the $\texttt{FFT}_{\text{LCH}}$ butterfly and the $\texttt{IFFT}_{\text{LCH}}$ butterfly.

3.3.3 The Frobenius map and the new evaluated points

To multiply polynomials f and $g \in \mathbb{F}_2[x]_{<d}$ such that $\deg(f \cdot g) < n = 2^{\ell_n} = 2(d+1)$, we can split f, g into ℓ-bit blocks, apply the `AFFT` to f and g in $\mathbb{F}_{2^m}[x]$ to evaluate them at V_{ℓ_n}, perform PWM, and finally apply the inverse `AFFT` to obtain $(f \cdot g)(x)$. Here the `AFFT` procedure first performs polynomial basis conversion using a function `BasisCvt`, and uses $\texttt{FFT}_{\text{LCH}}$ to evaluate polynomials in the `novelpoly` basis; the inverse `AFFT` performs $\texttt{IFFT}_{\text{LCH}}$ and `iBasisCvt`. This process exactly describes Kronecker segmentation algorithm, in 3.1, using evaluation points in Cantor basis.

Definition 2. We let ϕ_2 denote the Frobenius map (squaring) over $\mathbb{F}_2$, mapping $a \in \mathbb{F}_{2^m}$ to $\phi_2(a) := a^2$. Note that

$$C(\phi_2(a)) = \phi_2(C(a)) \quad \text{for all } C \in \mathbb{F}_2[x] \tag{5}$$

which means that the value of C at point $\phi_2(a)$ can be derived from the value $C(a)$ by computing $\phi_2(C(a))$.

The n evaluation points can be reduced using to $n_p = n/m$ using the Frobenius map and, as suggested in [LCK+18], by taking the set of evaluation point to be

$$\Sigma = \beta_{\ell_{n_p}+m/2} + V_{\ell_{n_p}} \tag{6}$$

where $\ell_{n_p} = \log(n_p) < m/2$ and $\ell_{n_p} + m/2 < m$. Note that $\#\Sigma = n_p = n/m$. The *Frobenius AFFT* (FAFFT) is the `AFFT` with Σ as the evaluation set, and exploiting ϕ_2 for evaluation.

3.3.4 Truncating the FAFFT using the Encoding function

As shown in [LCK+18], given $f \in \mathbb{F}_2[x]_{<n}$ and the point sets $\hat{\Sigma} = \alpha + V_{(\ell_{n_p}+\ell_m)}$, where $\alpha = \beta_{\ell_{n_p}+m/2}$, evaluating f at Σ corresponds to performing a traditional size-n evaluation of f at $\hat{\Sigma}$, avoiding all the redundant operations. This is called the `Truncated FAFFT`, or simply `FAFFT`.

Definition 3. Given $\hat{\Sigma}$, we define the invertible linear map $\texttt{encode} : \mathbb{F}_2[x]_{<n} \mapsto \mathbb{F}_{2^m}^{n_p}[x]$ by

$$\sum_{i=0}^{n} f_i x^i \in \mathbb{F}_2[x] \longmapsto \sum_{i=0}^{n_{p-1}} f'_i x^i \in \mathbb{F}_{2^m}^{n_p}[x]$$

where

$$f'_i := \sum_{j=0}^{m-1} r_j \cdot f_{j \cdot n_p + i} \quad \text{with} \quad r_j = \prod_{k=1}^{l_m} (\beta_{m/2-k})^{j_k} \quad \text{and} \quad j := \sum_{i=0}^{\ell_m} j_i 2^i \,.$$

The function `encode` performs the first ℓ_m layers of the $\texttt{FFT}_{\text{LCH}}$ process. The `Truncated FAFFT` therefore starts with the usual `novelpoly` basis conversion for a size-n polynomial. Then, ℓ_m layers of the size-n $\texttt{FFT}_{\text{LCH}}$ process is done with `encode`; finally, the last ℓ_{n_p} layers are computed with the usual size-n_p $\texttt{FFT}_{\text{LCH}}$ process over Σ.

3.4 Overall FAFFT Polynomial Multiplication

Algorithm 1 reproduces the binary polynomial-multiplication algorithm of [CCK21]. As noted above, the operation relies on six processes: `BasisCvt`, `Encode`, $\texttt{FFT}_{\text{LCH}}$, and their respective inverses `iBasisCvt`, `Decode`, and $\texttt{IFFT}_{\text{LCH}}$.

To evaluate two input polynomials $a(x)$ and $b(x) \in \mathbb{F}_2[x]_{<n}$ at the n_p evaluation points in $\mathbb{F}_{2^m}[x]$, defined by the Frobenius map, we first apply `BasisCvt` to convert them to the

Algorithm 1: FAFFT-based Polynomial Multiplication

Input: n: Maximum length of the output polynomial.

Procedure *FAFFT*$(f(x) \in \mathbb{F}_2[x]_{<d}, \Sigma)$
- $(f_0, \ldots, f_{n-1}) \in \mathbb{F}_2^n \leftarrow \texttt{BasisCvt}(f(x))$
- $(f'_0, \ldots, f'_{n_p-1}) \in \mathbb{F}_{2^m}^{n_p} \leftarrow \texttt{Encode}((f_0, f_1, \ldots, f_{n-1}), \Sigma)$
- $(\hat{f}_0, \ldots, \hat{f}_{n_p-1}) \in \mathbb{F}_{2^m}^{n_p} \leftarrow \texttt{FFT}_{\text{LCH}}((f'_0, \ldots, f'_{n_p-1}), \Sigma)$
- **return** $(\hat{f}_0, \ldots, \hat{f}_{n_p-1})$

Procedure *FAFFT*$^{-1}(\hat{f}(x) \in \mathbb{F}_{2^m}^{n_p}, \Sigma)$
- $(f'_0, \ldots, f'_{n_p-1}) \in \mathbb{F}_{2^m}^{n_p} \leftarrow \texttt{IFFT}_{\text{LCH}}((\hat{f}_0, \ldots, \hat{f}_{n_p-1}), \Sigma)$
- $(f_0, \ldots, f_{n-1}) \in \mathbb{F}_2^n \leftarrow \texttt{Decode}((f'_0, f'_1, \ldots, f'_{n-1}), \Sigma)$
- $f(x) \in \mathbb{F}_2[x]_{<n} \leftarrow \texttt{iBasisCvt}(f_0, f_1, \ldots, f_{n-1})$
- **return** $f(x)$

Procedure *BitPolyMult*$(a(x), b(x) \in \mathbb{F}_2[x]_{<d})$
- $(\hat{a}_0, \ldots, \hat{a}_{n_p-1}) \in \mathbb{F}_{2^m}^{n_p} \leftarrow \texttt{FAFFT}(a(x), \Sigma)$
- $(\hat{b}_0, \ldots, \hat{b}_{n_p-1}) \in \mathbb{F}_{2^m}^{n_p} \leftarrow \texttt{FAFFT}(b(x), \Sigma)$
- $(\hat{c}_0, \ldots, \hat{c}_{n_p-1}) \in \mathbb{F}_{2^m}^{n_p} \leftarrow (\hat{a}_0 \cdot \hat{b}_0, \ldots, \hat{a}_{n_p-1} \cdot \hat{b}_{n_p-1})$ `// PWM`
- $c(x) \in \mathbb{F}_2[x]_{<n} \leftarrow \texttt{FAFFT}^{-1}((\hat{c}_0, \ldots, \hat{c}_{n_p-1}), \Sigma)$
- **return** $c(x)$

`novelpoly` basis. Then we apply `Encode` to virtually execute the first ℓ_m layers of $\texttt{FFT}_{\text{LCH}}$. The resulting polynomials $a'(X), b'(X) \in \mathbb{F}_{2^m}^{n_p}[x]$ are represented with n_p coefficients in $\mathbb{F}_{2^m}[x]$. The remaining ℓ_{n_p} butterfly layers are processed by performing an usual size-n_p $\texttt{FFT}_{\text{LCH}}$ process with the evaluation-point set Σ. This yields $\hat{a}(X)$ and $\hat{b}(X) \in \mathbb{F}_{2^m}^{n_p}[x]$. To finalize the polynomial multiplication we first perform a linear point-wise multiplication, $\texttt{PWM}_{\text{LCH}}$, which amounts to ℓ_{n_p} multiplications in $\mathbb{F}_{2^m}[x]$; then we interpolate and convert back, using $\texttt{IFFT}_{\text{LCH}}$, `Decode`, and `iBasisCvt` to get the final result.

Table 3 lists the parameters we used to apply Algorithm 1 in HQC. The extension field used throughout is $\mathbb{F}_{2^{32}} = \mathbb{F}_2[x]/(g(x))$, where $g(x) = x^{32} + x^{22} + x^2 + x + 1$.

Table 3: FAFFT parameter for all HQC levels. Here n_{HQC} is the n-parameter from the HQC specification, and n denotes the next power of 2.

	n_{HQC}	n	m	n_p	l_{n_p}	Σ
`HQC-128`	17,669	65536	32	2048	11	$\beta_{27} + V_{11}$
`HQC-192`	35,851	131072	32	4096	12	$\beta_{28} + V_{12}$
`HQC-256`	57,637	131072	32	4096	12	$\beta_{28} + V_{12}$

4 Polynomial multiplication sharing strategy

We now define our hardware-based sharing strategy for FFT-based polynomial multiplication in ML-KEM and HQC. As we saw above, these polynomial multiplications follow the same operational approach and data handling scheme, but there are critical differences in the butterfly structures: first, the rings $\mathcal{R}_q$ (for ML-KEM) and $\mathbb{F}_{2^m}$ (for HQC) have completely different arithmetic operations, and second, each butterfly has a different order of dependence of operations. Our goal is to perform all of these different butterflies (and ultimately, both polynomial multiplications) using a single configurable agile hardware design.

