ARMv8 상에서 NTRU+KEM576 NTT 구현

https://youtu.be/LaiQHDS2zms

NTRU+

- KpqC 2라운드 후보군 알고리즘(격자 기반 알고리즘)
- NTRU의 약점을 보완하기 위해 안전성 강화를 초점으로 설계
  - NTRU의 제한사항: 복잡한 샘플링 분포 & 다른 격자 기반 알고리즘보다 느린 알고리즘
- NTRU보다 효율적으로 메시지에 대한 샘플링 가능
- Re-encryption을 사용하지 않는 FO(후지사키-오카모토) 적용하여 IND-CPA 보안성 만족
- NTT-friendly Ring과 최적화된 알고리즘을 활용하여 캡슐화와 디캡슐화 연산에서 효율적인 성능 달성
- Radix-3 NTT 구현 사용

Scheme	n	q	Public key	Secret key	Ciphertext
NTRU+576	576	3,457	864	1,760	864
NTRU+768	768		1,152	2,336	1,152
NTRU+864	864		1,296	2,624	1,296
NTRU+1152	1,152		1,728	3,488	1,728

NTRU+ update

10월에 수행한 내용과 달리 추가 수정된 구현 존재
추후 구현 예정
현재 구현은 NTRU+ v2.2.0 기준

[NTRU+] NTRU+ update (version 2.2.1) 조회수 45회

김종현

2024. 10. 10. 오후 1:17:55

받는사람 KpqC-bulletin

Dear all,

We would like to upload a revised version (denoted as version 2.2.1) of the NTRU+, including the revised specification and updated implementations.

Specification: [link]
Implementations: [link]

You can also download the above files in the Github:
https://github.com/ntruplus/ntruplus/tree/KpqC-Round2-(version-2.2.1)

1.Adjustment in Lazy Barrett Reduction
While the code functioned correctly, there was an issue with the placement of Barrett reduction in the inverse NTT implementation in worst-case scenarios. To resolve this, we adjusted the placement of Barrett reduction across all relevant parts of the code. In the optimized implementation, we further minimized the use of Barrett reduction by unrolling loops in the inverse NTT. Note that, prior to Version 2.2.1, the reference and optimized implementations were identical. However, starting with Version 2.2.1, we have decided to manage them separately.

2.Replacing Barrett Reduction with Montgomery Reduction
In both the reference and optimized implementations, Barrett reduction was applied at the end of the NTT layers. In the optimized implementation, we replaced the Barrett reduction with Montgomery reduction in the final NTT layer to improve performance.

3.Updates to consts.c for AVX2
We revised the values in the `consts.c` file for NTRU+{KEM,PKE}864, which were previously set in the range 0 to q-1. These values have now been updated to the range -(q-1)/2 to (q-1)/2.

NTRU+ update

```
void ntt(int16_t r[NTRUPLUS_N], const int16_t a[NTRUPLUS_N])
{
        int16_t t1,t2,t3;
        int16_t zeta1,zeta2;

        int k = 1;

        zeta1 = zetas[k++];

        for(int i = 0; i < NTRUPLUS_N/2; i++)
        {
                t1 = fqmul(zeta1, a[i + NTRUPLUS_N/2]);

                r[i + NTRUPLUS_N/2] = a[i] + a[i + NTRUPLUS_N/2] - t1;
                r[i               ] = a[i]                      + t1;
        }

    for(int step = NTRUPLUS_N/6; step >= 32; step = step/3)
    {
            for(int start = 0; start < NTRUPLUS_N; start += 3*step)
            {
                    zeta1 = zetas[k++];
                    zeta2 = zetas[k++];

                    for(int i = start; i < start + step; i++)
                    {
                            t1 = fqmul(zeta1, r[i +    step]);
                            t2 = fqmul(zeta2, r[i + 2*step]);
                            t3 = fqmul(-886, t1 - t2);

                            r[i + 2*step] = r[i] - t1 - t3;
                            r[i +   step] = r[i] - t2 + t3;
                            r[i         ] = r[i] + t1 + t2;
                    }
            }
    }
```

```
int16_t montgomery_reduce(int32_t a)
{
    int16_t t;

    t = (int16_t)a*QINV;
    t = (a - (int32_t)t*NTRUPLUS_Q) >> 16;
    return t;
}
```

