Accelerating NTRU+ Key Generation via Hierarchical Batch Inversion

Jonghyun Kim, Haehyun Cho, and Jong Hwan Park

***Abstract*—In KEM-based TLS 1.3 key establishment, the client generates a fresh KEM key pair for each connection, placing key generation on the handshake critical path. For NTRU+, a KEM based on the NTRU problem selected in the Korean Post-Quantum Cryptography (KpqC) competition, the dominant cost in this path is the polynomial inversion needed to compute the public key. Although NTRU+ uses an NTT-friendly ring and performs this inversion in the NTT domain, the routine still decomposes into many base inversions, each requiring a modular inversion computed by exponentiation.**

To accelerate polynomial inversion in the NTT domain, we collect the modular inversions arising from base inversions into a single stage. This makes it possible to apply Montgomery's trick, reducing the number of modular inversions to one at the cost of sequential product and recovery chains. These chains limit instruction-level parallelism (ILP). To address this dependency bottleneck, we apply hierarchical batching to these exposed denominator inversions, splitting the inputs into k groups to expose independent product chains and recursively batching the resulting k group-product inversions. This preserves the arithmetic cost of Montgomery's trick while improving ILP, thereby reducing cycle counts.

We evaluate hierarchical batch inversion across all NTRU+ parameter sets in both C and AVX2. For NTRU+864, the parameter set with the largest gains, compared with non-batched polynomial inversion, it reduces polynomial inversion latency by 48.91% in C and 59.57% in AVX2. For key generation, the corresponding speedups are 18.91% in C and 9.34% in AVX2.

I. Introduction

SHOR's quantum algorithm [27] showed that a sufficiently large quantum computer would break widely deployed public key cryptosystems such as RSA and elliptic-curve cryptography. This threat has driven the transition to post-quantum cryptography, including the NIST PQC standardization process and the Korean Post-Quantum Cryptography (KpqC) competition. In 2024, NIST published its first PQC standards, including ML-KEM [21], ML-DSA [22], and SLH-DSA [23], while the KpqC competition selected final algorithms in 2025, including NTRU+ [16], SMAUG-T [29], AIMer [1], and HAETAE [11]. These KpqC finalists are being pursued as Korean Industrial Standards (KS). As they move toward standardization and deployment, efficient implementations of these algorithms become increasingly important.

Jonghyun Kim is with the School of Cybersecurity, Korea University, Seoul, Republic of Korea (e-mail: yoswuk@korea.ac.kr).

Haehyun Cho is with the School of Software, Soongsil University, Seoul, Republic of Korea (e-mail: haehyun@ssu.ac.kr).

Jong Hwan Park is with the Department of Computer Science, Sangmyung University, Seoul, Republic of Korea (e-mail: jhpark@smu.ac.kr).

Among these KpqC finalists, NTRU+ [15] is a key encapsulation mechanism (KEM) building on the original NTRU cryptosystem [12]. As in related schemes based on the NTRU problem [9], [17], [31], its public key computation requires polynomial inversion f^{-1} to compute $h = gf^{-1}$, making it the dominant cost in key generation. Traditionally, key generation cost was considered less critical because key pairs could be generated in advance and reused. For example, RSA key transport in TLS 1.2 [8] relied on the server's long-term RSA key and therefore did not require fresh key pair generation during the handshake. However, in KEM-based TLS 1.3 key establishment [7], the client sends a fresh KEM public key in its KeyShare. Fresh key generation is therefore placed on the handshake critical path, and its cost can directly affect connection setup latency, especially when many handshakes are processed in parallel.

Unlike the original NTRU [12], NTRU+ is defined over a Number Theoretic Transform (NTT)-friendly ring also used in NTTRU [18]. This ring allows NTRU+ to benefit from arithmetic optimizations for lattice-based cryptography [2], [13], [26]. In particular, the NTT replaces arithmetic in the original quotient ring with componentwise arithmetic in low-degree rings, reducing the cost of the basic arithmetic operations. Under the generalized CRT induced by the NTT, $R_q \cong \prod_{i=0}^{n/d-1} \mathbb{Z}_q[x]/\langle x^d - \zeta_i \rangle$, the polynomial inversion computation poly_baseinv reduces inversion over R_q to $\ell = n/d$ independent inversions over smaller component rings, each denoted by baseinv. However, each baseinv produces a denominator δ that must be inverted by fqinv in $\mathbb{Z}_q$, yielding ℓ such calls in total. Since each fqinv computes $\delta^{-1} = \delta^{q-2}$ via Fermat's little theorem, these repeated calls remain a major bottleneck in NTRU+ key generation.

To mitigate this bottleneck, the NTRU+ team introduced two optimizations during the KpqC competition [16]. First, each fqinv was optimized by reusing intermediate powers, inspired by fast modular inversion in Curve25519 [3], rather than using a generic square-and-multiply chain. Second, the degree d of the component rings in the NTT decomposition was increased from 2 to 3, and further to 4 in the final $n \in \{768, 1152\}$ settings. For $n = 1152$, the number of baseinv instances, and hence fqinv calls, decreased from $\ell = 576$ to 384 and then to 288, thereby improving the performance of poly_baseinv. However, this second optimization cannot be pushed indefinitely. Increasing d reduces the number of component rings but raises the cost of each component inversion, eventually offsetting the benefit of fewer fqinv calls.

Even after these two optimizations, one poly_baseinv call still needs the inverses of its ℓ denominators. These denominators are a natural target for Montgomery's trick [20], which computes a batch of modular inverses using one modular inversion and multiplications. Montgomery's trick, denoted here by batch_fqinv, uses product_chain to multiply the denominators into one product, a single fqinv call to invert this product, and recover_inv to combine the product inverse with partial products and recover the individual inverses. This eliminates the repeated fqinv calls inside poly_baseinv, but it is still limited by the sequential dependencies in product_chain and recover_inv, which restrict instruction-level parallelism (ILP).

A. Our Contributions

To accelerate NTRU+ key generation, we make the following contributions to polynomial inversion in the NTT domain.

- For polynomial inversion in the NTT domain, we apply Montgomery's trick to the denominator inversions, reducing the number of fqinv calls from ℓ to one.
- We further optimize Montgomery's trick with hierarchical batching, which reorganizes its computation to expose more instruction-level parallelism and reduce latency without increasing the number of modular inversions.
- Across all NTRU+ parameter sets, our technique reduces polynomial inversion latency by 38.18–48.91% in C and 46.31–59.57% in AVX2 relative to non-batched polynomial inversion, yielding key generation latency reductions of 12.62–18.91% in C and 6.47–9.34% in AVX2.

B. Our Technique

Our technique realizes these contributions through two layers. The first layer exposes the denominator inversions inside poly_baseinv. In a conventional implementation, each baseinv computes and immediately inverts its own denominator. In contrast, we split baseinv at this inversion point into baseinv_1, which computes the denominator, and baseinv_2, which consumes its inverse. Based on this split, we reorganize poly_baseinv so that all ℓ baseinv_1 calls are performed before inversion, collecting the denominators into one batch. The mult_fqinv stage then computes the denominator inverses over $\mathbb{Z}_q$, and the corresponding baseinv_2 calls complete the component inversions.

The second layer instantiates mult_fqinv through a two-step restructuring of the direct application of Montgomery's trick, addressing the ILP limitation identified above. First, we define batch_fqinv_kway, a k-way variant of batch_fqinv that partitions the ℓ denominators into k groups and exposes independent product and recovery chains, but requires one fqinv call per group product. We then extend this to batch_fqinv_hier by recursively computing the inverses of these k products in one batch, bringing the inversion count back to one while retaining these independent chains. Fig. 1 compares the direct, k-way, and hierarchical dependency structures. Concretely, batch_fqinv_hier preserves the $3\ell - 3$ modular multiplications and single fqinv call of batch_fqinv, while retaining the independent product and recovery chains created by the k-way split. Section V shows that this dependency structure translates to cycle savings across both the C and AVX2 backends.