Our solution, SuperButterfly (described in §4.1), varies the execution of arithmetic operations while reducing resource consumption and communication overhead. We present Phoenix, our SuperButterfly-based hardware accelerator, in §4.2, and report experimental results in §5.

4.1 SuperButterfly

SuperButterfly is a configurable digital circuit that can perform all the butterfly structures involved in the targeted `NTT` and `FAFFT` polynomial multiplication operations.

A SuperButterfly Unit (SBU) (see Figure 3) is a pipelined hardware architecture with a latency of 8 clock cycles that processes 32-bit data. It takes three input data—two for coefficients and one for constants—and provides two outputs. The SBU can be configured to compute

- *one* AFFT-based butterfly for HQC, which requires a 32-bit wide data path; or
- *two* parallel NTT-based butterflies for ML-KEM, operating on 12-bit coefficients in the higher and lower 16-bit halves of the data path.

This reduces the data dependency problem that occurs in the `PWM1` configuration in ML-KEM, where the calculation of the final result depends on intermediate values computed in another butterfly.

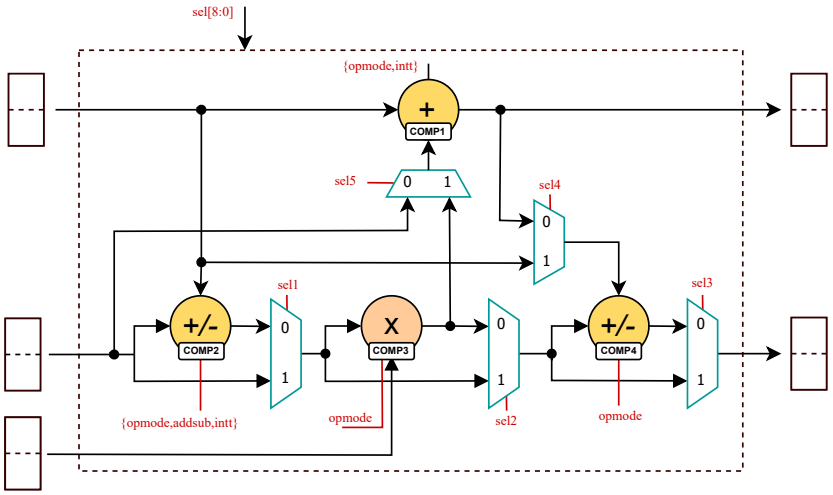


Figure 3: SuperButterfly hardware architecture.

The SBU circuit consists of four fundamental arithmetic units, defined in §4.1.1 and §4.1.2:

COMP1 an **agile modular adder** computing modular addition (with or without carries), followed by an optional modular division by 2 (for a GS butterfly);

COMP2 an **agile modular arithmetic** unit extending COMP1 with modular subtraction;

COMP3 an **agile modular multiplier** supporting carrying and carryless multiplication;

COMP4 a second **agile modular arithmetic** unit like COMP2, but omitting division-by-2.

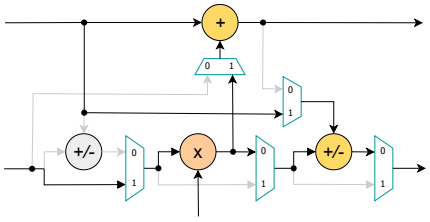
The SBU can be programmed in seven different butterfly configurations: four for NTT multiplication (Figure 4), and three for FAFFT multiplication (Figure 5). We select butterfly configurations using nine control signals:

- `opmode` (`sel[8]`) changes the operation type depending on which polynomial ring we are using;
- `addsub` (`sel[7]`) switches addition and subtraction for NTT butterflies;
- `intt` (`sel[6]`) enables division by 2 in the $\texttt{INTT}_{\text{GS}}$ butterfly;
- `pwm` (`sel[5]`) selects different data arrangements for `PWM0` and `PWM1` in the NTT;
- `sel[4:0]` guarantee correct data flow in a given butterfly.

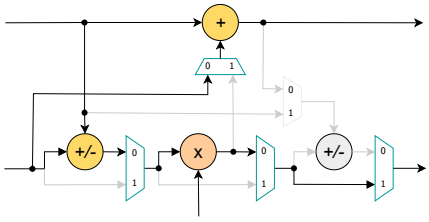
Table 4 lists the configurations with their control signals and active components.

Table 4: Configuration signals for programming SuperButterfly

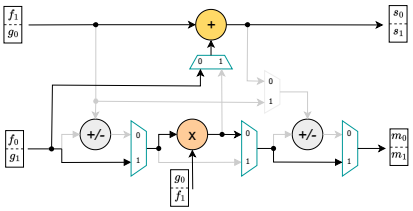
Function	`sel[8:0]`	COMP1	COMP2	COMP3	COMP4
$\texttt{NTT}_{\text{CT}}$	010010011	✓		✓	✓
$\texttt{INTT}_{\text{GS}}$	011000100	✓	✓	✓	
PWM0	000010100	✓		✓	
PWM1	000110000	✓		✓	✓
MOD_ADD	000000000	✓			
$\texttt{FFT}_{\text{LCH}}$	100011001	✓		✓	✓
$\texttt{IFFT}_{\text{LCH}}$	100001101	✓	✓	✓	
$\texttt{PWM}_{\text{LCH}}$	100010100			✓	



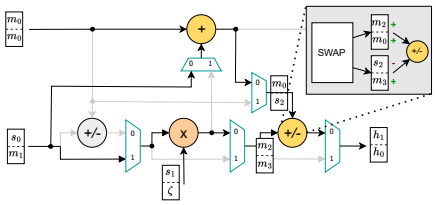
(a) Cooley-Tukey NTT butterfly



(b) Gentlemen-Sande INTT butterfly



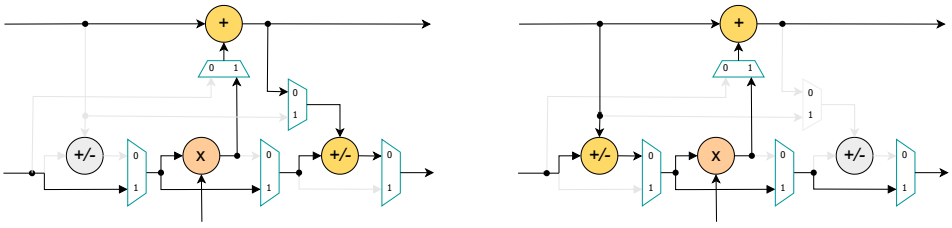
(c) PWM0 configuration



(d) PWM1 configuration

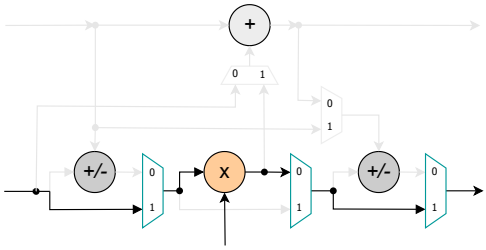
Figure 4: SuperButterfly configuration features for NTT multiplication. CT and GS are used for the NTT and INTT, respectively, while PWM0 and PWM1 which are chained to compute the 2-way Karatsuba PWM.

Remark 1. The SBU can also be used to compute some other ML-KEM operations in hardware beyond NTT multiplication, such as stand-alone modular addition (MOD_ADD in Table 4). This configuration requires only COMP1 and is not used in NTT-based polynomial multiplication, so it is not shown in Figure 4.



(a) FFT LCH butterfly

(b) IFFT LCH butterfly



(c) PWM butterfly

Figure 5: SuperButterfly configuration features for FAFFT multiplication. Note that PWM is much simpler here than in the NTT, both in configuration and at the operational level: only a carryless modular multiplication is used.

4.1.1 Agile modular arithmetic

The FAFFT and NTT use completely different arithmetic operations. The FAFFT uses carryless sums, *i.e.* XORs, and addition and subtraction are identical. The NTT works in $\mathbb{Z}_q$, with values represented by integers in $[0, q)$, and addition and subtraction are distinct. Given representatives a, b for elements of $\mathbb{Z}_q$, the representatives for $(a \pm b) \bmod q$ are

$$c = (a+b) \bmod q = \begin{cases} c_t & \text{if } c_t < q\,, \\ c_t + (\overline{q}+1) & \text{otherwise} \end{cases} \qquad \text{where } c_t := a + b\,; \tag{7}$$

$$d = (a-b) \bmod q = \begin{cases} d_t & \text{if } d_t < q\,, \\ d_t + q & \text{otherwise} \end{cases} \qquad \text{where } d_t := a + (\overline{b}+1)\,. \tag{8}$$

(Here $\overline{x}$ denotes bit-wise NOT, so $\overline{x}+1$ is the two's complement representation of $-x$.)

Figure 6 describes an agile hardware modular adder inspired by the implementation of [ZLL+21]. There are two internal computational units. In the NTT, each computes a separate butterfly; in the FAFFT, with carryless additions, the butterfly result is the concatenation of the two outputs. We can clearly see the two cases of equation (7) in the internal unit: the second requires a sum of three terms. By using a Carry-Save-Adder (CSA), we can perform a carryless addition, as we only need to set a third term to 0.

This circuit can also divide the final result by 2 when needed for GS butterflies in the INTT. We do this with a shift and an addition: $\frac{x}{2} \bmod q = (x \gg 1) + x[0] \cdot \left(\frac{q+1}{2}\right)$.