NTRU+ v2.2.1 (241010)

```
    for(int step = 16; step >= 4; step >>= 1)
    {
            for(int start = 0; start < NTRUPLUS_N; start += (step << 1))
            {
                    zeta1 = zetas[k++];

                    for(int i = start; i < start + step; i++)
                    {
                            t1 = fqmul(zeta1, r[i + step]);

                            r[i + step] = barrett_reduce(r[i] - t1);
                            r[i       ] = barrett_reduce(r[i] + t1);
                    }
            }
    }
```

NTRU+ v2.2.0 (241005)

```
for(int step = 16; step >= 4; step >>= 1)
{
    for(int start = 0; start < NTRUPLUS_N; start += (step << 1))
    {
        zeta1 = zetas[k++];

        for(int i = start; i < start + step; i++)
        {
            t1 = fqmul(zeta1, r[i + step]);

            r[i + step] = r[i] - t1;
            r[i       ] = r[i] + t1;
        }
    }
}

for (int i = 0; i < NTRUPLUS_N; i++)
{
    r[i] = barrett_reduce(r[i]);
}
```

state-of-the-art

IACR Communications in Cryptology 2024 게재된 논문

IACR Communications in Cryptology
ISSN 3006-5496, Vol. 1, No. 2, 63 pages. https://doi.org/10.62056/a0ivr-10k

Check for updates

A Survey of Polynomial Multiplications for Lattice-Based Cryptosystems

Vincent Hwang

Max Planck Institute for Security and Privacy, Bochum, Germany

Abstract. We survey various mathematical tools used in software works multiplying polynomials in

with instruction set architectures/extensions Armv7-M, Armv7E-M, Armv8-A, and AVX2.

Communications in Cryptology

Home Papers Submissions FAQ Contact Editorial board

Home / Volume 1 / Issue 2 / 1

A Survey of Polynomial Multiplications for Lattice-Based Cryptosystems

Authors

Vincent Hwang

Author Info

PDF

DOI: https://doi.org/10.62056/a0ivr-10k

Keywords: Lattice-based cryptography, Polynomial multiplication, Modular arithmetic, Homomorphism, Vectorization

