Efficient Implementation of Kyber on Mobile Devices

Lirui Zhao[1], Jipeng Zhang[1], Junhao Huang[2], Zhe Liu[13(✉)], Gerhard Hancke[4]

[1]Nanjing University of Aeronautics and Astronautics, Jiangsu, China

[2]Beijing Normal University-Hong Kong Baptist University United International College, Guangdong, China

[3]State Key Laboratory of Cryptology, Beijing, China

[4]City University of Hong Kong, Hong Kong, China

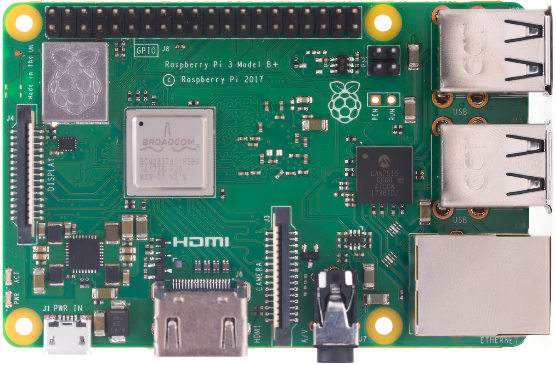
December 10, 2021

Lattice-based Cryptography

- RSA and ECC: Discrete Logarithm and Integer Factorization Problems
  - Hard problems can be solved by Shor's algorithm
- Lattice-based Cryptography: Hard for quantum computers
  - Kyber is a key encapsulation mechanism (Round 3 candidate): it has three parameter sets to scale security to different levels
- NIST Post-Quantum-Cryptography (PQC) Project
  - 2016, Formal call for proposals
  - 2017, Round 1 algorithms announced (69 submissions)
  - 2019, Round 2 algorithms announced (26 algorithms)
  - 2020, Round 3 algorithms announced (7 Finalists and 8 Alternates)
  - ......

Implementation Platform

- Raspberry Pi 3 (including ARM Cortex-A53 processor)
    - ARMv8-A is the first processor architecture of ARM that supports 64-bit instruction set.
    - The SIMD instruction set NEON in ARMv8-A is widely used to parallelize instructions.



Motivation & Contribution

- Motivation
  - The most important and time-consuming operation in Kyber scheme is polynomial multiplication.
  - The Number Theoretic Transform (NTT) can greatly improve the performance of polynomial multiplication.
- Contributions: Efficient implementation of Kyber
  - Parallel design: NEON instruction set.
  - Optimized Barrett and Montgomery modular reductions.
  - Improved utilization of registers.
  - The NTT layer merging technique and various strategies.

Kyber scheme

- Kyber is an IND-CPA secure encryption scheme.
- It includes key-generation, encryption, and decryption.

Algorithm 1 Kyber.CPAPKE.KeyGen

1: $\rho, \sigma \xleftarrow{\$} \{0,1\}^{256} \times \{0,1\}^{256}$
2: $\hat{\mathbf{A}} \in \mathcal{R}_q^{k\times k} \leftarrow$ SampleUniform (ρ)
3: $\mathbf{s}, \mathbf{e} \in \mathcal{R}_q^k \leftarrow \mathrm{SampleCBD}(\sigma)$
4: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s}) + \mathrm{NTT}(\mathbf{e})$
5: **return** $pk = (\rho, \hat{\mathbf{t}}), sk = \hat{\mathbf{s}}$

Algorithm 2 Kyber.CPAPKE.Enc

Require: Public key $pk = (\rho, \hat{\mathbf{t}})$
Require: Message $m \in \mathcal{R}_q$
Require: Random coins $r \in \{0,1\}^{256}$
Ensure: Ciphertext $(\mathbf{u}', v')$
1: $\hat{\mathbf{A}} \in \mathcal{R}_q^{k\times k} \leftarrow$ SampleUniform (ρ)
2: $\mathbf{r}, \mathbf{e}_1 \in \mathcal{R}_q^k \leftarrow$ SampleCBD (r)
3: $e_2 \in \mathcal{R}_q \leftarrow \mathrm{SampleCBD}(r)$
4: $\hat{\mathbf{r}} \leftarrow \mathrm{NTT}(\mathbf{r})$
5: $\mathbf{u} \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
6: $v \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + m$
7: **return** $(\mathrm{Compress}(\mathbf{u}), \mathrm{Compress}(v))$