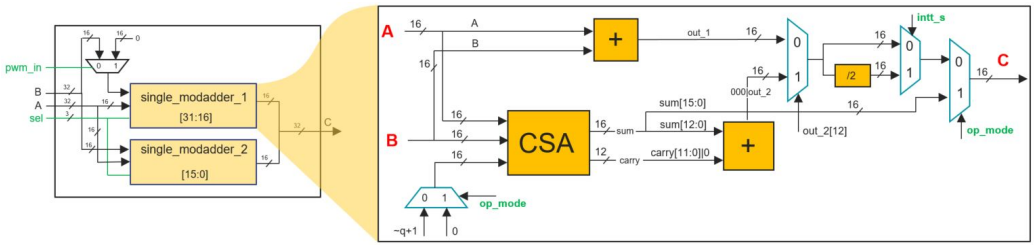


Figure 6: The agile modular adder circuit (COMP1). Configurations and control signals are listed in Table 5.

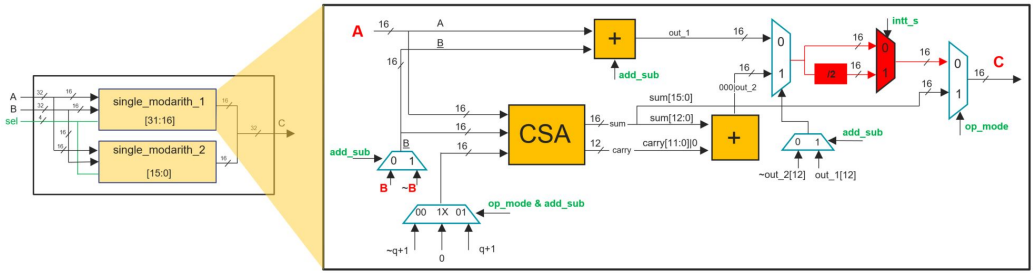


Figure 7: Agile modular arithmetic. The division-by-2 unit, highlighted in red, is included in COMP2 (where it is needed for the GS butterfly), but not in COMP4 (where it is not used). Configurations and control signals are listed in Table 5.

It is easy to modify COMP1 to enable agile subtraction, which is used in COMP2 and COMP4. For the FAFFT this is trivial (addition and subtraction are identical, and computed by an XOR), and for the NTT we use equation (8). Figure 7 shows the resulting circuit; the supported operations and corresponding control signals are listed in Table 5.

Table 5: Configurations and control signals for the agile modular adder and arithmetic units (COMP1, COMP2, and COMP4).

Supported operation	COMP 1	2	4	Control signals `opmode`	`addsub`	`intt`	KEM
Modular addition	✓	✓	✓	0	0	0	ML-KEM
Modular addition + div-by-2	✓	✓		0	0	1	ML-KEM
Modular subtraction		✓	✓	0	1	0	ML-KEM
Modular subtraction + div-by-2		✓		0	1	1	ML-KEM
Carryless addition	✓	✓	✓	1	0	0	HQC

4.1.2 Agile modular multiplier

Our agile modular multiplication unit (COMP3, Figure 8) processes data in two steps:

1. a carryless 32-bit multiplication (with 64-bit output) for the FAFFT, or two 16-bit carrying multiplications (each with 32-bit output) for the NTT, followed by
2. the appropriate modular reduction in each case.

For agility in the first step we use a *hybrid 2-way Karatsuba* architecture, called **agile_multiplier**, which performs two 12-bit carry multiplications for the NTT, or one 32-bit carryless multiplication for the FAFFT.

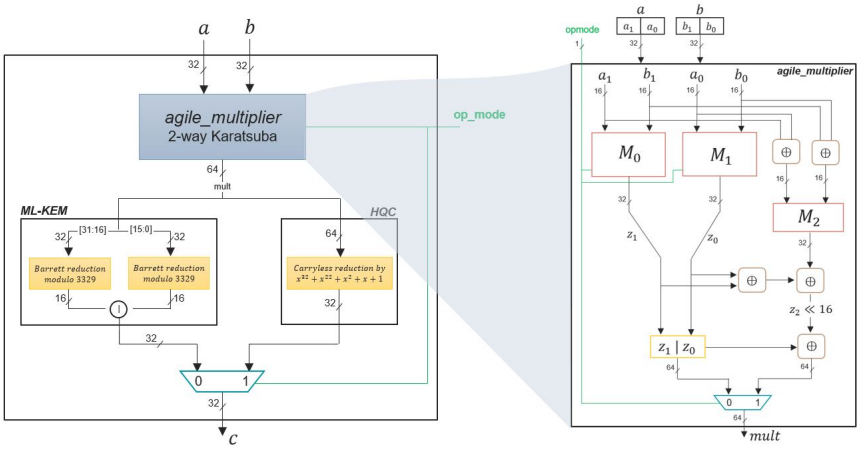


Figure 8: Agile modular multiplier (COMP3)

This implements Algorithm 2, a divide-and-conquer approach following the classic Karatsuba algorithm: given two m-bit inputs, the result is the concatenation of intermediate values computed on $m/2$-bit operands. The same algorithm can also be used for carryless multiplication if all sums and subtractions are replaced by XORs. We will describe this algorithm in detail below.

For agility in the reduction step we simply parallelize the two methods, then select the desired result. This does not represent hardware sharing, but its effective resource consumption is still very low. Modular reduction for the NTT uses the modified Barrett reduction from [XL21]. (This is a more hardware-friendly approach than the Montgomery reduction in the ML-KEM reference implementation, since it can be optimised for $q = 3329$ with shifts and additions.) Modular reduction for the FAFFT uses the same shift-and-add strategy, but with carryless addition: we XOR the 32 least significant bits of the operand and left-shift the corresponding 32 bits of the highest value, for each of the monomials in the modulus $g(x)$, until the 32 most significant bits of the result are equal to 0.

Algorithm 2: Hybrid 2-way Karatsuba Algorithm

Data: Two m-bit numbers a and b, one *opmode* control bit
Result: The product $P = a \cdot b$

Split a into two $m/2$-bit parts: a_1 (high) and a_0 (low)
Split b into two $m/2$-bit parts: b_1 (high) and b_0 (low)
$z_0 \leftarrow a_0 \times_{M_0} b_0$ `// Multiplier` M_0 `supports carry and carryless`
$z_1 \leftarrow a_1 \times_{M_1} b_1$ `// Multiplier` M_1 `supports carry and carryless`
$z_2 \leftarrow \Big((a_0 \oplus a_1) \times_{M_2} (b_0 \oplus b_1)\Big) \oplus z_0 \oplus z_1$ `// Multiplier` M_2`: carryless only`
if *opmode* $= NTT$ **then**
  return $(z_1 \cdot 2^m) \oplus z_0$
else
  return $(z_1 \cdot 2^m) \oplus (z_2 \cdot 2^{m/2}) \oplus z_0$

Now look more closely at Algorithm 2. The multiplier M_2 operates only in carryless mode, so we implement it with a carryless 2-way Karatsuba strategy operating on $m/4$-bit input data. The main innovation, and the main contribution to agility, is in the multipliers M_0 and M_1. Enabling carry propagation determines the results expected by the two butterfly structures integrated in the SuperButterfly, whereas if we disable it, the final

result will be that of a m-bit carryless multiplication.

We implement M_0 and M_1 using an *Array-like Schoolbook* hardware multiplier [AMR23], which offers better performance than the sequential schoolbook approach. This design is characterized by a regular and repetitive structure which makes it easy to scale. The architecture exploits the parallelism of partial products to generate and accumulate them simultaneously, thus calculating the least significant m bits of the result with a critical path of $m/2$ full-adders. The remaining m most significant bits are computed using a traditional Ripple-Carry Adder (RCA) to add the intermediate sum and report values calculated during the accumulation phase. The value of the intermediate partial product accumulation at level i depends on the input carry values, which are output at level $i-1$. If these values are forced to 0 (which we trigger using multiplexers depending on the `opmode` control values), both in the array-like structure and in the RCA, then the result is the partial product accumulation without carry propagation: that is, a carryless multiplication.

4.2 PHOENIX Design Rationale

PHOENIX is a hardware accelerator that uses SUPERBUTTERFLY units to accelerate polynomial multiplication for ML-KEM and HQC with an effective and agile sharing strategy. We give a high-level description in §4.2.1, describe memory organization, data arrangement, and our conflict-free memory strategy in §4.2.2, and discuss constant management in §4.2.3.