History

Submitted: 2023-12-27
Accepted: 2024-06-04
Published: 2024-07-08

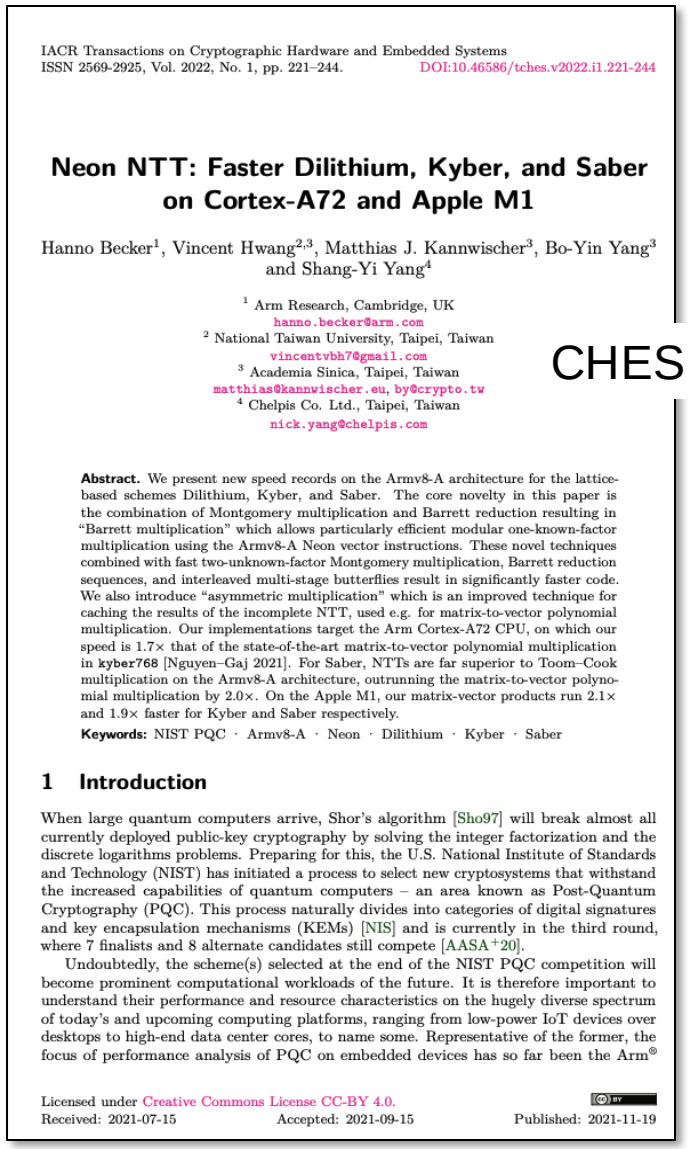
Abstract

Prime, and Saber.' All polynomial multiplications takes the form $\mathbb{Z}_q[x]/\langle x^n - \alpha x - \beta \rangle$. We

E-mail: vincentvbh7@gmail.com (Vincent Hwang)

Received: 2023-12-27 Accepted: 2024-06-04

CHES 2022 발표된 논문



IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 1, pp. 221–244. DOI:10.46586/tches.v2022.i1.221-244

Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1

Hanno Becker[1], Vincent Hwang[2,3], Matthias J. Kannwischer[3], Bo-Yin Yang[3] and Shang-Yi Yang[4]

[1] Arm Research, Cambridge, UK
hanno.becker@arm.com
[2] National Taiwan University, Taipei, Taiwan
vincentvbh7@gmail.com
[3] Academia Sinica, Taipei, Taiwan
matthias@kannwischer.eu, by@crypto.tw
[4] Chelpis Co. Ltd., Taipei, Taiwan
nick.yang@chelpis.com

Abstract. We present new speed records on the Armv8-A architecture for the lattice-based schemes Dilithium, Kyber, and Saber. The core novelty in this paper is the combination of Montgomery multiplication and Barrett reduction resulting in "Barrett multiplication" which allows particularly efficient modular one-known-factor multiplication using the Armv8-A Neon vector instructions. These novel techniques combined with fast two-unknown-factor Montgomery multiplication, Barrett reduction sequences, and interleaved multi-stage butterflies result in significantly faster code. We also introduce "asymmetric multiplication" which is an improved technique for caching the results of the incomplete NTT, used e.g. for matrix-to-vector polynomial multiplication. Our implementations target the Arm Cortex-A72 CPU, on which our speed is 1.7× that of the state-of-the-art matrix-to-vector polynomial multiplication in **kyber768** [Nguyen–Gaj 2021]. For Saber, NTTs are far superior to Toom–Cook multiplication on the Armv8-A architecture, outrunning the matrix-to-vector polynomial multiplication by 2.0×. On the Apple M1, our matrix-vector products run 2.1× and 1.9× faster for Kyber and Saber respectively.

Keywords: NIST PQC · Armv8-A · Neon · Dilithium · Kyber · Saber

1 Introduction

When large quantum computers arrive, Shor's algorithm [Sho97] will break almost all currently deployed public-key cryptography by solving the integer factorization and the discrete logarithms problems. Preparing for this, the U.S. National Institute of Standards and Technology (NIST) has initiated a process to select new cryptosystems that withstand the increased capabilities of quantum computers – an area known as Post-Quantum Cryptography (PQC). This process naturally divides into categories of digital signatures and key encapsulation mechanisms (KEMs) [NIS] and is currently in the third round, where 7 finalists and 8 alternate candidates still compete [AASA+20].

Undoubtedly, the scheme(s) selected at the end of the NIST PQC competition will become prominent computational workloads of the future. It is therefore important to understand their performance and resource characteristics on the hugely diverse spectrum of today's and upcoming computing platforms, ranging from low-power IoT devices over desktops to high-end data center cores, to name some. Representative of the former, the focus of performance analysis of PQC on embedded devices has so far been the Arm®

Received: 2021-07-15 Accepted: 2021-09-15 Published: 2021-11-19

https://cic.iacr.org/p/1/2/1

state-of-the-art

Neon NTT: Faster Dilithium, Kyber, and Saberon Cortex-A72 and Apple M1
https://tches.iacr.org/index.php/TCHES/article/view/9295/8861

3.2.3 Montgomery multiplication via doubling

Ordinary Montgomery multiplication does not lend itself to a straightforward implementation in Neon because Neon does not offer a multiply-high instruction. This is the reason why e.g. [NG21] use the long multiply `UMULL` to implement Montgomery multiplication.