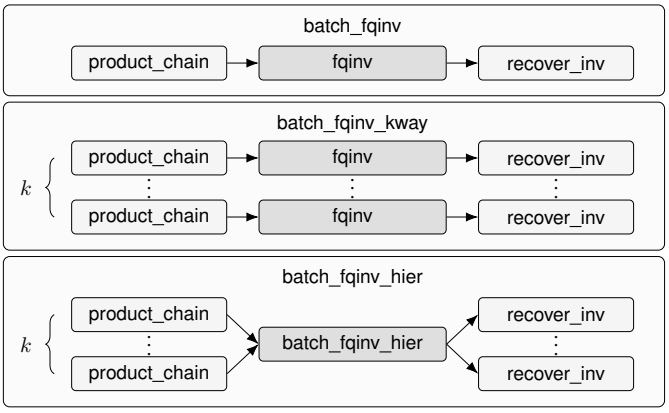

Fig. 1. Dependency structure of the three batch inversion variants. batch_fqinv (top) threads all ℓ inputs through one product_chain, a single fqinv call, and one recover_inv. batch_fqinv_kway (middle) partitions the batch into k groups, thereby creating independent chains that expose ILP, but uses k fqinv calls. batch_fqinv_hier (bottom) keeps these chains and recursively applies the same batch inversion to the k group products, requiring only a single fqinv call.

C. Related Work

Polynomial inversion has been central to NTRU-based cryptography since the original NTRU [12], appearing in both KEMs [4], [25], [32] and digital signatures [24], [30]. Early implementations used variants of the extended Euclidean algorithm (EEA), including the Almost-Inverse method [28], with input-dependent divisions and quotient updates. Later constant-time approaches [6], [14] replace this sequence with exponentiation or a fixed iteration schedule.

Other work moves polynomial inversion to the NTT domain. Under the NTT-induced decomposition, polynomial inversion is reduced to inversions in degree-d component rings. For $d = 1$, these component inversions are modular inversions in $\mathbb{Z}_q$, handled by EEA in NTRUEncrypt [32] and by Fermat exponentiation in Falcon [24]. For $d > 1$, NTTRU [18] treats $d = 3$, an earlier NTRU+ [15] treats $d = 2$ analogously, and BAT [10] uses recursive reductions when d is composite.

While the works above change how each polynomial inversion is computed, another direction accelerates polynomial inversion by batching inversions across independent instances. OpenSSLNTRU [5] applies Montgomery's trick [20] to polynomial ring inversions across 32 Streamlined NTRU Prime key generations. Such across-keygen batching is useful when many key pairs are generated together and stored for later use, but it does not apply to single-handshake settings such as KEM-based TLS 1.3. In contrast, we batch denominators within one poly_baseinv call for a single NTRU+ key generation. To the best of our knowledge, Montgomery's trick has not been applied to denominator inversions within a single NTT domain polynomial inversion.

A separate line of work improves batch inversion itself by changing the dependency structure of Montgomery's trick. Mishra [19] replaces Montgomery's sequential chain with a complete binary tree schedule to exploit available parallel multipliers. Our hierarchical batch inversion uses a similar tree structure, but keeps the split parameter k tunable to control the number of independent product and recovery chains, rather than fixing a binary schedule.

D. Paper Organization

Section II recalls the NTT domain decomposition and key generation of NTRU+, NTT domain inversion, and Montgomery's trick. Section III isolates the denominator stage in component ring inversion and lifts this staged form to poly_baseinv. Section IV develops the k-way and hierarchical batch inversion variants used for mult_fqinv. Section V evaluates the C and AVX2 implementations at the mult_fqinv, poly_baseinv, and key generation levels, and reports the component degree analysis. Section VI concludes the paper.

II. Preliminaries

A. NTT-Domain Decomposition in NTRU+

NTRU+ is defined over the NTT-friendly quotient ring

$$R_q = \mathbb{Z}_q[x]/\langle x^n - x^{n/2} + 1\rangle, \qquad n = 2^a 3^{b-1}\ (a, b \in \mathbb{N}).$$

If $\mathbb{Z}_q$ contains a primitive $(3n/d)$-th root of unity ζ for a divisor d of n, the generalized CRT gives

$$\mathsf{NTT} : R_q \to \prod_{i=0}^{n/d-1} \mathbb{Z}_q[x]/\langle x^d - \zeta_i\rangle,$$

where each $\zeta_i = \zeta^j$ for some $j \equiv \pm 1 \pmod 6$. We denote the forward and inverse transforms by poly_ntt and poly_invntt, respectively. For the resulting $\ell = n/d$ component rings, we denote componentwise multiplication and inversion by poly_basemul and poly_baseinv, respectively. In the final NTRU+ parameter sets considered here [16], $q = 3457$ and $n \in \{768, 864, 1152\}$, with $d = 4$ for $n \in \{768, 1152\}$ and $d = 3$ for $n = 864$.

B. Key Generation in NTRU+

NTRU+ key generation samples $f', g' \sim \psi_1^n$ and forms $f = 3f' + 1$ and $g = 3g'$. It then computes the public key $h = gf^{-1}$ and stores the precomputed inverse $h^{-1} = fg^{-1}$ in the secret key. Thus both f and g must be invertible in R_q, and sampling is repeated until the required inverses exist. Algorithm 1 summarizes the procedure.

Algorithm 1 $\mathsf{NTRU+KeyGen}(1^\lambda)$ (Simplified)

while true **do**
  $f' \leftarrow \psi_1^n$
  $f \leftarrow 3f' + 1$
  $\hat{f} \leftarrow \mathsf{poly_ntt}(f)$
  $(b, \hat{f}^{-1}) \leftarrow \mathsf{poly_baseinv}(\hat{f})$
  if $b = 1$, **break**
while true **do**
  $g' \leftarrow \psi_1^n$
  $g \leftarrow 3g'$
  $\hat{g} \leftarrow \mathsf{poly_ntt}(g)$
  $(b, \hat{g}^{-1}) \leftarrow \mathsf{poly_baseinv}(\hat{g})$
  if $b = 1$, **break**
$\hat{h} \leftarrow \mathsf{poly_basemul}(\hat{g}, \hat{f}^{-1})$
$\hat{h}^{-1} \leftarrow \mathsf{poly_basemul}(\hat{f}, \hat{g}^{-1})$
$pk \leftarrow \hat{h}$
$sk \leftarrow (\hat{f},\ \hat{h}^{-1},\ \mathsf{F}(pk))$
return (pk, sk)

C. Inversion in the NTT Domain

Under the NTT, inversion in R_q reduces to n/d independent component ring inversions in $\mathbb{Z}_q[x]/\langle x^d - \zeta_i\rangle$. We handle these recursively as in BAT [10]. For $d = km$ with $k \in \{2, 3\}$, one reduction step moves the remaining denominator to $\mathbb{Z}_q[y]/\langle y^m - \zeta_i\rangle$ with $y = x^k$. The recursion ends at $d = 1$, where the operation is modular inversion in $\mathbb{Z}_q$. Thus the NTRU+ cases $d = 3$ and $d = 4$ use one $d = 3m$ step and two $d = 2m$ steps, respectively.

1) Case $d = 1$: $\mathbb{Z}_q[x]/\langle x - \zeta_i\rangle$ is isomorphic to $\mathbb{Z}_q$, so the base case is modular inversion. For $q = 3457$, fqinv (Algorithm 2) computes $a^{-1} = a^{3455}$ by Fermat's little theorem using 11 squarings and 5 multiplications, rather than the 11 squarings and 9 multiplications of naive square-and-multiply. A single fqinv call nevertheless remains roughly $16\times$ as expensive as a single modular multiplication.