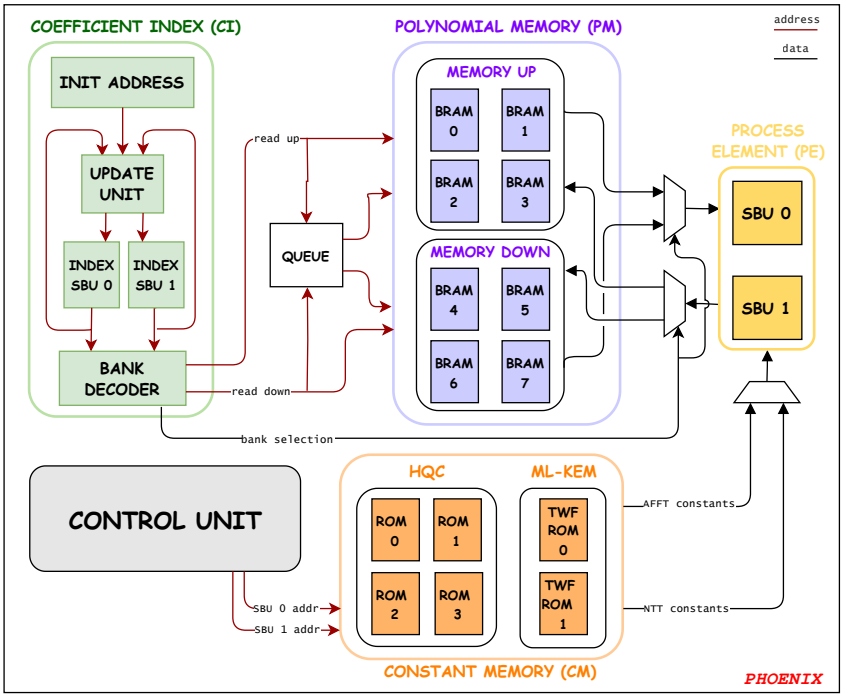


Figure 9: The top-level hardware architecture of PHOENIX. Data and address signals are in black and red, respectively, with control signals omitted.

4.2.1 Overall architecture

Figure 9 shows the PHOENIX architecture. The datapath can be divided into four blocks:

- `Processing element` contains two SuperButterfly units, SBU0 and SBU1. Generally, these work independently. The only case where they must work together is for PWM in the NTT. Then, `Control Unit` configures SBU0 in `PWM0` and SBU1 in `PWM1`, with the output of SBU0 rearranged for input to SB1.
- `Polynomial Memories` divides the memory elements in two parts, *memory-up* and *memory-down*, which store the polynomial coefficients. The size of each part is determined by the maximum number of coefficients in each polynomial, packed into 32-bit words.
- `Constant Memories` stores NTT Twiddle Factors and FAFFT constants.
- `Coefficient Index` computes the index in memory of the coefficients consumed by `Processing element`. `INDEX-SBU 0` and `INDEX-SBU 1` contain registers to store the index-pairs of coefficients to be processed by SBU0 and SBU1, respectively. These are initialized by `INIT-ADDRESS` (according to the polynomial size, depending on the KEM) and incremented by `UPDATE-UNIT`. `BANK-DECODER` computes the memory addresses and instances of coefficients to be processed.

The `Control Unit` orchestrates everything, sending control signals to the other blocks.

Phoenix performs operations *in-place*: the memory addresses used to read coefficients are also used to write results. The operations process different coefficient couples at each clock cycle; given the fixed latency of SuperButterfly, we use delayed data blocks managed by the `QUEUE` element to correctly synchronize `Processing element` results.

4.2.2 Memories and memory access in PHOENIX

Our memory-access scheme uses the methodology of [MRW+22] to address the different levels of parallelism in butterfly executions throughout FFT-like operations. From now on, we refer to a single memory instance as a memory bank, or BRAM. For a given array processing element of width w_{PE} and depth d_{PE}, we use $b := 2w_{PE}$ banks to store polynomial coefficients. The scheduling algorithm of [MRW+22] generates indexes of coefficients to be consumed by the $w_{PE} \times d_{PE}$ processing element units. The authors of [MRW+22] formally prove that this scheme guarantees

- **conflict-free access**: the $2w_{PE}$ coefficients consumed in the `Processing element` unit are read from $2w_{PE}$ different banks in all FFT-like operations. This avoids simultaneously reading two or more coefficients from a single bank, though it also induces an overhead in the bank decoder: more coefficients implies more additions (and thus more resources) for memory-bank calculations.
- **no data-dependency**: the coefficient scheduling algorithm ensures the absence of (read-after-write) data-dependency if $N \leq bc_{PE}$, where N is the number of input coefficients and c_{PE} is the latency for a butterfly from coefficient index generation until storage in memory. For example, if $d_{PE} = 1$, then the coefficient at `index` has

$$\texttt{bank} = \left(\sum_{i=0}^{L-1} d_i\right) \bmod b \qquad \text{and} \qquad \texttt{address} = \left\lfloor \frac{\texttt{index}}{b} \right\rfloor \tag{9}$$

  where $d_0, \ldots, d_{L-1}$ are the digits of the base-b expansion of `index`.

In our case there are only two SBUs in `Processing element`, so $w_{PE} = 2$ and $b = 4$.

`Polynomial Memories` contains 8 Dual-Port BRAMs divided in two sets of four, one for each polynomial. Each BRAM has 1024 32-bit data words to handle the maximum polynomial size. For the FAFFT, one coefficient fits well in a word; for the NTT, the corresponding pair of even and odd index coefficients share a BRAM address. We detail the `Constant Memories` block below.

4.2.3 NTT constant storage and FAFFT constant generation

Constants in embedded accelerators are typically precalculated and stored in a read-only memory. The NTT and FAFFT use different constant values, so they cannot be shared in common memory spaces. We therefore divide the `Constant Memories` block in two sections, one for each set of constants.

For the NTT, the powers of the twiddle factors are often related and grouped within a single address, but data must be replicated when dealing with many PE instances in `Processing element` units. The incomplete NTT used in ML-KEM guarantees processing at least two butterflies per round. We take advantage of this to use two read-only memories (one for each SBU), each storing 256 pre-computed 16-bit twiddle factors (128 for $\texttt{NTT}_{\text{CT}}$ and 128 for $\texttt{INTT}_{\text{GS}}$).

The FAFFT requires a very high number of constants, so we precompute some (stored in four different read-only memories with a size of 16 x 32 bits) and recompute on the fly the others. Indeed, as we saw above, the FAFFT uses fast evaluation of expressions $s_k(\alpha) = \beta_{m/2+i} + s_k(\gamma)$ where $1 \leq i \leq s$ and γ ranges over all the points in V_s. For HQC192 and HQC256 we have $s = 12$, which means 4096 32-bit evaluation points; HQC128 has $s = 11$. Storing all these points requires 32KB of memory, which is inconvenient for embedded applications.

Instead, we store a subset of V_{12}, and recompute the other elements as we require them. The simplest approach would be to store the basis to combine $\beta_0, \ldots, \beta_{11}$ and use a network of XORs to combine them. We reduce overhead using the so-called *four Russians* method (M4R) [ADKF70]. First, we prepare 3 read-only memories $ROM_1, \ldots, ROM_3$ containing all of the elements in $V_4 = \langle \beta_0, \ldots, \beta_3 \rangle$, $\langle \beta_4, \ldots, \beta_7 \rangle$, and $\langle \beta_8, \ldots, \beta_{11} \rangle$, respectively. Splitting the binary vector $\overline{\gamma}$ of $\gamma \in V_{12}$ into 4-bit chunks, i.e. $\overline{\gamma} = \overline{\gamma}_1 + \overline{\gamma}_2 + \overline{\gamma_3}$ with each $\overline{\gamma}_i \in \langle \beta_{4i-4}, \ldots, \beta_{4i-1} \rangle$, we have $\gamma = ROM_1[\overline{\gamma}_1] \oplus ROM_2[\overline{\gamma}_2] \oplus ROM_2[\overline{\gamma}_2]$, using the $\overline{\gamma}_i$ as read memory addresses. To complete the evaluation of $s_k(\alpha)$ in hardware we prepare another read-only memory, ROM_0, containing the Cantor basis values $\beta_{m/2+1}, \ldots, \beta_{m/2+s}$. In total, this requires storing only 60 32-bit points.

5 Implementing PHOENIX

We now describe the integration of Phoenix as a loosely-coupled accelerator in a real System-on-Chip (SoC) scenario, and measure performance results for polynomial multiplication. We will give further detail on integrating Phoenix to accelerate ML-KEM and HQC in §6 below.

We implemented SuperButterfly and Phoenix using Verilog Hardware Description Language (HDL). We simulated, synthetized and implemented them using Vivado 2019.1 design suite on a Xilinx Zynq-7020 SoC (`xc7z020clg400-1`) device containing a dual-core ARM-A9 processor, a 512 MB DDR3L RAM and an Artix-7 based FPGA (13300 Slice, 140 32-Kib BRAM36 and 220 DSP), embedded on Zybo Z7-20 Diligent board. The critical path delay of Phoenix, imposed by the SuperButterfly unit, is 8ns; this corresponds to the maximum frequency of 125MHz. Only one CPU core is used and its clock frequency is set to 125MHz to be close as possible to a final system without clock domain crossing issue. To perform the profiling of the ML-KEM and HQC cryptosystems, we use the PQClean reference[2] [KSSW22] as a baseline software implementation, without using NEON instruction or any assembly optimizations.

[2] https://github.com/PQClean/PQClean/

5.1 PHOENIX integration

We connect PHOENIX to the internal communication bus of our Xilinx Zybo Z7-20 board using an AXI4-Lite hardware slave interface. Although this is not well-suited to high-bandwidth communication compared to the full AXI4 protocol, it offers a lightweight solution for integrating accelerators. The full SoC, depicted in Figure 10, runs at a working frequency of 125MHz.