We propose to use the doubling multiply-high instruction `QDMULH` instead to implement Algorithm 7. Moreover, the final step $\mathbf{mont}_N^{\pm}(z) \leftarrow \frac{z-c}{2}$ can be implemented via the halving subtract instruction `SHSUB`.

The resulting Neon sequence is shown in Algorithm 12. It provides a 5-instruction sequence for Montgomery multiplication of two unknown values, and a 4-instruction sequence if one factor is a constant.

Algorithm 12 Montgomery multiplication with doubling.

Require: N odd modulus, $R = 2^n > N$
Require: $a, b \in \mathbb{Z}$ representative mod N with $|a|, |b| < R$.
Require: Precomputed $T = \overline{N}^{-1} \in \mathbb{Z}_R$.
Ensure: $\mathbf{mont}_N^+(a,b)$ representative of $\underline{abR}^{-1} \in \mathbb{Z}_N$ satisfying $|\mathbf{mont}_N^+(a,b)| \leq \frac{|a||b|}{2^n} + \frac{N}{2}$.

```
1: sqdmulh z, a, b
2: mul k, a, bT mod± R
3: sqdmulh c, k, N
4: shsub z, z, c
```

3.2.2 Barrett reduction

We comment on the implementation of Barrett reduction (Algorithm 3) in the Neon instruction set. For Barrett reduction of a single-width value, we choose R as large as possible in Algorithm 3 so that $\lfloor \frac{R}{N} \rceil$ is still a single-width signed value, increasing precision of the approximation. To compute $\left\lfloor \frac{z \lfloor \frac{R}{N} \rceil}{R} \right\rceil$, however, we can no longer use `SQRDMULH` as in Algorithm 10 but have to split it into `SQDMULH` and a rounding right shift `SRSHR`. We show our implementation of Barrett reduction in the Neon instruction set in Algorithm 11.

Old Algorithm 11 Barrett Reduction (Vectorized) [NG21, Algorithm 13]

Require: Odd modulus `N`, i bits long.
Require: Radix $R = 2^{16}$ or 2^{32}
Require: Rounding constant `r` $= 2^{i-3}$.
Require: Multiplier `V` $= \lceil 2^{i-2}R/\mathtt{N} \rfloor$
Ensure: $z \equiv B \pmod N$, $\frac{-N}{2} \leq z < \frac{N}{2}$

```
1: smull T0, B, V
2: smull2 T1, B, V
3: uzp2 T0, T0, T1
4: add T1, r, T0
5: sshr T1, T1, #(i − 2)
6: mls z, T1, N
```

Algorithm 11 Barrett Reduction for Neon

Require: Odd modulus `N`, i bits long.
Require: Radix $R = 2^{16}$ or 2^{32}
Require: Multiplier `V` $= \lceil 2^{i-2}R/\mathtt{N} \rfloor$
Ensure: $z \equiv B \pmod N$, $\frac{-N}{2} \leq z < \frac{N}{2}$

```
1: sqdmulh T0, B, V
2: srshr T1, T1, #(i − 1)
3: mls z, T1, N
```

`sqdmulh` = doubling multiplication, high half
`sshr` = signed shift right
`srshr` = signed shift right with rounding
Note: `N,V` can be one lane of a Neon register

- ***shsub***(Signed Halving Subtract): (a – b) / 2
- ***sqdmulh***(Signed Saturating Doubling Multiply Returning High Half)
  : a x b 수행 후, 상위 (32비트 또는 16비트) 반환
- ***srshr***(Signed Rounding Shift Right) : 우측 쉬프트 연산 수행 후, 소수점 이하의 값을 반올림
- ***mls***(Multiply and Subtract) : mls result, a, b 일 때 : result-(a x b) 값을 반환

구현 기법

```
int16_t montgomery_reduce(int32_t a)
{
    int16_t t;

    t = (int16_t)a*QINV;
    t = (a - (int32_t)t*NTRUPLUS_Q) >> 16;
    return t;
}
```

```
for(int step = 16; step >= 4; step >>= 1)
{
    for(int start = 0; start < NTRUPLUS_N; start += (step << 1))
    {
        zeta1 = zetas[k++];

        for(int i = start; i < start + step; i++)
        {
            t1 = fqmul(zeta1, r[i + step]);

            r[i + step] = r[i] - t1;
            r[i       ] = r[i] + t1;
        }
    }
}
```

3.2.3 Montgomery multiplication via doubling

Ordinary Montgomery multiplication does not lend itself to a straightforward implementation in Neon because Neon does not offer a multiply-high instruction. This is the reason why e.g. [NG21] use the long multiply `UMULL` to implement Montgomery multiplication.