Algorithm 2 $\mathsf{fqinv}(a \in \mathbb{Z}_q)$ for $q = 3457$

$t_1 \leftarrow a^2$
$t_2 \leftarrow (t_1)^4$ // a^8
$t_3 \leftarrow (t_2)^2$ // a^{16}
$t_1 \leftarrow t_1 \cdot t_2$ // a^{10}
$t_2 \leftarrow t_1 \cdot t_3$ // a^{26}
$t_2 \leftarrow (t_2)^2 \cdot a$ // a^{53}
$t_1 \leftarrow t_1 \cdot t_2$ // a^{63}
$t_2 \leftarrow (t_2)^{2^6} \cdot t_1$ // a^{3455}
return t_2

2) Case $d = 2m$: With $y = x^2$ and $S_i = \mathbb{Z}_q[y]/\langle y^m - \zeta_i\rangle$, the even/odd split of BAT [10] gives

$$\mathbb{Z}_q[x]/\langle x^{2m} - \zeta_i\rangle \cong S_i[x]/\langle x^2 - y\rangle.$$

Write the image of $a(x) = \sum_{j=0}^{2m-1} a_j x^j$ as $\bar{a}(x) = \bar{a}_0 + \bar{a}_1 x$, where

$$\bar{a}_0 = \sum_{j=0}^{m-1} a_{2j} y^j, \qquad \bar{a}_1 = \sum_{j=0}^{m-1} a_{2j+1} y^j.$$

Here $\bar{a}_0, \bar{a}_1 \in S_i$. Then

$$\bar{a}(x)(\bar{a}_0 - \bar{a}_1 x) = \bar{a}_0^2 - \bar{a}_1^2 x^2 = \bar{a}_0^2 - \bar{a}_1^2 y,$$

and hence

$$\bar{a}(x)^{-1} = (\bar{a}_0 - \bar{a}_1 x)(\bar{a}_0^2 - \bar{a}_1^2 y)^{-1}.$$

Thus the remaining inversion is that of $\bar{a}_0^2 - \bar{a}_1^2 y \in S_i$.

3) Case $d = 3m$: With $y = x^3$ and $S_i = \mathbb{Z}_q[y]/\langle y^m - \zeta_i\rangle$, the three-way split of NTTRU [18] gives

$$\mathbb{Z}_q[x]/\langle x^{3m} - \zeta_i\rangle \cong S_i[x]/\langle x^3 - y\rangle.$$

The image of $a(x) = \sum_{j=0}^{3m-1} a_j x^j$ is $\bar{a}(x) = \bar{a}_0(y) + \bar{a}_1(y)x + \bar{a}_2(y)x^2$, where $\bar{a}_i(y) = \sum_{j=0}^{m-1} a_{i+3j} y^j \in S_i$ for $i \in \{0, 1, 2\}$. Define

$$\bar{r}_0 = \bar{a}_0^2 - \bar{a}_1 \bar{a}_2 y, \quad \bar{r}_1 = \bar{a}_2^2 y - \bar{a}_0 \bar{a}_1, \quad \bar{r}_2 = \bar{a}_1^2 - \bar{a}_0 \bar{a}_2,$$

and set $\delta = \bar{a}_0 \bar{r}_0 + (\bar{a}_1 \bar{r}_2 + \bar{a}_2 \bar{r}_1) y \in S_i$. Then

$$\bar{a}(x)^{-1} = \delta^{-1}(\bar{r}_0 + \bar{r}_1 x + \bar{r}_2 x^2).$$

Thus the remaining inversion is that of $\delta \in S_i$.

Algorithm 3 indep_fqinv($\mathbf{a} \in (\mathbb{Z}_q^*)^\ell$)

$\mathbf{b} \leftarrow \mathbf{0} \in \mathbb{Z}_q^\ell$
for $i \leftarrow 0, \ldots, \ell-1$ **do**
  $b_i \leftarrow \mathsf{fqinv}(a_i)$
return $\mathbf{b}$

Algorithm 4 batch_fqinv($\mathbf{a} \in (\mathbb{Z}_q^*)^\ell$)

$\mathbf{p} \leftarrow \mathsf{product_chain}(\mathbf{a})$
$w \leftarrow \mathsf{fqinv}(p_{\ell-1})$ // $p_{\ell-1} = \Delta$
$\mathbf{b} \leftarrow \mathsf{recover_inv}(\mathbf{a}, \mathbf{p}, w)$
return $\mathbf{b}$

Algorithm 5 product_chain($\mathbf{a} \in \mathbb{Z}_q^\ell$)

$\mathbf{p} \leftarrow \mathbf{0} \in \mathbb{Z}_q^\ell$
$p_0 \leftarrow a_0$
for $i \leftarrow 1, \ldots, \ell-1$ **do**
  $p_i \leftarrow p_{i-1} \cdot a_i$
return $\mathbf{p}$

Algorithm 6 recover_inv($\mathbf{a}, \mathbf{p} \in \mathbb{Z}_q^\ell,\ w \in \mathbb{Z}_q$)

$\mathbf{b} \leftarrow \mathbf{0} \in \mathbb{Z}_q^\ell$
for $i \leftarrow \ell-1, \ldots, 1$ **do**
  $b_i \leftarrow p_{i-1} \cdot w$
  $w \leftarrow w \cdot a_i$
$b_0 \leftarrow w$
return $\mathbf{b}$

D. Batch Inversion

Given a vector $\mathbf{a} = (a_0, \ldots, a_{\ell-1}) \in (\mathbb{Z}_q^*)^\ell$, batch inversion returns the entrywise inverses $(a_0^{-1}, \ldots, a_{\ell-1}^{-1}) \in (\mathbb{Z}_q^*)^\ell$. The baseline indep_fqinv (Algorithm 3) applies fqinv separately to each entry. Montgomery's trick [20] instead forms the product $\Delta = \prod_{i=0}^{\ell-1} a_i$, inverts Δ once, and recovers all a_i^{-1} from Δ^{-1} and prefix products. We denote this algorithm by batch_fqinv (Algorithm 4). batch_fqinv uses product_chain (Algorithm 5) to compute prefix products $p_i = a_0 \cdots a_i$ recursively by $p_i = p_{i-1}a_i$, so $p_{\ell-1} = \Delta$. After fqinv computes $w = \Delta^{-1}$, it recovers the inverses using recover_inv (Algorithm 6), which scans backward, maintaining $w = p_i^{-1}$ at iteration i and outputting $a_i^{-1} = p_{i-1}w$ for $i \geq 1$, with $a_0^{-1} = w$ at the end.

product_chain and recover_inv use $\ell-1$ and $2(\ell-1)$ modular multiplications, respectively. Hence batch_fqinv replaces the ℓ fqinv calls of indep_fqinv with one fqinv and $3\ell-3$ modular multiplications. The tradeoff is reduced parallelism. Whereas indep_fqinv exposes the ℓ inversions as independent operations, batch_fqinv threads the work through sequential product_chain and recover_inv chains, limiting ILP.

III. NTT-Domain Inversion for Batch Inversion

For each of the $\ell = n/d$ component rings, baseinv performs one modular inversion in $\mathbb{Z}_q$ on a denominator δ. To expose these ℓ inversions as a batch, we split baseinv into baseinv_1, which produces δ, and baseinv_2, which consumes its inverse $\eta = \delta^{-1}$. We then reorder poly_baseinv so that all denominators are collected and inverted between these two stages.