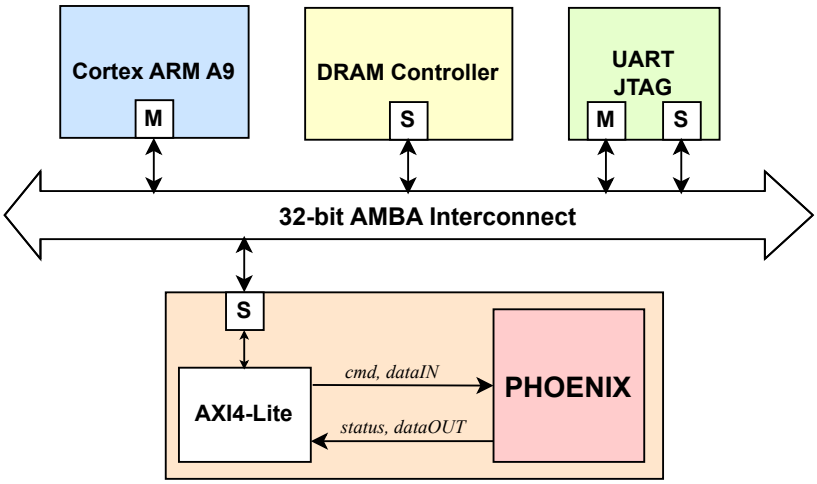


Figure 10: Overall System-on-Chip FPGA integration of PHOENIX.

PHOENIX supports the hardware operations listed in Table 6. The CPU configures PHOENIX for these operations using the instruction set sketched in Figure 11.

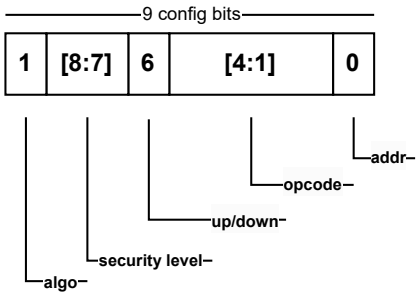


Figure 11: The PHOENIX instruction set. The configuration fields are labelled as follows: `algo` selects the cryptosystem (ML-KEM or HQC); `security level` specifies the security level of the chosen cryptosystem; `u/d` selects operation on *Memory-up*, *Memory-down*, or both; `addr`: defines the pre-compute starting memory address for ML-KEM operations involving polynomial error; and `opcode`: encodes the value of the operation to be executed.

Remark 2. We note that in ML-KEM, all three security levels use the same polynomial size ($n = 256$, variable parameters are the vector and matrix dimensions: k and $k \times k$), so the execution of a single `NTT`, `INTT`, or `PWM` is the same for each level (see §6.1). In HQC,

Table 6: Configurations of supported operations by opcode. The vector operations in the last columns support ML-KEM optimizations discussed in §6.1.

Opcode	Operation	Opcode	Operation	Opcode	Vector operation
0100	NTT	0100	$\texttt{FFT}_{\text{LCH}}$	0101	kNTT
0001	INTT	1100	$\texttt{FFT}_{\text{LCH}}$ + LOAD	0011	kPWM
0010	PWM	0001	$\texttt{IFFT}_{\text{LCH}}$	1010	kACCUMULATE
		0010	$\texttt{PWM}_{\text{LCH}}$	1011	kACCUMULATE + LOAD

on the other hand, HQC128 uses one polynomial size while HQC192 and HQC256 use another.

5.2 Resource consumption

Table 7 breaks down resource consumption for our design (post-implementation), including the major submodules described in §4.2.1 and the four components of the SBU seen in §4.1. The most area-consuming module in PHOENIX is the `Processing element` block, containing two SBU units and accounting for more than 65% of the LUTs and 76% of the FFs. Within each SBU, the major contribution is the agile modular multipler (COMP3), accounting for 66% of the total LUTs and 40% of the FFs. The remaining 60% of FFs are needed for data synchronization. Contrary to traditional NTT-based hardware design, our SBUs do not use DSP resources (this will be important for the area comparison in §5.4). COMP4 is the smallest design sub-block in the SBU, due to the absence of the division-by-2 feature.

The `Polynomial Memories` block contains 8 BRAM, 4 in each sub-block. For the `Constant Memories` block, the two TWF ROM memory banks for ML-KEM require $2 \times 256 \times 16$ bits, and are mapped into a half BRAM instance. The HQC constant memory banks only require 16×32-bits ROMs each; these are not mapped into BRAMs, but rather implemented using LUTs and FFs. The 15% of FFs of PHOENIX are used for data and address synchronization because of the internal latency in the SBU units. Finally, the `Control Unit` accounts for 16% of LUTs and 33 internal FSM states. This non-negligible percentage shows the complexity of this block in managing a variety of operations on polynomials of different lengths.

The lack of existing FAFFT hardware implementations prevents us from measuring the real advantage in resource consumption of the integrated sharing strategy in PHOENIX. Our new main objectives will be to demonstrate the performance alignment, as well as the overhead due to agility, of our accelerator compared with state-of-the-art FAFFT and NTT implementations.

5.3 FAFFT performance with PHOENIX

In this section, we analyse the impact of PHOENIX on a single polynomial multiplication, at all HQC security levels, using the FAFFT. Since there is no previous hardware acceleration of FAFFT to compare for comparison, we compare PHOENIX against our own software-only FAFFT implementation running on the same target platform (described above), operating at 125MHz. Table 8 shows the improvement in a single FAFFT polynomial multiplication.

As we can see, the operational steps targeted for the acceleration are $\texttt{FFT}_{\text{LCH}}$, $\texttt{PWM}_{\text{LCH}}$ and $\texttt{FFT}_{\text{LCH}}$. On these steps, PHOENIX speeds up performance by two orders of magnitude. *Other* represents clock cycles during a whole polynomial multiplication dedicated to operations that are not reported in Table 8. This cost is higher in HW-SW implementations due to memory transfer between the software and PHOENIX. In the two case studies reported, the overall polynomial multiplication is accelerated by *3.3*× and *3.5*× respectively.

Table 7: PHOENIX hardware resource counts for our Artix-7 implementation. `Datapath` includes the given components but also the internal data-selection hardware, performed by multiplexer-based components, that are not explicitly counted elsewhere.

Module	**LUT**	**FF**	**BRAM**	**DSP**
`Control Unit`	826	131	0	0
`Datapath`	4250	3952	8.5	0
⎸`Coeff memory address`	238	128	0	0
⎸`Write address`	30	612	0	0
⎸`Polynomial Memories`	0	0	8	0
⎸`Constant Memories`	37	98	0.5	0
⎸`Processing element`	3319	3114	0	0
⎸1× SUPERBUTTERFLY	1432	1380	0	0
⎸⎸`COMP1`	160	0	0	0
⎸⎸`COMP2`	176	0	0	0
⎸⎸`COMP3`	948	548	0	0
⎸⎸`COMP4`	148	0	0	0
Total = `Control Unit` + `Datapath`	**5076**	**4083**	**8.5**	**0**

The difference between the speedup of the single operations in FAFFT and the total polynomial multiplication is mostly due to the contribution of the software encoding and decoding operations of the polynomials: polynomial basis conversions are relatively inexpensive, because they only use XOR operations.

Table 8: Performance impact of PHOENIX (Cycles, with speedup in italics) for a single FAFFT polynomial operation. Recall that HQC-128 multiplies polynomials of length $n < 2^{15}$, while HQC-192 and HQC-256 multiply polynomials of $n < 2^{16}$. SW refers to our own software-only implementation; HW-SW uses PHOENIX.

Function	**HQC-128**			**HQC-192/HQC-256**		
	SW	HW-SW		SW	HW-SW	
2× `BasisCvt`	109,940	109,940	=	235,096	235,096	=
2× `Encode`	553,766	553,766	=	1,110,258	1,110,258	=
2× $\texttt{FFT}_{\text{LCH}}$	3,055,142	11,520	*256*×	6,651,156	25,316	*262*×
1× $\texttt{PWM}_{\text{LCH}}$	252,073	1,102	*228*×	508,071	2,258	*225*×
1× $\texttt{IFFT}_{\text{LCH}}$	1,522,278	5,760	*264*×	3,308,442	12,658	*261*×
1× `Decode`	764,880	764,880	=	1,532,064	1,532,064	=
1× `iBasisCvt`	98,576	98,576	=	207,426	207,426	=
Other	1,029	332,353		9,900	673,096	
Total	**6,357,684**	**1,877,897**	***3.3***×	**13,562,413**	**3,798,172**	***3.5***×

5.4 NTT performance with PHOENIX

We now describe the impact of PHOENIX on performance and resources for a single NTT operation. Unlike the AFFT, there have been many hardware implementations of the NTT operation for ML-KEM. We decided to compare PHOENIX with implementations offering either low resource consumption [IUH22] [LTHW22] [BNAMK21], or the best overall cryptosystem performance [YMÖS21]. We note that all these implementations integrate 4 butterflies in their process element unit, which corresponds to our 2 SBUs.

To compare resources, we use the *Slice Equivalent Cost* (SEC) metric [LTHW22], which collects and weights various hardware resource costs in a single value. For our Artix-7

FPGA platform,

$$\text{SEC} := \lfloor \text{LUT}/4 \rfloor + \lfloor \text{FF}/8 \rfloor + 200 \cdot \text{BRAM} + 100 \cdot \text{DSP}\,.$$

We also use the *Area-Time Product* (ATP) metric, which multiplies a design's area (in this case SEC) and latency, which is the cycle count divided by the maximum frequency.

Table 9: Area and performance of PHOENIX on the Artix-7 FPGA and comparison with state-of-the-art solutions on the same FPGA. Reported execution time is referred to the execution the NTT and INTT operations.