Algorithm 3 Kyber.CPAPKE.Dec

Require: Secret key $sk = \hat{\mathbf{s}}$
Require: Compressed ciphertext $(\mathbf{u}', v')$
Ensure: Message $m \in \mathcal{R}_q$
1: $\mathbf{u} \leftarrow$ Decompress $(\mathbf{u}')$
2: $v \leftarrow$ Decompress (v')
3: **return** $v - \mathrm{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathrm{NTT}(\mathbf{u}))$

Modular Reduction

- Modular reduction is the core operation when computing NTT-based polynomial multiplication.
- The traditional method of calculating modular reduction contains division, which is expensive and non-constant time.
- The common optimizations: Barrett reduction and Montgomery reduction.

Algorithm 4 Signed Montgomery reduction [14]

Require: $0 < q < \frac{\beta}{2}$ odd , $-\frac{\beta}{2}q \leq a = a_1\beta + a_0 < \frac{\beta}{2}q$ where $0 \leq a_0 < \beta,\ \beta = 2^{16}$
Ensure: $r' \equiv \beta^{-1}a(\text{mod } q), -q < r' < q$

1: $m \leftarrow a_0 q^{-1} \bmod {}^{\pm}\beta$ ▷ Only low-limb needed
2: $t_1 \leftarrow \left\lfloor \frac{mq}{\beta} \right\rceil$ ▷ Only high-limb needed
3: $r' \leftarrow a_1 - t_1$

Algorithm 5 Signed Barrett reduction for one word [14]

Require: $0 \leq q < \frac{\beta}{2}, -\frac{\beta}{2} \leq a < \frac{\beta}{2}, \beta = 2^{16}$
Ensure: $r \equiv a \pmod{q}$ with $0 \leq r \leq q$

1: $v \leftarrow \left\lfloor \frac{2^{\lfloor \log(q) \rfloor - 1}\beta}{q} \right\rceil$
2: $t \leftarrow \left\lfloor \frac{av}{2^{\lfloor \log(q) \rfloor - 1}\beta} \right\rceil$ ▷ Only high-limb needed
3: $t \leftarrow tq \bmod \beta$ ▷ Only low-limb needed
4: $r \leftarrow a - t$

Barrett Reduction

The Barrett reduction approximately represents $\frac{1}{q}$ by using multiplication and shift operations instead of division:

$$\frac{1}{q} \approx \frac{v}{2^k} \rightarrow v = \lfloor \frac{2^k}{q} \rceil \tag{1}$$

The result of the Barrett reduction is computed by:

$$a\ mod\ q \ = a - ((a * v) >> k) \cdot q \tag{2}$$

In Kyber, $q = 3329$, $k = 14$, $v = 5$. a and v are 16-bit integers, so the computing of $a * v$ will produce a 32-bit integer. We propose an improved Barrett reduction as follows:

$$a\ mod\ q \ = a - (((a >> 3) * v) >> 11) \cdot q \tag{3}$$

Barrett Reduction

In our implementation, the multiplication $(a >> 3) * v$ doesn't extend the data type to 32-bit, but only uses the 16-bit intermediate to make full use of the bandwidth advantage of the NEON registers, as given in Listing 1.