Algorithm 7 baseinv($\mathbf{a} \in \mathbb{Z}_q^d;\ \zeta \in \mathbb{Z}_q$) for $d \in \{3, 4\}$

$(\mathbf{b}, \delta) \leftarrow \mathsf{baseinv_1}(\mathbf{a}; \zeta)$
if $\delta = 0$ **then**
  return failure
$\eta \leftarrow \mathsf{fqinv}(\delta)$ // $\eta = \delta^{-1}$
$\mathbf{b} \leftarrow \mathsf{baseinv_2}(\mathbf{b}, \eta)$ // scale the numerator vector
return $\mathbf{b}$

Algorithm 8 baseinv_1($\mathbf{a} \in \mathbb{Z}_q^3;\ \zeta \in \mathbb{Z}_q$) for $d = 3$

$b_0 \leftarrow a_1 a_2$
$b_1 \leftarrow a_2^2$
$b_2 \leftarrow a_1^2 - a_0 a_2$
$b_0 \leftarrow a_0^2 - \zeta b_0$
$b_1 \leftarrow \zeta b_1 - a_0 a_1$
$\delta \leftarrow \zeta(b_2 a_1 + b_1 a_2) + b_0 a_0$
return $(\mathbf{b}, \delta)$

Algorithm 9 baseinv_1($\mathbf{a} \in \mathbb{Z}_q^4;\ \zeta \in \mathbb{Z}_q$) for $d = 4$

$t_0 \leftarrow a_2^2 - 2a_1 a_3$
$t_1 \leftarrow a_3^2$
$t_0 \leftarrow a_0^2 + \zeta t_0$
$t_1 \leftarrow a_1^2 + \zeta t_1 - 2a_0 a_2$
$t_2 \leftarrow \zeta t_1$
$\delta \leftarrow t_0^2 - t_1 t_2$
$b_0 \leftarrow a_0 t_0 + a_2 t_2$
$b_1 \leftarrow -(a_3 t_2 + a_1 t_0)$
$b_2 \leftarrow a_2 t_0 + a_0 t_1$
$b_3 \leftarrow -(a_1 t_1 + a_3 t_0)$
return $(\mathbf{b}, \delta)$

Algorithm 10 baseinv_2($\mathbf{a} \in \mathbb{Z}_q^d,\ \eta \in \mathbb{Z}_q$) for $d \in \{3, 4\}$

$\mathbf{b} \leftarrow \mathbf{0} \in \mathbb{Z}_q^d$
for $i \leftarrow 0, \ldots, d-1$ **do**
  $b_i \leftarrow a_i \eta$
return $\mathbf{b}$

A. Component Ring Inversion

Following Section II-C, we instantiate baseinv to invert elements in the component ring $\mathbb{Z}_q[x]/\langle x^d - \zeta_i \rangle$ for $d \in \{3, 4\}$. The $d = 3$ instantiation uses one $d = 3m$ split, while the $d = 4$ instantiation uses two $d = 2m$ splits. To expose the denominator δ, we further factor baseinv (Algorithm 7) into baseinv_1 and baseinv_2. baseinv_1 returns $(\mathbf{b}, \delta)$, with δ the uninverted $d = 1$ denominator. Algorithms 8 and 9 give the $d = 3$ and $d = 4$ cases, respectively. baseinv_2 (Algorithm 10) scales $\mathbf{b}$ by $\eta = \delta^{-1}$ to obtain the component ring inverse.

B. Product Ring Inversion

We instantiate poly_baseinv over the $\ell = n/d$ component rings $\mathbb{Z}_q[x]/\langle x^d - \zeta_i \rangle$. A direct implementation applies baseinv once per component ring. In that form, the denominator δ_i produced for component i is inverted immediately by fqinv and consumed before the next denominator is produced. Hence $\delta_0, \ldots, \delta_{\ell-1}$ never appear together as a vector over $\mathbb{Z}_q$, and one poly_baseinv call exposes no batch inversion target.

Algorithm 11 poly_baseinv($\mathbf{a} \in \mathbb{Z}_q^n$) for $d \in \{3, 4\}$

$\ell \leftarrow n/d$ // number of component rings
Initialize $\mathbf{b} \in (\mathbb{Z}_q^d)^\ell$ and $\boldsymbol{\delta} \in \mathbb{Z}_q^\ell$
for $i \leftarrow 0, \ldots, \ell - 1$ **do**
  $\mathbf{a}^{(i)} \leftarrow (a_{id}, \ldots, a_{(i+1)d-1})$
  $(\mathbf{b}^{(i)}, \delta_i) \leftarrow$ baseinv_1$(\mathbf{a}^{(i)}; \zeta_i)$
  if $\delta_i = 0$ **then**
    return failure
$\boldsymbol{\eta} \leftarrow$ mult_fqinv$(\boldsymbol{\delta})$ // $\eta_i = \delta_i^{-1}$ for $\delta_i \neq 0$
for $i \leftarrow 0, \ldots, \ell - 1$ **do**
  $\mathbf{b}^{(i)} \leftarrow$ baseinv_2$(\mathbf{b}^{(i)}, \eta_i)$
return $\mathbf{b} = (\mathbf{b}^{(0)}, \ldots, \mathbf{b}^{(\ell-1)})$

To expose such a target, Algorithm 11 first computes all denominators and delays their inversion. It applies baseinv_1 to each component ring, collecting $\boldsymbol{\delta} = (\delta_0, \ldots, \delta_{\ell-1})$. It then calls mult_fqinv, the interface for batch inversion, to compute $\boldsymbol{\eta} = (\delta_0^{-1}, \ldots, \delta_{\ell-1}^{-1})$ and finally applies baseinv_2 to component i with η_i for each i. This keeps the same baseinv_1 and baseinv_2 operations, while exposing the ℓ denominator inversions as a single vector input to mult_fqinv. Note that mult_fqinv can be instantiated using the baselines indep_fqinv (Algorithm 3) and batch_fqinv (Algorithm 4) from Section II-D, as well as the batch_fqinv_kway and batch_fqinv_hier variants later introduced in Section IV.

IV. Hierarchical Batch Inversion

In Section III, mult_fqinv exposes the ℓ denominator inversions of poly_baseinv. Instantiating this with batch_fqinv applies Montgomery's trick, reducing the ℓ fqinv calls to one. However, batch_fqinv forms sequential product_chain and recover_inv chains over all ℓ inputs, thereby leaving little ILP to exploit. We address this dependency bottleneck by splitting the computation into k groups and applying batch inversion within each group, while recursively batching the inversion of the k intermediate group products. Section IV-A presents batch_fqinv_kway by applying the k-way split to batch_fqinv, and Section IV-B presents batch_fqinv_hier by extending this construction with recursive batching.

A. k-Way Batch Inversion

The baseline batch_fqinv in Algorithm 4 consists of three stages, product_chain, a single call to fqinv, and recover_inv, where the first and third stages are sequential dependency chains over the ℓ inputs. To replace these long chains by independent ones, batch_fqinv_kway (Algorithm 12) partitions the input into k groups of size $s = \ell/k$ and applies the same three-stage pattern to the groups. product_chain_kway (Algorithm 13) computes within-group prefix products and outputs the k group products. indep_fqinv (Algorithm 3) uses k fqinv calls to invert these group products. recover_inv_kway (Algorithm 14) uses the corresponding group-product inverse to recover the inverses within each group. Thus batch_fqinv_kway exposes k independent product_chain chains and k independent recover_inv chains, but requires k fqinv calls. Theorem 1 gives the cost in modular multiplications and fqinv calls.