						Time			ATP	
	LUT	**FF**	**BRAM**	**DSP**	**SEC**	Cycles	MHz	μs	Value	Ratio
This work	5076	4083	8.5	0	3479	236	125	1.88	6569	*1*
[YMÖS21]	2543	792	9	4	2935	232	182	1.27	3741	*÷1.8*
[IUH22]	904	811	2.5	4	1227	268	216	1.24	1522	*÷4.3*
[LTHW22]	1170	1164	2	4	1638	235	303	0.78	1270	*÷5.2*
[BNAMK21]	801	717	2	4	1090	324	222	1.46	1590	*÷4.1*

The results are shown in Table 9. As expected, compared with dedicated hardware NTT accelerators, hardware-agility implies a higher consumption of resources. For example: the high number of BRAMs in Phoenix is imposed by the need to compute with long HQC polynomials. The agility overhead of Phoenix is lower compared with [YMÖS21], due to the smaller difference in SEC: while Phoenix has 2× the LUTs and 5×5 the FFs, it uses the same number of BRAMs and benefits from the absence of DSPs. The overhead of agility is worse when comparing Phoenix with lower-resource approaches, such as [IUH22][BNAMK21][LTHW22], with ATP overheads between 4.1× and 5.2×. It should be noted that this overhead is due to a singular feature of our work. Our hardware can be used to accelerate ML-KEM and HQC.

Reassuringly, the time-overhead of agility is mild. Phoenix executes NTT and INTT operations with a number of cycles that is consistent with the current state of the art. However, as the lowest-frequency solution, Phoenix has the longest total execution time. For PWM, due to the cascading configuration of the two SBUs during the execution of this operation, the latency is 152 clocks (128 for processing and 24 for pipeline latency).

Performances for ML-KEM `KeyGen`, `Encaps` and `Decaps` are reported in Section 6.3.

6 An Agile SoC for ML-KEM and HQC with PHOENIX

We now look at the impact of Phoenix on KEM performance for ML-KEM and HQC. We optimize ML-KEM integration using the internal resources of our design in §6.1. We show how to improve HQC performance using an alternative FAFFT-domain data representation in §6.1.2. Section 6.2 gives results for HQC, providing a complete overview between the pure-software and the hardware-software co-design scenarios. Additionally, we describe their comparison using the optimization proposed in §6.1.2. We give results for ML-KEM in §6.3, showing the speedup with respect to the reference version. Furthermore we analyze the Phoenix positioning in the co-design and fully in the hardware state of the art ecosystem, highlighting the impact of our proposed agile solution.

6.1 Optimizing KEM integrations

Integrating Phoenix in HQC is straightforward. As we saw in §3.4.3, the FFT is a monolithic operation divided into a sequence of subroutines. Polynomials A and B are converted to the new polynomial basis and encoded, then offloaded to the accelerator to speed up the subsequent $\texttt{FFT}_{\text{LCH}}$, $\texttt{PWM}_{\text{LCH}}$, and $\texttt{IFFT}_{\text{LCH}}$. Phoenix can simultaneously perform an

$\texttt{FFT}_{\text{LCH}}$ and an offload to reduce the latency of the entire polynomial multiplication, but these two operations must be performed separately between the *memory-up* (MU) and *memory-down* (MD) blocks in order to be parallelized. To do this, we offload the first polynomial A into MU, perform $\texttt{FFT}_{\text{LCH}}$ on A and, at the same time, we offload the second polynomial B into MD. After the second $\texttt{FFT}_{\text{LCH}}$ operation on B, $\texttt{PWM}_{\text{LCH}}$ is done and the result, stored in MU, is interpolated back with $\texttt{IFFT}_{\text{LCH}}$. Finally, the output is transferred from MU to the processor's main memory, where it is decoded and converted back to the original polynomial basis.

Integration in ML-KEM is different, for two reasons: first, keys are stored in the NTT domain, and second, we must multiply matrices and vectors of polynomials. This does not favour a monolithic hardware-based operation integration, so the overall acceleration factor is heavily influenced by continuous interoperability and data exchange between hardware and software. We mitigate this with vector operations for Phoenix in §6.1.1. (For HQC, we optimise at the level of data representation instead, without adding new hardware operations to Phoenix.)

6.1.1 Array-Array Polynomial Multiplication in ML-KEM

Given two k-vectors $\hat{\mathbf{a}}_0 = (\hat{a}_{00}, \ldots, \hat{a}_{0k-1})$ and $\mathbf{y} = (y_0, \ldots, y_{k-1})$ of polynomials, with the elements of $\hat{\mathbf{a}}_0$ in the NTT-domain (and k depending on the ML-KEM security level), we can compute $t_0 = \mathbf{a}_0 \cdot \mathbf{y}$ as

$$t_0 = \texttt{NTT}^{-1}(\hat{\mathbf{a}}_0 \cdot \hat{\mathbf{y}}) \tag{10}$$

which first computes the polynomial vector $\hat{\mathbf{y}} = NTT(\mathbf{y})$, then the scalar product $\hat{t}_0 = \hat{\mathbf{a}}_0 \cdot \hat{\mathbf{y}}$ using PWM, before interpolating it back in the original polynomial domain to get the result t_0. In fact, this scalar product is the sum of all the PWM operations between corresponding elements of the input vectors:

$$\hat{t}_0 = \hat{\mathbf{a}}_0 \cdot \hat{\mathbf{y}} = \sum\nolimits_{i=0}^{k-1} \hat{a}_{0i} \times_{PWM} \hat{y}_i \tag{11}$$

This sum corresponds to the *accumulation* phase of all the temporary PWM results.

In a traditional NTT integration approach, the accumulation phase is performed in software with PWM in hardware. In this scenario, an NTT accelerator receives the polynomial pair to be multiplied and stores back the result, with this process repeated k times.

We can do better in Phoenix by exploiting the large available amount of memory (imposed by the long polynomials in HQC) to perform polynomial vector multiplication. To this end, we add three further operations in the `Control Unit` of our accelerator: `kNTT`, `kPWM` and `kACCUMULATE`, together with some support functions at the hardware-software interface level to read and write data structures such as polynomials and polynomial vectors. The opcodes for these "vector" operations appeared in Table 6 above.

Given two polynomial vectors already offloaded in the Phoenix memories, we can use these vector instructions to compute entirely in hardware at each ML-KEM security level (equation (11)). For example, given a matrix $\hat{\mathbf{A}}$ and a polynomial k-vector $\hat{\mathbf{y}}$, the matrix-vector product $\mathbf{t} = \hat{\mathbf{A}}\mathbf{y}$ can be computed using k vector-vector polynomial multiplications:

$$\hat{\mathbf{t}} = \hat{\mathbf{A}}\hat{\mathbf{y}} = \sum_{j=0}^{k-1} \sum_{i=0}^{k-1} \hat{a}_{ji} \times_{PWM} \hat{y}_i \,.$$

6.1.2 Alternative data representation for HQC

We can optimize the use of AFFT in HQC by changing the data representation of the private and public keys to the FAFFT domain. (This is compatible with any HQC

implementation using the FAFFT, and in particular with PHOENIX, where it requires no new instructions.) This alternative data representation doubles the public key size, but the efficiency benefits should still make this approach interesting for applications where public keys are rarely transmitted, e.g. with static or pre-shared public keys.

We define a procedure $\texttt{FAFFT}_f(\cdot)$ that converts to `novelpoly`, and encodes and evaluates an input polynomial using the new evaluation point sets imposed by Frobenius map. The inverse procedure $\texttt{FAFFT}_f^{-1}(\cdot)$ interpolates back, decodes, and converts back to the original basis. Recall that in HQC, the secret key is (x, y) and the public key is $pk = (h, s = x + h \cdot y)$. If we precompute $pk = (\ddot{h} = \texttt{FAFFT}_f(h), \ddot{s} = \texttt{FAFFT}_f(s))$, then we can save two $\texttt{FAFFT}_f(\cdot)$s in the *Encrypt* function in the underlying PKE scheme:

$$u = r_1 + \texttt{FAFFT}_f^{-1}(\ddot{h} \times \texttt{FAFFT}_f(r_2)) \,, \tag{12}$$

$$v = mG + \texttt{FAFFT}_f^{-1}(\ddot{s} \times \texttt{FAFFT}_f(r_2)) + e \,. \tag{13}$$

Precomputing $\ddot{y} = \texttt{FAFFT}_f(y)$ saves a further $\texttt{FAFFT}_f(\cdot)$ in the *Decrypt* function:

$$m = \mathit{Decode}(v - \texttt{FAFFT}_f^{-1}(\texttt{FAFFT}_f(u) \times \ddot{y})) \,. \tag{14}$$

6.2 HQC with PHOENIX

Table 10 shows the integration results for HQC-128, HQC-192, and HQC-256 using FAFFT for polynomial multiplication. We use the PQClean reference as a baseline software implementation, as there is no state of the art that includes hardware implementations on the FAFFT applied to HQC. We consider "standard" HQC (compatible with the NIST Round-4 submission specification), and "optimized HQC" proposed in §6.1.2 (with keys in the AFFT domain). Since the "optimized" version targets applications with long-term keys, we only report times for `Encaps` and `Decaps`.

For standard HQC, using the FAFFT in software already gives a substantial improvement: for example, `Encaps` for HQC-128, -192, and -256 runs 3.8, 5.2, and 8.2 times faster, respectively. Integrating PHOENIX brings further improvements: looking at `Encaps` again, the HW-SW codesign runs HQC-128, -192, and -256 9.0, 12.6, and 16.7 times faster, respectively. For "optimized" HQC, we can see further improvements in the performance of `Encaps` and `Decaps`, both in the software and with PHOENIX.

Table 10: Comparison of KEM performance, in number of clock cycles: PQClean reference implementation (SW) vs. this work (SW-based FAFFT multiplication, and HW-SW using PHOENIX). The *optimized* version of HQC is defined in §6.1.2. Values are in MCycles, with speedups over baseline implementation in italics.

		Standard HQC					*Optimized* HQC			
		Ref.	**FAFFT**		**PHOENIX**		**FAFFT**		**PHOENIX**	
HQC-128	`KeyGen`	28.9	7.3	*3.9*×	2.8	*10.3*×	-	-	-	-
	`Encaps`	58.7	15.5	*3.8*×	6.5	*9.0*×	9.6	*6.1*×	5.1	*11.5*×
	`Decaps`	91.9	27.0	*3.4*×	13.6	*6.8*×	20.6	*4.5*×	13.1	*7.0*×
HQC-192	`KeyGen`	85.1	15.8	*5.4*×	6.0	*14.2*×	-	-	-	-
	`Encaps`	171.9	33.3	*5.2*×	13.7	*12.6*×	20.7	*8.3*×	10.9	*15.8*×
	`Decaps`	262.4	54.4	*4.8*×	25.2	*10.4*×	39.9	*6.6*×	23.7	*11.1*×
HQC-256	`KeyGen`	155.8	17.7	*8.8*×	7.9	*19.7*×	-	-	-	-
	`Encaps`	314.6	38.4	*8.2*×	18.8	*16.7*×	25.5	*12.3*×	15.6	*20.2*×
	`Decaps`	483.4	69.1	*7.0*×	39.8	*12.1*×	52.4	*9.2*×	36.1	*13.4*×

We emphasize that all of these results include significant overhead from the slow data exchange between the CPU and our loosely-coupled accelerator, with long polynomials pushed through an AXI-4 Lite interface that does not offer high-performance data

transmission. A better communication interface should further improve the results in Table 10.

6.3 ML-KEM with PHOENIX

Table 11 presents our results for PHOENIX integration in ML-KEM. In order to compare the obtained results, we selected, from the current state-of-the-art, two significant hardware-software co-design solutions: [FSS20] and [WZZ+24].

It is worth pointing out that a direct comparison of cycle counts for the three implementations is not fair: each was run on a different architecture and board. In particular, both of the other system-on-chip solutions are based on the RISC-V microprocessor architecture; there were no public ML-KEM implementations targeting our development board (i.e., based on the ARM architecture). For this reason, Table 11 shows not only raw cycle counts, but also the acceleration of each HW-SW solution with respect to its own SW baseline. This offers a fairer comparison of the acceleration capability of each solution.

Table 11: State-of-the-art comparison for NTT acceleration in ML-KEM HW-SW codesign implementations. Clock cycles are reported for HW-SW, with speedups over the respective SW baselines in italics.

		Device	KeyGen	Encaps	Decaps
ML-KEM 512	[FSS20]	PULPino	939,932 ×*1.21*	1,223,887 ×*1.26*	1,051,003 ×*1.45*
	This Work	Zybo-Z7-20	564,189 ×*1.14*	570,438 ×*1.30*	639,403 ×*1.38*
	[WZZ+24]	PolarFire	326,983 ×*1.91*	415,496 ×*2.01*	394,661 ×*2.42*
ML-KEM 768	[FSS20]	PULPino	1,768,400 ×*1.19*	2,138,810 ×*1.23*	1,889,930 ×*1.36*
	This Work	Zybo-Z7-20	916,828 ×*1.15*	954,992 ×*1.25*	1,048,867 ×*1.33*
	[WZZ+24]	PolarFire	536,213 ×*1.92*	671,082 ×*1.98*	639,024 ×*2.32*
ML-KEM 1024	[FSS20]	PULPino	2,856,302 ×*1.18*	3,312,957 ×*1.21*	2,989,896 ×*1.32*
	This Work	Zybo-Z7-20	1,450,164 ×*1.14*	1,492,047 ×*1.22*	1,614,779 ×*1.28*
	[WZZ+24]	PolarFire	844,008 ×*1.89*	1,015,251 ×*1.91*	972,598 ×*2.18*

PHOENIX accelerates different ML-KEM functions by different factors: `Encaps` improves more than `KeyGen`, and `Decaps` improves even more. While the vector-based optimisations in §6.1 allow a better acceleration of polynomial multiplication with respect to state-of-the-art NTT accelerators, this is not immediately evident from Table 11, due to the higher communication latency imposed by the slow bus protocol on our board. Proof of this is provided by the slightly greater speedup for `Decaps`. In particular, the `PKE.Decrypt` function (see 2.1) is essentially one vector-based polynomial multiplication, which is precisely the operation targeted by our accelerator, while the `PKE.KeyGen` and `PKE.Encrypt` functions can be seen as a series of vector-based multiplications with data transfer between each. Even considering these limitations, the acceleration offered by PHOENIX is similar (and even slightly better in some cases) to that of the *tightly-coupled*[3] solution in [FSS20].

When comparing our proposal to [WZZ+24], we see that PHOENIX shows a lower acceleration factor. For context, we note that the authors of [WZZ+24] propose assembly-based optimizations for the Keccak functions coupled with the NTT hardware acceleration. This explains some of the difference with our solution.

7 Conclusion and Further works

In this paper we introduce PHOENIX, a crypto-agile loosely-coupled hardware accelerator for ML-KEM and HQC, aiming to demonstrate performance improvements and resource

[3] *Tightly-coupled* hardware accelerators operate close to the CPU, thus offering reduced data transfer between processor and hardware unit. This allows a high-performance solution, suitable for embedded systems with resource contraints, like RISC-V-based designs.

savings compared with non-agile approaches to the individual cryptosystems. At its heart is a new configurable hardware design, SuperButterfly, providing an efficient hardware sharing strategy between completely different mathematical structures to speed FFT-based polynomial multiplication operations in ML-KEM and HQC, namely the NTT and the FAFFT.

The use of the FAFFT in HQC is innovative: our software implementation is the first of its kind. Not only does it improve performance, even in software, but it is also the key to agility. (It also permits an interesting optimization where keys are stored and transmitted in the FAFFT domain; since this trades faster speeds for larger keys, it is better-suited to applications with static or pre-shared public keys.)

We have shown that integrating Phoenix yields a significant overall speedup of HQC, and ML-KEM performance close to the state-of-the-art. Comparing Phoenix's resource consumption with efficient NTT-focused hardware solutions, we see that agility (i.e., simultaneously supporting HQC) inevitably adds an overhead of between 1.8 and 5.2×, depending on the reference implementation. For HQC, the lack of competing state-of-the-art hardware FAFFT implementations targeting HQC makes it hard to fully assess the real impact of Phoenix in this arena, but we can still show strong performance improvements—and there is also value in being the first of its kind.

Phoenix could also accelerate polynomial multiplication in other lattice- or code-based cryptosystems, thus offering even more general *lattice-code* crypto-agility. For example, Phoenix could be used in ML-DSA [Nat24a], or any other cryptosystem that uses the NTT, by slightly modifying the internal structure of our SuperButterfly. Indeed, the flexibility design of SuperButterfly could be extended to add other butterfly structures, thus accelerating another FFT-based bottleneck operation contained in HQC's Reed–Muller decoder: the *Fast Hadamard Transform*.

Acknowledgment

This work was supported by the French National Agency within the framework of the "France 2030" program (ANR-10-AIRT-05) and it is also financially supported by the Defense Innovation Agency (AID) from the French Ministry of the Armed Forces.

References

[ABPS24] Francesco Antognazza, Alessandro Barenghi, Gerardo Pelosi, and Ruggero Susella. A high efficiency hardware design for the post-quantum KEM HQC. In *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 431–441. IEEE, 2024.

[ADKF70] V.L. Arlazov, Y.A. Dinitz, M.A. Kronrod, and I.A. Faradzhev. On economical construction of the transitive closure of an oriented graph. *Dokl. Akad. Nauk SSR*, 194:487–488, 1970.

[AG25] Ciadoux P Cooper D Dang Q Dang T Kelsey J Lichtinger J Liu YK Miller C Moody D Peralta R Perlner R Robinson A Silberg H Smith-Tone D Waller Alagic G, Bros M. (2025) status report on the fourth round of the nist post-quantum cryptography standardization process. 2025. https://doi.org/10.6028/NIST.IR.8545.

[AMAB+24] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuville, Arnaud Dion, Philippe Gaborit, Jérôme Lacan, Edoardo Persichetti, Jean-Marc Robert, Pascal Véron, and Gilles Zémor. HQC (Hamming Quasi-Cyclic), 2024.

[AMDD+22] Carlos Aguilar-Melchor, Jean-Christophe Deneuville, Arnaud Dion, James Howe, Romain Malmain, Vincent Migliore, Mamuri Nawan, and Kashif Nawaz. Towards automating cryptographic hardware implementations: a case study of HQC. In *Code-Based Cryptography Workshop*, pages 62–76. Springer, 2022.

[AMI+22] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. KaLi: A crystal for post-quantum security using Kyber and Dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(2):747–758, 2022.

[AMR23] Harish Prasad Allam, Suraj Mandal, and Debapriya Basu Roy. A comparative analysis between karatsuba, toom-cook and ntt multiplier for polynomial multiplication in ntru on fpga. In *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6. IEEE, 2023.

[ASW+23] Nouri Alnahawi, Nicolai Schmitt, Alexander Wiesmaier, Andreas Heinemann, and Tobias Grasmeyer. On the state of crypto-agility. Cryptology ePrint Archive, Paper 2023/487, 2023.

[BC14] Daniel J. Bernstein and Tung Chou. Faster binary-field multiplication and faster binary-field MACs. In *International Conference on Selected Areas in Cryptography*, pages 92–111. Springer, 2014.