Listing 1 Barrett Reduction (BarR)

Input: va.8h = $[a_0, a_1, \ldots, a_7]$
Input: vq.h[0] = q = 3329
Input: vc.8h = $1 << 10$
Input: vt1, vt2 is intermediate vector register
Output: va.8h = $[a_0, a_1 \ldots, a_7]$

```
1: sshr vt1.8h, va.8h, 3            ▷ t1 = a >> 3
2: shl vt2.8h, vt1.8h, 2            ▷ t2 = t1 << 2 = t1 * 4
3: add vt1.8h, vt1.8h, vt2.8h       ▷ t1 = t1 * 5
4: add vt1.8h, vt1.8h, vc.8h        ▷ t1+ = (1 << 10)
5: sshr vt1.8h, vt1.8h, 11          ▷ t1 = t1 >> 11
6: mls va.8h, vt1.8h, vq.8h         ▷ a- = t * q
```

Lazy Reduction

In Kyber, the addition of polynomial coefficients when computing INTT (inverse of NTT) is followed by the Barrett reduction. Not all the results need to be reduced, so when the addition do not overflow, Barrett reduction can be removed (Lazy Reduction).

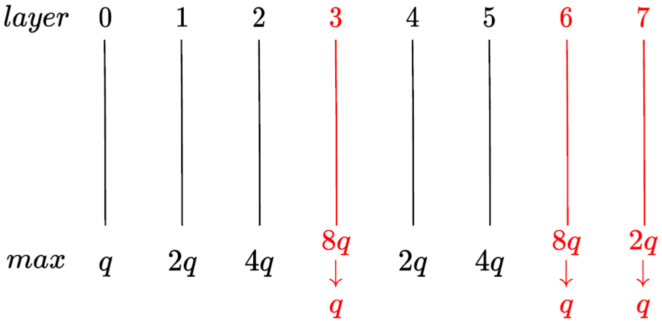


Figure 1: The position of Barrett reduction in the INTT

Montgomery Reduction

- Montgomery multiplication and Montgomery reduction:

Listing 2 Montgomery Multiplication (MontM)

Input: va.8h = $[a_0, a_1, \ldots, a_7]$
Input: vb.8h = $[b_0, b_1, \ldots, b_7]$
Input: vt1, vt2 are intermediate vector registers
Output: va1.8h = $[a_0, a_1 \ldots, a_7]$

```
smull vt1.4s, va.4h, vb.4h      ▷ t1 = (L)a * b
smull2 va.4s, va.8h, vb.8h      ▷ va = (H)a * b
MontR vt1.4s, va.4s, vt2        ▷ MontR(a * b)
```

Listing 3 Montgomery Reduction (MontR)

Input: va1.4s = $[a_0, a_1, \ldots, a_3]$
Input: va2.4s = $[a_4, a_5, \ldots, a_7]$
Input: vq = q = 3329
Input: vr.4s = $[2^{16} - 1, \ldots, 2^{16} - 1]$
Input: vqp = q^{-1} = 62209
Input: vt is intermediate vector register
Output: va1.8h = $[a_0, a_1 \ldots, a_7]$

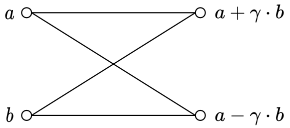
```
mul vt.4s, va1.4s, vqp              ▷ t = a1 * q^-1
and vt.16b, vt.16b, vr.16b          ▷ t = (LSB)t
mls va1.4s, vt.4s, vq               ▷ a1- = t * q
mul vt.4s, va2.4s, vqp              ▷ t = a2 * q^-1
and vt.16b, vt.16b, vr.16b          ▷ t = (LSB)t
mls va2.4s, vt.4s, vq               ▷ a2- = t * q
uzp2 va1.8h, va1.8h, va2.8h         ▷ a1 = (MSB)(a1, a2)
```

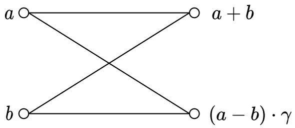
NTT (Number Theoretic Transform)

- NTT is used to speed up polynomial multiplication.