Algorithm 12 batch_fqinv_kway($\boldsymbol{\delta} \in \mathbb{Z}_q^\ell$, k)

$(\mathbf{p}, \mathbf{g}) \leftarrow$ product_chain_kway$(\boldsymbol{\delta}, k)$
$\boldsymbol{\tau} \leftarrow$ indep_fqinv$(\mathbf{g})$
$\boldsymbol{\eta} \leftarrow$ recover_inv_kway$(\boldsymbol{\delta}, \mathbf{p}, \boldsymbol{\tau}, k)$
return $\boldsymbol{\eta}$

Algorithm 13 product_chain_kway($\boldsymbol{\delta} \in \mathbb{Z}_q^\ell$, k)

$s \leftarrow \ell/k$
$\mathbf{p} \leftarrow \mathbf{0} \in \mathbb{Z}_q^\ell$
$\mathbf{g} \leftarrow \mathbf{0} \in \mathbb{Z}_q^k$
for $i \leftarrow 0, \ldots, k - 1$ **do**
  $\boldsymbol{\delta}^{(i)} \leftarrow (\delta_{is}, \ldots, \delta_{(i+1)s-1})$
  $\mathbf{p}^{(i)} \leftarrow$ product_chain$(\boldsymbol{\delta}^{(i)})$
  $(p_{is}, \ldots, p_{(i+1)s-1}) \leftarrow \mathbf{p}^{(i)}$
  $g_i \leftarrow p_{(i+1)s-1}$
return $(\mathbf{p}, \mathbf{g})$

Algorithm 14 recover_inv_kway($\boldsymbol{\delta}, \mathbf{p} \in \mathbb{Z}_q^\ell$, $\boldsymbol{\tau} \in \mathbb{Z}_q^k$, k)

$s \leftarrow \ell/k$
$\boldsymbol{\eta} \leftarrow \mathbf{0} \in \mathbb{Z}_q^\ell$
for $i \leftarrow 0, \ldots, k - 1$ **do**
  $\boldsymbol{\delta}^{(i)} \leftarrow (\delta_{is}, \ldots, \delta_{(i+1)s-1})$
  $\mathbf{p}^{(i)} \leftarrow (p_{is}, \ldots, p_{(i+1)s-1})$
  $\boldsymbol{\eta}^{(i)} \leftarrow$ recover_inv$(\boldsymbol{\delta}^{(i)}, \mathbf{p}^{(i)}, \tau_i)$
  $(\eta_{is}, \ldots, \eta_{(i+1)s-1}) \leftarrow \boldsymbol{\eta}^{(i)}$
return $\boldsymbol{\eta}$

Theorem 1: Let ℓ, k be positive integers with $k \mid \ell$, let $s = \ell/k$, and let $\boldsymbol{\delta} = (\delta_0, \ldots, \delta_{\ell-1}) \in (\mathbb{Z}_q^*)^\ell$. On input $(\boldsymbol{\delta}, k)$, Algorithm 12 returns $(\delta_0^{-1}, \ldots, \delta_{\ell-1}^{-1})$ using k fqinv calls and $3\ell - 3k$ modular multiplications.

Proof: Algorithm 12 executes k independent length-s batch_fqinv instances. It groups their product, inversion, and recovery operations into product_chain_kway, indep_fqinv, and recover_inv_kway, respectively. Correctness follows groupwise from batch_fqinv.

For the operation count, each length-s batch_fqinv computation uses one fqinv call and $3(s-1)$ modular multiplications by Section II-D. Summing over the k groups gives k fqinv calls and $3k(s-1) = 3\ell - 3k$ modular multiplications. ■

By Theorem 1, batch_fqinv_kway saves $3(k-1)$ modular multiplications but adds $k-1$ fqinv calls compared with the length-ℓ batch_fqinv, which uses one fqinv call and $3\ell - 3$ modular multiplications. If one fqinv costs c_{inv} modular multiplications, the cost increase over batch_fqinv is

$$(k-1)c_{\text{inv}} - 3(k-1) = (k-1)(c_{\text{inv}} - 3).$$

Thus batch_fqinv_kway reduces modular arithmetic cost only when $c_{\text{inv}} < 3$. For NTRU+, however, fqinv (Algorithm 2) uses 11 squarings and 5 multiplications, giving $c_{\text{inv}} = 16$ when squarings count as multiplications. Any speedup must therefore come from dependency reduction, not from lower modular arithmetic cost. This motivates the recursive batching below, which removes the extra fqinv calls.

Algorithm 15 batch_fqinv_hier($\boldsymbol{\delta} \in \mathbb{Z}_q^{\ell}$, k)

```
(p, g) ← product_chain_kway(δ, k)
if k is divisible by 2 then
    τ ← batch_fqinv_hier(g, k/2)
else if k is divisible by 3 then
    τ ← batch_fqinv_hier(g, k/3)
else
    τ_0 ← fqinv(g_0)
η ← recover_inv_kway(δ, p, τ, k)
return η
```

B. Hierarchical Batch Inversion

In batch_fqinv_kway, product_chain_kway produces the k group products, which form a length-k vector inversion problem. indep_fqinv solves this problem with k separate fqinv calls. To reduce these k fqinv calls, one could apply batch_fqinv directly to this vector, reducing the k inversions to one fqinv call. However, this would reintroduce a sequential chain of length k. batch_fqinv_hier instead applies k-way batch inversion recursively to this smaller vector, avoiding both the extra fqinv calls and the sequential length-k chain. Figure 1 compares this hierarchy with batch_fqinv and batch_fqinv_kway.

The remaining design choice is how to split the vector inversion at each recursive level. Since $\ell = n/d$ with $d \mid n$ and the degrees n considered here have only factors 2 and 3, every nontrivial recursive length can be reduced by one of these factors. In this work, Algorithm 15 uses a fixed rule, choosing factor 2 whenever possible and factor 3 otherwise, mainly for analytical simplicity. Exploring alternative factor priorities is outside our scope. Theorem 2 gives the resulting cost in modular multiplications and fqinv calls.

Theorem 2: Let ℓ be a positive integer with no prime factors other than 2 and 3. Let k be a positive integer with $k \mid \ell$, let $s = \ell/k$, and let $\boldsymbol{\delta} = (\delta_0, \ldots, \delta_{\ell-1}) \in (\mathbb{Z}_q^*)^{\ell}$. On input $(\boldsymbol{\delta}, k)$, Algorithm 15 returns $(\delta_0^{-1}, \ldots, \delta_{\ell-1}^{-1})$ using one fqinv call and $3\ell - 3$ modular multiplications.

Proof: Correctness follows by repeatedly applying the correctness argument for batch_fqinv_kway. At each nonbase level, the recursive call supplies the group-product inverses required by recover_inv_kway. At length 1, fqinv provides the base inverse. Products of nonzero values remain nonzero, so every recursive input is well defined. Hence the outer call returns $(\delta_0^{-1}, \ldots, \delta_{\ell-1}^{-1})$.

For the operation count, let $m_0 = \ell, m_1 = k, \ldots, m_t = 1$ be the successive vector lengths from the outer input to the base case, where t is the final level. At level i, the chosen split reduces a length-m_i vector to m_{i+1} group products, which form the next recursive input. By Theorem 1, this costs $3(m_i - m_{i+1})$ modular multiplications outside the recursive call. Summing over all levels gives

$$\sum_{i=0}^{t-1} 3(m_i - m_{i+1}) = 3(m_0 - m_t) = 3\ell - 3.$$

Thus the total is one fqinv call and $3\ell - 3$ modular multiplications. ∎

Theorem 2 shows that the modular arithmetic cost of batch_fqinv_hier matches that of batch_fqinv. Relative to batch_fqinv_kway, it keeps the same independent outer chains but removes the extra fqinv calls by recursively batching the group products. In this sense, batch_fqinv_hier combines the single fqinv call of batch_fqinv with the shorter dependency chains of batch_fqinv_kway. Since the operation count is unchanged, the remaining question is whether scheduling these shorter chains outweighs the recursion overhead. Section V evaluates whether this dependency structure translates into cycle-count improvements.

V. Experimental Results

This section evaluates hierarchical batch inversion at the mult_fqinv, NTT-domain polynomial inversion, and NTRU+ key generation levels. Section V-A describes the experimental setup, including the benchmark parameters and performance measurement. Section V-B benchmarks the isolated batch inversion interface mult_fqinv. Section V-C measures the effect of these instantiations inside poly_baseinv. Section V-D evaluates the resulting impact on NTRU+ key generation. Section V-E separately sweeps the component ring degree d at $(n, q) = (1152, 3457)$ to check whether increasing d remains beneficial with hierarchical batching.