We propose to use the doubling multiply-high instruction `QDMULH` instead to implement Algorithm 7. Moreover, the final step $\mathbf{mont}_N^{\pm}(z) \leftarrow \frac{z-c}{2}$ can be implemented via the halving subtract instruction `SHSUB`.

The resulting Neon sequence is shown in Algorithm 12. It provides a 5-inst sequence for Montgomery multiplication of two unknown values, and a 4-inst sequence if one factor is a constant.

Algorithm 12 Montgomery multiplication with doubling.

Require: N odd modulus, $R = 2^n > N$
Require: $a, b \in \mathbb{Z}$ representative mod N with $|a|, |b| < R$.
Require: Precomputed $T = \overline{N}^{-1} \in \mathbb{Z}_R$.
Ensure: $\mathbf{mont}_N^{+}(a, b)$ representative of $\underline{abR}^{-1} \in \mathbb{Z}_N$ satisfying $|\mathbf{mont}_N^{+}(a, b)| \leq \frac{|a||b|}{2^n} + \frac{N}{2}$.

1: `sqdmulh` z, a, b
2: `mul` k, a, bT `mod`$^{\pm}$ R
3: `sqdmulh` c, k, N
4: `shsub` z, z, c

```
.macro mont_reduce s, n, t
    sqdmulh.8h \s, \n, \t
    mul.8h  v27, \t, v3
    mul.8h  v8, \n, v27
    sqdmulh v16.8h, v8.8h, v6.8h
    shsub.8h    \s, \s, v16
.endm
```

```
.macro len4
mov     x4, #36

loop6:
    ld1R     {v2.8h}, [x2], #2 //zetas [a]
    ld1R     {v22.8h},[x2], #2 //zetas[a+1]
    zip1.2d        v2, v2, v22 //zetas[a]||zetas[a+1]

    ld1     {v1.8h}, [x0] //r[j]
    add     x0, x0, #16 //2*4 = 8  2(16bit)
    ld1     {v0.8h}, [x0] //r[j+len}

    zip1.2d        v18, v1, v0
    zip2.2d         v19, v1, v0

    mont_reduce     v7, v19, v2

    sub.8h      v19, v18,v7
    add.8h      v18, v18, v7


    zip1.2d        v1, v18, v19
    zip2.2d         v0, v18, v19


    ST1         {v0.8h}, [x0]
    add         x0, x0, #-16
    ST1         {v1.8h}, [x0], #16  //2*8


add         x4, x4, #-1
add         x0, x0, #16
cbnz        x4, loop6

.endm
```

구현 기법(Barret reduction)

```
for (int i = 0; i < NTRUPLUS_N; i++)
{
    r[i] = barrett_reduce(r[i]);
}
```

```
int16_t barrett_reduce(int16_t a)
{
    int16_t t;
    const int16_t v = ((1<<26) + NTRUPLUS_Q/2)/NTRUPLUS_Q;

    t  = ((int32_t)v*a + (1<<25)) >> 26;
    t *= NTRUPLUS_Q;
    return a - t;
}
```

3.2.2 Barrett reduction

We comment on the implementation of Barrett reduction (Alg
instruction set. For Barrett reduction of a single-width value, v
possible in Algorithm 3 so that $\lfloor \frac{R}{N} \rceil$ is still a single-width signed va
of the approximation. To compute $\left\lfloor \frac{z\lfloor \frac{R}{N} \rceil}{R} \right\rceil$, however, we can no longer use `SQRDMULH` as in Algorithm 10 but have to split it into `SQDMULH` and a rounding right shift `SRSHR`. We show our implementation of Barrett reduction in the Neon instruction set in Algorithm 11.