$$f * g = NTT^{-1}(NTT(f) \odot NTT(g)) \tag{4}$$

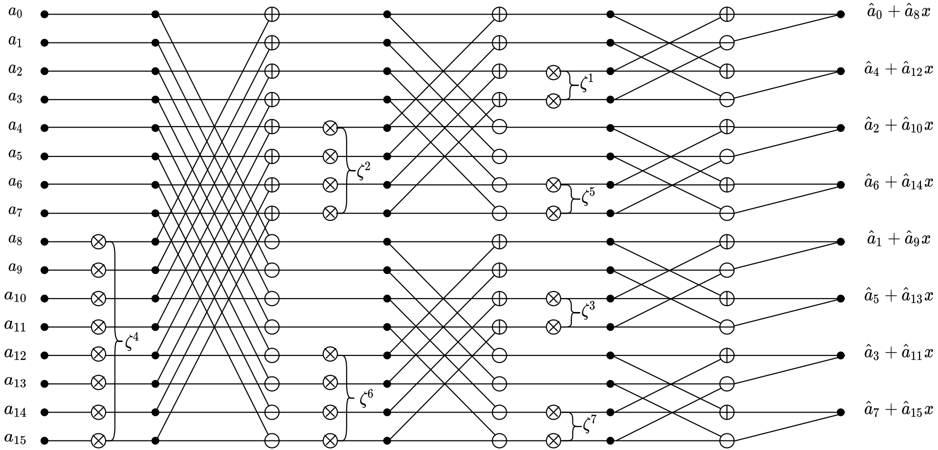
  where $\odot$ denotes the coefficient-wise multiplication.
- Basic operation is the butterfly transform.
- There are two types of butterfly units, Cooley-Tukey(CT) and Gentleman-Sande(GS) below.
- NTT: CT butterfly unit; INTT: GS butterfly unit.





NTT Layer Merging

- In NTT, the polynomial coefficients of each layer need to be loaded and stored.
- The NTT using CT butterfly operations for n = 16:



NTT Layer Merging

- For Kyber, $n = 256$, the 7-layer incomplete-NTT is available.
- The 7-layer incomplete-NTT of Kyber has various layer merging strategies, such as 1+6, 2+5, 3+4 layer merging.
- The 5-layer merging is more appropriate because we have enough registers to accommodate all values.
- As a result, we adopted the 2+5 layer merging strategy on ARMv8-A to implement NTT/INTT.

Implementation Results

- This table shows the results of optimized modules in Kyber512.
- Compared with pure C implementations, our Barrett and Montgomery reduction shows 8.52 and 8.89 times faster than the reference implementation.
- Our NTT and INTT achieved 11.89 and 13.45 times speedups compared with the reference implementation.

Module	ref	Our work	ref / Our work
BarR	2675	314	8.52
MontR	3413	384	8.89
NTT	16575	1394	11.89
INTT	27284	2028	13.45

Table: Performance and Comparison of Kyber512

Implementation Results

- The key encapsulation mechanism (KEM) in Kyber has different implementations of three parameter sets.
- Our optimized software achieved 1.77×, 1.85×, and 2.16× speedups for key generation, encapsulation, and decapsulation compared with Kyber's reference implementation.

Schemes		ref	Our work	ref / Our work
Kyber512	K	464238	262249	1.77
	E	637189	343538	1.85
	D	791471	367236	2.16
Kyber768	K	807544	484745	1.67
	E	1030702	594449	1.73
	D	1274856	641491	1.99
Kyber1024	K	1189371	783209	1.52
	E	1491847	930112	1.60
	D	1727240	1011992	1.71

Table II: Performance and Comparison of KEM

Conclusion

- Efficient Implementation of Kyber on Mobile Devices
  - The optimized Barrett reduction increases the utilization of vector registers.
  - Montgomery multiplication and reduction greatly improves the efficiency.
  - The layer merging technique substantially accelerates the efficiency in NTT.
- Our optimizations of modular reduction and NTT operations are useful for other works, such as NewHope.

Thanks

Thanks for listening