A. Experimental Setup

To match NTRU+ key generation, the benchmarks cover the three NTRU+ parameter sets NTRU+768, NTRU+864, and NTRU+1152. All three use the same modulus $q = 3457$. Their corresponding (n, d) values are $(768, 4)$, $(864, 3)$, and $(1152, 4)$, respectively. Thus $\ell = n/d$ equals 192 for NTRU+768 and 288 for NTRU+864 and NTRU+1152. These ℓ values are used for the mult_fqinv benchmark, and the same (n, d) values are used for poly_baseinv.

For these parameter settings, we evaluate a C implementation and an AVX2-optimized variant. Since $q = 3457 < 2^{12}$, the AVX2 variant packs 16 field elements per 256-bit register, and fqinv computes 16 lane-wise inversions per call. Both implementations use Montgomery reduction, with AVX2 modular multiplication following Seiler [26]. Lazy reductions are placed per parameter set and implementation according to coefficient bounds and valid reduction input and output ranges.

For inputs accepted by poly_baseinv, our implementations follow fixed operation and memory-access patterns, including the fqinv calls and product/recovery chains used in batch inversion. A poly_baseinv call can return failure early when some denominator δ_i is zero, but the corresponding sample is discarded in NTRU+ key generation rather than retained in the secret key.

The benchmarks were run under Ubuntu 24.04.4 LTS on an Intel Core i7-8700K processor with 16 GB RAM. To stabilize timing, the CPU clock was fixed at 3.7 GHz, Hyper-Threading was disabled, and benchmark runs were pinned to a single core isolated at boot. The code was compiled with clang 18.1.3 using -O3 -fomit-frame-pointer, with -mavx2 added for AVX2. Each cycle count is the median of five run-level averages, each computed over 100,000 iterations. The code is available at https://github.com/yoswuk/hierarchical_batch_inversion.

TABLE I
CYCLE COUNTS FOR mult_fqinv AT $\ell \in \{192, 288\}$ OVER THE SPLIT PARAMETER k. COLUMNS "k-WAY" AND "HIER." DENOTE THE TWO SPLIT VARIANTS. BOLDFACE MARKS COLUMN MINIMA, AND DASHES MARK INAPPLICABLE SPLITS.

k	$\ell = 192$				$\ell = 288$			
	C		AVX2		C		AVX2	
	k-way	Hier.	k-way	Hier.	k-way	Hier.	k-way	Hier.
ℓ	10334	-	1296	-	15510	-	1900	-
1	5103	-	488	-	7575	-	637	-
2	2913	2883	**411**	381	4278	4249	**478**	453
3	2449	2353	466	360	3520	3428	513	409
4	**2273**	2116	528	**350**	3231	3074	-	-
6	2279	1967	687	**350**	**3150**	2838	723	**387**
8	2439	1966	-	-	3304	2830	-	-
9	-	-	-	-	3385	2834	994	392
12	2760	**1959**	-	-	3625	**2821**	-	-
16	3120	1967	-	-	3984	2829	-	-
18	-	-	-	-	4154	2846	-	-
24	3814	2017	-	-	4694	2896	-	-
32	4541	2039	-	-	5453	2927	-	-
36	-	-	-	-	5776	2964	-	-
48	5952	2108	-	-	6913	3019	-	-
64	7344	2228	-	-	-	-	-	-
72	-	-	-	-	9054	3203	-	-
96	10043	2182	-	-	11180	3432	-	-
144	-	-	-	-	14919	3152	-	-

B. mult_fqinv *Performance*

Table I lists isolated mult_fqinv cycle counts at $\ell \in \{192, 288\}$, where ℓ is the number of inputs per call. It compares the k-way and hierarchical variants against indep_fqinv and batch_fqinv. The first column gives the group count k. Rows $k = \ell$ and $k = 1$ correspond to indep_fqinv and batch_fqinv. For $k \geq 2$, "k-way" and "Hier." denote the two variants. The C rows list proper divisors k of ℓ. The AVX2 rows list proper divisors of $\ell/16$ so each group contains complete 16-lane vectors.

Before considering the split variants, we first compare batch_fqinv with indep_fqinv. This comparison tests whether collapsing the ℓ independent fqinv calls to a single fqinv call compensates for the loss of ILP. The benchmark answers this question affirmatively. At $\ell = 192$, batch_fqinv uses 50.62% fewer cycles in C and 62.35% fewer cycles in AVX2 than indep_fqinv. At $\ell = 288$, the corresponding cycle-count reductions are 51.16% in C and 66.47% in AVX2.

For batch_fqinv_kway, cycles first decrease with k as independent product and recovery chains expose ILP. After this ILP gain saturates, the extra fqinv calls make cycles rise despite the multiplication savings in Theorem 1. The minima occur in C at $k = 4$ and $k = 6$ for $\ell = 192$ and $\ell = 288$, respectively, and in AVX2 at $k = 2$ for both lengths. Relative to the unsplit batch_fqinv baseline, these minima reduce cycles in C by 55.46% and 58.42%, and in AVX2 by 15.78% and 24.96%, for $\ell = 192$ and $\ell = 288$, respectively. Relative to indep_fqinv, these points reduce cycles in C by 78.00% and 79.69%, and in AVX2 by 68.29% and 74.84%.

TABLE II
CYCLE COUNTS FOR poly_baseinv ON THE NTRU+ PARAMETER SETS OVER THE SPLIT PARAMETER k. BOLDFACE MARKS COLUMN MINIMA, AND DASHES MARK INAPPLICABLE SPLITS.

k	NTRU+768		NTRU+864		NTRU+1152	
	C	AVX2	C	AVX2	C	AVX2
ℓ	22177	1993	26004	2476	33257	2954
1	16970	1211	18087	1249	25376	1717
2	14749	1105	14820	1066	22020	1520
3	14281	1074	13945	1018	21338	1475
4	14043	**1070**	13589	-	20846	-
6	13813	**1070**	13307	**1001**	20594	**1452**
8	**13709**	-	**13286**	-	**20443**	-
9	-	-	13295	1008	20460	1475
12	13721	-	13325	-	20453	-
16	13715	-	13319	-	20458	-

For batch_fqinv_hier, cycles decrease at small k as the split exposes ILP and then stay near their minima once this gain saturates. Because the inversion count is fixed, the small increases at larger k mainly reflect recursive scheduling overhead. Table I shows that the fine-grained choices $k = \ell/2$ and $k = \ell/3$, related to Mishra's binary schedule [19], are not optimal in C. Instead, the minima occur at $k = 12$ in C for both $\ell = 192$ and $\ell = 288$, and in AVX2 at $k = 4$ and $k = 6$ for $\ell = 192$ and at $k = 6$ for $\ell = 288$. Relative to batch_fqinv, these minima reduce cycles in C by 61.61% and 62.76%, and in AVX2 by 28.28% and 39.25%, for $\ell = 192$ and $\ell = 288$, respectively. Relative to indep_fqinv, the same minima reduce cycles in C by 81.04% and 81.81%, and in AVX2 by 72.99% and 79.63%. Relative to the best batch_fqinv_kway points, they reduce cycles in C by 13.81% and 10.44%, and in AVX2 by 14.84% and 19.04%.

C. poly_baseinv *Performance*

Table II lists poly_baseinv cycle counts for the NTRU+ parameter sets. The table uses indep_fqinv, batch_fqinv, and batch_fqinv_hier as mult_fqinv instantiations. Among split variants, we use batch_fqinv_hier rather than batch_fqinv_kway because the hierarchical variant has lower isolated mult_fqinv minima for both ℓ values in C and AVX2.