[BCC+25] Elaine Barker, Lily Chen, David Cooper, Dustin Moody, Andrew Regenscheid, Murugiah Souppaya, William Newhouse, Russ Housley, and Sean Turner. Considerations for achieving cryptographic agility: Strategies and practices (draft). Technical report, US Department of Commerce, 2025.

[BNAMK21] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. High-speed NTT-based polynomial multiplication accelerator for post-quantum cryptography. In *2021 IEEE 28th symposium on computer arithmetic (ARITH)*, pages 94–101. IEEE, 2021.

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *arXiv preprint arXiv:1910.07557*, 2019.

[Can89] David G Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2):285–300, 1989.

[CCK+17] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Faster multiplication for long binary polynomials. *arXiv preprint arXiv:1708.09746*, 2017.

[CCK+18] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Multiplying boolean polynomials with Frobenius partitions in additive Fast Fourier Transform. *arXiv preprint arXiv:1803.11301*, 2018.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the Intel Haswell and ARM Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–124, 2021.

[CLRS22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[DMSS24] Stefano Di Matteo, Ivan Sarno, and Sergio Saponara. CRYPHTOR: A memory-unified NTT-based hardware accelerator for post-quantum CRYSTALS algorithms. *IEEE Access*, 12:25501–25511, 2024.

[DRA+24] Moody D., Perlner R., Regenscheid A., Robinson A., and Cooper D. Transition to post-quantum cryptography standards. Technical Report NIST Internal Report (IR) NIST IR 8547, National Institute of Standards and Technology, Gaithersburg, MD, 2024. https://doi.org/10.6028/NIST.IR.8547.ipd.

[DXN+23] Sanjay Deshpande, Chuanqi Xu, Mamuri Nawan, Kashif Nawaz, and Jakub Szefer. Fast and efficient hardware implementation of HQC. In *International Conference on Selected Areas in Cryptography*, pages 297–321. Springer, 2023.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 537–554, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 239–280, 2020.

[GM10] Shuhong Gao and Todd Mateer. Additive Fast Fourier Transforms over finite fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, 2010.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki–Okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017.

[HTB+24] Pengzhou He, Yazheng Tu, Tianyou Bao, Çetin Çetin Koç, and Jiafeng Xie. HSPA: High-throughput sparse polynomial multiplication for code-based post-quantum cryptography. *ACM Transactions on Embedded Computing Systems*, 2024.

[HTX23] Pengzhou He, Yazheng Tu, and Jiafeng Xie. LOCS: Low-latency and constant-timing implementation of fixed-weight sampler for HQC. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2023.

[IUH22] Yuma Itabashi, Rei Ueno, and Naofumi Homma. Efficient modular polynomial multiplier for NTT accelerator of Crystals-Kyber. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 528–533. IEEE, 2022.

[KMK+24] ByungJun Kim, Han-Gyeol Mun, Shinwoong Kim, JongMin Lee, and Jaeyoon Sim. A 1.03 MOPS/W lattice-based post-quantum cryptography processor for IoT devices. *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE*, 24(1):55–61, 2024.

[KML+24] ByungJun Kim, Han-Gyeol Mun, Kyeong-Jun Lee, Jeong-Min Woo, Kiseo Kang, Hyunwoo Son, and Jae-Yoon Sim. A 243 KOPS/W crypto-processor supporting homomorphic encryption and post-quantum cryptography for IoT devices. In *2024 IEEE European Solid-State Electronics Research Conference (ESSERC)*, pages 468–471. IEEE, 2024.

[KMMM23] Kristjane Koleci, Paolo Mazzetti, Maurizio Martina, and Guido Masera. A flexible NTT-based multiplier for post-quantum cryptography. *IEEE Access*, 11:3338–3351, 2023.

[KSFS24] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. Post-quantum signatures on RISC-V with hardware acceleration. *ACM Transactions on Embedded Computing Systems*, 23(2):1–23, 2024.

[KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

[LANH16] Sian-Jheng Lin, Tareq Y Al-Naffouri, and Yunghsiang S Han. FFT algorithm for binary extension finite fields and its application to Reed–Solomon codes. *IEEE Transactions on Information Theory*, 62(10):5343–5358, 2016.

[LCK+18] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive Fast Fourier Transform. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, pages 263–270, 2018.

[LGH+24] Daniel Loebenberger, Stefan-Lukas Gazdag, Daniel Herzinger, Eduard Hirsch, and Christian Näther. Formalizing the cryptographic migration problem. *CoRR*, abs/2408.05997, 2024.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[LST+23] Chen Li, Suwen Song, Jing Tian, Zhongfeng Wang, and Çetin Kaya Koç. An efficient hardware design for fast implementation of HQC. In *2023 IEEE 36th International System-on-Chip Conference (SOCC)*, pages 1–6. IEEE, 2023.

[LTHW22] Minghao Li, Jing Tian, Xiao Hu, and Zhongfeng Wang. Reconfigurable and high-efficiency polynomial multiplication accelerator for Crystals-Kyber. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(8):2540–2551, 2022.

[LZL+24] Jiahao Lu, Jiaming Zhang, Zhixiang Luo, Aobo Li, Tianze Huang, Dongsheng Liu, and Chi Cheng. An efficient and configurable hardware architecture of polynomial modular operation for CRYSTALS-Kyber and Dilithium. In *2024 IEEE 67th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 29–32. IEEE, 2024.

[MBB+23] Konstantina Miteloudi, Joppe W Bos, Olivier Bronchain, Björn Fay, and Joost Renes. PQ.V.ALU.E: Post-quantum RISC-V custom ALU extensions on Dilithium and Kyber. In *International Conference on Smart Card Research and Advanced Applications*, pages 190–209. Springer, 2023.

[MRW+22] Jianan Mu, Yi Ren, Wen Wang, Yizhong Hu, Shuai Chen, Chip-Hong Chang, Junfeng Fan, Jing Ye, Yuan Cao, Huawei Li, et al. Scalable and conflict-free NTT hardware accelerator design: Methodology, proof, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(5):1504–1517, 2022.

[Nat15] National Institute of Standards and Technology. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 202, August 4, 2015, U.S. Department of Commerce, Washington, D.C., 2015.

[Nat24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 204, August 13, 2024, U.S. Department of Commerce, Washington, D.C., 2024.

[Nat24b] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 203, August 13, 2024, U.S. Department of Commerce, Washington, D.C., 2024.

[NKDNPH24] Trong-Hung Nguyen, Binh Kieu-Do-Nguyen, Cong-Kha Pham, and Trong-Thuc Hoang. High-speed NTT accelerator for CRYSTAL-Kyber and CRYSTAL-Dilithium. *IEEE Access*, 2024.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[SFW23] Maximilian Schöffel, Johannes Feldmann, and Norbert Wehn. Efficient hardware implementation of constant time sampling for HQC. *arXiv preprint arXiv:2309.16493*, 2023.

[SKN+24] Jonas Schupp, Patrick Karl, Jens Nöpel, Alexander Hepp, Tim Music, and Georg Sigl. RISC-V triplet: Tapeouts for security applications. In *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pages 1–6. IEEE, 2024.

[THCX24] Yazheng Tu, Pengzhou He, Chip-Hong Chang, and Jiafeng Xie. LTE: Lightweight and time-efficient hardware encoder for post-quantum scheme HQC. *IEEE Computer Architecture Letters*, 2024.

[THKX23] Yazheng Tu, Pengzhou He, Çetin Kaya Koç, and Jiafeng Xie. LEAP: Lightweight and efficient accelerator for sparse polynomial multiplication of HQC. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(6):892–896, 2023.

[vDHL17] Joris van Der Hoeven and Robin Larrieu. The Frobenius FFT. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, pages 437–444, 2017.

[VZGG03] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2003.

[WZZ+24] Tengfei Wang, Chi Zhang, Xiaolin Zhang, Dawu Gu, and Pei Cao. Optimized hardware-software co-design for Kyber and Dilithium on RISC-V SoC FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):99–135, 2024.

[XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 328–356, 2021.

[YMÖS21] Ferhat Yaman, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER PQC scheme. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1020–1025. IEEE, 2021.

[YSZ+24] Zewen Ye, Ruibing Song, Hao Zhang, Donglong Chen, Ray Chak-Chung Cheung, and Kejie Huang. A highly-efficient lattice-based post-quantum cryptography processor for IoT applications. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):130–153, 2024.

[ZLL+21] Cong Zhang, Dongsheng Liu, Xingjie Liu, Xuecheng Zou, Guangda Niu, Bo Liu, and Quming Jiang. Towards efficient hardware implementation of NTT for Kyber on FPGAs. In *2021 IEEE international symposium on circuits and systems (ISCAS)*, pages 1–5. IEEE, 2021.

[ZZL+22] Yihong Zhu, Wenping Zhu, Chongyang Li, Min Zhu, Chenchen Deng, Chen Chen, Shuying Yin, Shouyi Yin, Shaojun Wei, and Leibo Liu. RePQC: A 3.4-uJ/Op 48-kOPS post-quantum crypto-processor for multiple-mathematical problems. *IEEE Journal of Solid-State Circuits*, 58(1):124–140, 2022.

[ZZO+24] Yihong Zhu, Wenping Zhu, Yi Ouyang, Junwen Sun, Qi Zhao, Min Zhu, Jinjiang Yang, Chen Chen, Qichao Tao, Hanning Wang, et al. PQPU: A 4.4-μJ/Op 69.4-kOPS agile post-quantum crypto-processor across multiple mathematical problems. *IEEE Journal of Solid-State Circuits*, 2024.