Old Algorithm 11 Barrett Reduction (Vectorized) [NG21, Algorithm 13]

Require: Odd modulus `N`, i bits long.
Require: Radix $R = 2^{16}$ or 2^{32}
Require: Rounding constant $\mathtt{r} = 2^{i-3}$.
Require: Multiplier $\mathtt{V} = \lceil 2^{i-2}R/\mathtt{N} \rfloor$
Ensure: $z \equiv B \pmod N$, $\frac{-N}{2} \leq z < \frac{N}{2}$

```
1: smull T0, B, V
2: smull2 T1, B, V
3: uzp2 T0, T0, T1
4: add T1, r, T0
5: sshr T1, T1, #(i − 2)
6: mls z, T1, N
```

Algorithm 11 Barrett Reduction for Neon

Require: Odd modulus `N`, i bits long.
Require: Radix $R = 2^{16}$ or 2^{32}
Require: Multiplier $\mathtt{V} = \lceil 2^{i-2}R/\mathtt{N} \rfloor$
Ensure: $z \equiv B \pmod N$, $\frac{-N}{2} \leq z < \frac{N}{2}$

```
1: sqdmulh T0, B, V
2: srshr T1, T1, #(i − 1)
3: mls z, T1, N
```

`sqdmulh` = doubling multiplication, high half
`sshr` = signed shift right
`srshr` = signed shift right with rounding
Note: `N`,`V` can be one lane of a Neon register

```
.macro barret
    movi.16b v25, #0
    trn1.8h          v20, v25, v20
    trn1.8h          v6, v25, v6

    mov      x4, #72
loop:
    ld1      {v1.8h}, [x0] //r[j]
    trn1.8h         v18, v25, v1
    trn2.8h          V19, v25, v1
/*
    smull       v26.4s, v18.4h, v20.4h
    smull2       v27.4s, v18.8h, v20.8h
    uzp2.8h         v26, v26, v27

    smull       v27.4s, v19.4h, v20.4h
    smull2       v28.4s, v19.8h, v20.8h
    uzp2.8h         v27, v27, v28
  */

    sqdmulh.4s      v26, v18, v20
    sqdmulh.4s      v27, v19, v20
    srshr.4s     v26, v26, #27 //#26 : i-2 #27  :i-1
    srshr.4s     v27, v27, #27
    mls.4s       v18, v26, v6
    mls.4s       v19, v27, v6

    trn2.8h         v18, v18, v19

    st1      {v18.8h}, [x0],#16
    add          x4, x4, #-1
    cbnz         x4, loop
.endm
```

성능 평가

- 구현 환경
  - Apple M1 pro 칩이 탑재된 Apple Macbook Pro 16(3.2GHz)
  - Framework: Xcode Integrated Development Environment
  - Compiled with –O3 option(i.e. fastest)

- 평가 방법
  - ARMv8 상에서 NTRU+ 최적화 구현이 없기 때문에 성능은 KpqCleanver2 프로젝트 reference-C와 비교
  - 동작시간 및 clock cycle을 측정하기 위해 NTRU+ 알고리즘을 **10,000**회 반복

unit : cc

NTRU+KEM576 (241005ver)	Keygen	Encap	Decap
Reference-C(KpqClean)	128,564	65,247	45,593
asm	**112,920**	**51,399**	**32,204**
diff	**1.14x**	**1.27x**	**1.41x**

향후 계획

- Inverse NTT 구현
- 모든 NTRU+ 파라미터에 대해 구현
- NTT 개선 예정
  - NEON NTT 논문에 따르면

NTT 내부에서 사용되는 Montgomery 구현을 바렛 리덕션 과 섞어 구현하는 기법이 main idea

➔HAETAE on ARMv8은 현재 NTT 내부에서 Montgomery 구현만 사용하였으므로 개선이 가능할 것으로 보입니다(pqclean의 딜리시움 코드 추가 분석 필요)

➔하지만, NTRU+의 경우 레퍼런스 NTT 내부에서 이미 Montgomery와 바렛을 같이 사용하기 때문에 개선 가능한지는 잘 모르겠습니다.

➔우선 NEON NTT 논문 더 분석해서 NTRU+도 적용한지 확인해보도록 하겠습니다.

Q & A