In poly_baseinv, batch_fqinv still reduces cycles relative to indep_fqinv, but the reduction is smaller than in the isolated mult_fqinv benchmark because only the mult_fqinv call changes, while baseinv_1 and baseinv_2 are unchanged. For NTRU+768, 864, and 1152, batch_fqinv reduces cycles in C by 23.48%, 30.45%, and 23.70%, and in AVX2 by 39.24%, 49.56%, and 41.88%, respectively. With batch_fqinv_hier, the C minima move from the isolated $k = 12$ to $k = 8$, while the AVX2 minima remain at $k = 4$ or $k = 6$ across all parameter sets. Within poly_baseinv, the best batch_fqinv_hier entries for NTRU+768, 864, and 1152 reduce cycles relative to batch_fqinv in C by 19.22%, 26.54%, and 19.44%, and in AVX2 by 11.64%, 19.86%, and 15.43%, respectively. Relative to indep_fqinv in the same routine, they reduce cycles in C by 38.18%, 48.91%, and 38.53%, and in AVX2 by 46.31%, 59.57%, and 50.85%.

TABLE III
CYCLE COUNTS FOR NTRU+ KEYGEN, ENCAP, AND DECAP. IN THE KEYGEN BLOCK, "INDEP." AND "HIER." USE indep_fqinv AND batch_fqinv_hier INSIDE poly_baseinv. "GAIN" MARKS THE KEYGEN CYCLE REDUCTION.

Setting	C					AVX2				
	KeyGen			Encap	Decap	KeyGen			Encap	Decap
	Indep.	Hier.	Gain			Indep.	Hier.	Gain		
NTRU+768	117346	100424	14.42%	73618	85940	28759	26899	6.47%	30870	18957
NTRU+864	133654	108374	18.91%	85309	100852	32432	29403	9.34%	35732	23683
NTRU+1152	201576	176135	12.62%	112127	134081	47598	44519	6.47%	45869	30347

TABLE IV
CYCLE COUNTS FOR THE COMPONENT RING DEGREE SWEEP AT $n = 1152$ AND $q = 3457$ OVER DEGREES $d \mid n$ UP TO 8. BOLDFACE MARKS COLUMN MINIMA, AND DASHES MARK INAPPLICABLE SPLITS.

Metric	k	C						AVX2					
		$d=1$	$d=2$	$d=3$	$d=4$	$d=6$	$d=8$	$d=1$	$d=2$	$d=3$	$d=4$	$d=6$	$d=8$
poly_ntt	-	23212	20624	16829	16409	14699	13895	1890	1745	1616	1468	1434	1348
poly_invntt	-	27136	24954	20776	21614	16943	16712	1594	1488	1441	1402	1327	1267
poly_basemul	-	4715	9387	11129	12640	15166	17706	351	505	697	919	1327	1571
poly_baseinv	ℓ	63676	39398	34666	33254	34733	35455	7430	4093	3289	2960	3030	3236
	1	31466	23396	24083	25370	29529	31588	2073	1466	1586	1714	2281	2673
	2	18379	16715	19695	22018	27336	29942	1273	1083	1341	1515	2174	-
	3	14666	15083	18518	21219	26777	29523	1039	969	1267	1474	**2155**	**2583**
	4	13275	14362	18048	20846	26532	29352	950	942	1241	-	2159	-
	6	12281	13876	17702	20624	26367	29229	906	916	**1231**	**1452**	**2155**	-
	8	**12194**	**13824**	17680	**20442**	26281	**29217**	**901**	-	1251	-	-	-
	9	**12194**	13833	-	20446	-	29229	990	**914**	-	1473	-	2596
	12	12212	13827	**17659**	20446	**26233**	29225	-	-	-	-	-	-
	16	12249	13845	17666	20456	26296	29243	-	-	-	-	-	-

D. End-to-End NTRU+ Performance

Table III gives cycle counts for NTRU+ KeyGen, Encap, and Decap. In the KeyGen block, "Indep." and "Hier." use indep_fqinv and batch_fqinv_hier inside poly_baseinv. For batch_fqinv_hier, we use $k = 8$ in C and $k = 6$ in AVX2 for all parameter sets. These k values attain the column minima in Table II. Encap and Decap are included to show the KeyGen share of the full NTRU+ cost and to separate the KeyGen improvement from its contribution to the full KEM workload.

KeyGen cycles decrease for every parameter set, in C by 14.42%, 18.91%, and 12.62%, and in AVX2 by 6.47%, 9.34%, and 6.47% for NTRU+768, 864, and 1152, respectively. The reductions are smaller in AVX2 because the vectorized implementation already exploits the parallel structure of NTT-based polynomial arithmetic. As a result, poly_baseinv accounts for a smaller share of the AVX2 KeyGen count than of the C KeyGen count. In both C and AVX2, the largest KeyGen reduction occurs at NTRU+864, which uses $d = 3$. Among the three sets, NTRU+864 has the cheapest baseinv_1 and baseinv_2 stages, so the batched mult_fqinv stage accounts for a larger share of poly_baseinv. Batching therefore removes a larger fraction of the poly_baseinv cost, consistent with the degree sweep in Section V-E. Since KeyGen accounts for a smaller share of the AVX2 total, the end-to-end reductions are also smaller. For NTRU+768, 864, and 1152, the aggregate counts decrease in C by 6.12%, 7.90%, and 5.73%, and in AVX2 by 2.38%, 3.32%, and 2.61%, respectively.

E. Component Ring Degree Sweep

The previous results motivate revisiting the component ring degree tradeoff for poly_baseinv under batch_fqinv_hier. Under indep_fqinv, increasing d raises the aggregate baseinv_1 and baseinv_2 cost but lowers the number of fqinv calls to $\ell = n/d$. Under batch_fqinv_hier, however, increasing d no longer reduces the fqinv count, which is fixed at one per mult_fqinv call regardless of d. We therefore ask whether decreasing d lowers poly_baseinv cycles by reducing the remaining baseinv_1 and baseinv_2 cost.

To answer this question, we sweep over d with $n = 1152$ and $q = 3457$, the NTRU+1152 parameters. This setting includes $d = 1$, so the sweep covers every divisor $d \mid n$ up to 8. Table IV lists the resulting poly_baseinv cycle counts under indep_fqinv, batch_fqinv, and batch_fqinv_hier. It also lists poly_ntt, poly_invntt, and poly_basemul cycle counts, whose costs are also affected by d.

Table IV shows that batch_fqinv_hier changes the best component ring degree for poly_baseinv relative to the indep_fqinv baseline. With indep_fqinv, the minimum occurs at $d = 4$ in both C and AVX2, matching the component ring degree used by the NTRU+1152 parameter set [16]. In contrast, because batch_fqinv_hier fixes the inversion count at one, the minimum shifts to $d = 1$ in both backends. At $d = 1$, each component lies in $\mathbb{Z}_q$, so batching applies directly to the fqinv inputs and avoids the aggregate baseinv_1 and baseinv_2 arithmetic required for $d \geq 2$.

However, $d = 1$ is not optimal for NTRU+ once the other NTT-domain operations are included. Increasing d reduces the number of NTT layers, which lowers the poly_ntt/poly_invntt costs, but it raises the poly_basemul cost by increasing the component size of each multiplication. To find the optimal d for NTRU+, we weight the table entries by how often NTRU+ calls each polynomial operation in KeyGen, Encap, and Decap. For simplicity, we omit the rejection-sampling factor. Summing the calls over one KeyGen, Encap, and Decap gives weights $6 = 2 + 2 + 2$ for poly_ntt, $1 = 0 + 0 + 1$ for poly_invntt, $5 = 2+1+2$ for poly_basemul, and $2 = 2+0+0$ for poly_baseinv, where the three summands are the call counts in KeyGen, Encap, and Decap. Applying these weights to the $n = 1152$ sweep gives a minimum at $d = 3$ in C and at $d = 2$ in AVX2, rather than at $d = 1$. Note that the optimal d can differ across schemes, because each scheme may weight the same operation costs differently.

VI. Conclusion

This paper showed that the repeated fqinv calls that dominate poly_baseinv in NTRU+ key generation can be reduced to one, and that the resulting product and recovery chains can be accelerated by exposing ILP. To do so, we separated denominator generation from denominator inversion in poly_baseinv, collected the $\ell = n/d$ denominators into one $\mathbb{Z}_q$ batch, and applied Montgomery's trick to reduce the ℓ inversions to one. Unlike across-keygen batching, this batch is formed from denominators internal to one polynomial inversion. To the best of our knowledge, this is the first application of Montgomery's trick in this setting. This separation is the key interface change, since the denominators are materialized as a vector before any inverse is consumed. We then instantiated hierarchical batching for this denominator batch using the tree scheduling view of Montgomery's trick [19]. Splitting the batch into k groups creates independent product and recovery chains, while recursive batching removes the introduced group inversions without increasing the $3\ell - 3$ modular multiplication count.

The poly_baseinv and KeyGen measurements show lower cycle counts with hierarchical batching for all three NTRU+ parameter sets, consistent with fewer fqinv calls and exposed ILP. Relative to the indep_fqinv baseline, hierarchical batching reduces poly_baseinv cycles by 38.18% to 48.91% in C and by 46.31% to 59.57% in AVX2, and the corresponding KeyGen reductions range from 12.62% to 18.91% in C and from 6.47% to 9.34% in AVX2. The largest reductions occur at NTRU+864, whose cheaper baseinv_1 and baseinv_2 stages give the batched inversion a larger share of poly_baseinv.

These gains have a notable limitation. Once Montgomery's trick has reduced the denominator batch to one inversion, the hierarchical split does not further reduce the arithmetic count. It still uses one fqinv call and $3\ell-3$ modular multiplications, so its additional benefit comes from exposing ILP in the product and recovery chains. The gain is largest when these chains bottleneck the target backend and diminishes when they are not the limiting factor or when the available ILP is already saturated. This explains why the fastest split is an intermediate k rather than the deepest tree, and why k must be tuned per backend and parameter set.

Beyond the speedups for the fixed NTRU+ parameter sets, the component ring degree sweep shows that hierarchical batching changes the parameter tradeoff. For NTRU+1152, hierarchical batching minimizes poly_baseinv at $d = 1$, whereas the indep_fqinv baseline favors the $d = 4$ used by the parameter set. When the full NTRU+ operation mix is considered, however, the best choice at the scheme level shifts again, to $d = 3$ in C and $d = 2$ in AVX2. The component ring degree d and batching split k should therefore be chosen together in the context of the target scheme, not from the base inversion cost alone.

This scheme dependence points to two directions for future work. First, because the optimal d depends on how a scheme weights each polynomial operation, the same batching strategy is worth evaluating in other NTRU-based schemes such as DAWN [17] and NEV [31]. Second, since the denominator batch is formed within a single poly_baseinv call, batching across independent key generations remains an orthogonal option for settings that generate many key pairs at once.

References

[1] AIMer Team. The AIMer signature scheme. KpqC Competition Final Version Specification, 2026. Available from KpqC Algorithms page.

[2] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR TCHES*, 2022(1):221–244, 2022.

[3] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Berlin, Heidelberg, April 2006.

[4] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. NIST Post-Quantum Cryptography Standardization Round 3 Submission, 2020. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[5] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 845–862. USENIX Association, August 2022.

[6] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR TCHES*, 2019(3):340–398, 2019.

[7] Deirdre Connolly. ML-KEM post-quantum key agreement for TLS 1.3. IETF Internet-Draft draft-ietf-tls-mlkem-07, February 2026. Active Internet-Draft.

[8] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, August 2008.

[9] Julien Duman, Kathrin Hövelmanns, Eike Kiltz, Vadim Lyubashevsky, Gregor Seiler, and Dominique Unruh. A thorough treatment of highly-efficient NTRU instantiations. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 65–94. Springer, Cham, May 2023.

[10] Pierre-Alain Fouque, Paul Kirchner, Thomas Pornin, and Yang Yu. BAT: Small and fast KEM over NTRU lattices. *IACR TCHES*, 2022(2):240–265, 2022.

[11] HAETAE Team. HAETAE: Hyperball bimodal module rejection. KpqC Competition Final Version Specification, 2026. Document modified February 4, 2026.

[12] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In *Third Algorithmic Number Theory Symposium (ANTS)*, volume 1423 of *LNCS*, pages 267–288. Springer, June 1998.

[13] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR TCHES*, 2022(4):614–636, 2022.

[14] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 232–252. Springer, Cham, September 2017.

[15] Jonghyun Kim and Jong Hwan Park. NTRU+: Compact Construction of NTRU Using Simple Encoding Method. *IEEE Transactions on Information Forensics and Security*, 18:4760–4774, 2023.

[16] Jonghyun Kim and Jong Hwan Park. NTRU+: Compact construction of NTRU using simple encoding method. KpqC Competition Final Version Specification, January 2026. Version dated January 30, 2026; available at https://www.kpqc.or.kr/images/pdf2/NTRU+.pdf.

[17] Yijian Liu, Yu Zhang, Xianhui Lu, Yao Cheng, and Yongjian Yin. DAWN: Smaller and faster NTRU encryption via double encoding. In Goichiro Hanaoka and Bo-Yin Yang, editors, *ASIACRYPT 2025, Part III*, volume 16247 of *LNCS*, pages 396–427. Springer, Singapore, December 2025.

[18] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly fast NTRU using NTT. *IACR TCHES*, 2019(3):180–201, 2019.

[19] Pradeep Kumar Mishra. Efficient simultaneous inversion in parallel and application to point multiplication in ECC. In *Information Security and Cryptology (CISC 2005)*, volume 3822 of *LNCS*, pages 324–335. Springer, 2005.

[20] Peter L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[21] National Institute of Standards and Technology. FIPS 203: Module-lattice-based key-encapsulation mechanism standard. Federal Information Processing Standards Publication 203, August 2024.

[22] National Institute of Standards and Technology. FIPS 204: Module-lattice-based digital signature standard. Federal Information Processing Standards Publication 204, August 2024.

[23] National Institute of Standards and Technology. FIPS 205: Stateless hash-based digital signature standard. Federal Information Processing Standards Publication 205, August 2024.

[24] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. NIST Post-Quantum Cryptography Standardization Round 3 Submission, 2020. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[25] John M. Schanck, Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. NTRU-HRSS-KEM. NIST Post-Quantum Cryptography Standardization Round 1 Submission, 2017. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions.

[26] Gregor Seiler. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018.

[27] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[28] Joseph H. Silverman. Almost inverses and fast NTRU key creation. Technical Report NTRU Tech Report #014v1, NTRU Cryptosystems, March 1999. Technical report dated March 15, 1999.

[29] SMAUG-T Team. SMAUG-T: the key exchange algorithm based on module-lwe and module-lwr. KpqC Competition Final Version Specification, 2026. Document modified February 4, 2026.

[30] Joo Woo, Jonghyun Kim, Ga Hee Hong, Seungwoo Lee, Minkyu Kim, Hochang Lee, and Jong Hwan Park. NTRU+sign: Compact NTRU-based signatures using bimodal distributions. *Designs, Codes and Cryptography*, 94, 2026. Article number 2.

[31] Jiang Zhang, Dengguo Feng, and Di Yan. NEV: Faster and smaller NTRU encryption using vector decoding. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VII*, volume 14444 of *LNCS*, pages 157–189. Springer, Singapore, December 2023.

[32] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, and William Whyte. NTRUEncrypt. NIST Post-Quantum Cryptography Standardization Round 1 Submission, 2017. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions.