NTRU 다항식 곱셈

https://youtu.be/m5viVBG3EQc

NTRU

NTRU public key cryptosystem

Table 1. Performance analysis of NTRU with existing public key cryptosystems

Algorithm	Message Size (bits)	Key size (bits)	Key generation (ms)	Encryption (ms)	Decryption (ms)
RSA1024	1024	1024	1432	4.28	48.5
ECC168	160	168	65	140	67
NTRU263	416	1841	19.8	1.9	3.5

https://www.kci.go.kr/kciportal/ci/sereArticleSearch/kci?sereArticleSearchBean.artiId=ARI002345480)

- RLWE를 기반으로 **polynomial ring에서 기본 연산 수행**
  - $Z[x]$: Z에 대한 다항식 링 → 정수 계수를 갖는 모든 다항식들의 집합
  - $R = Z[x]/(Xn - 1)$: 모든 다항식들의 집합은 ring R 에서 정의
    - → 계수가 정수이고, n-1차 다항식 사용 : $a = a_0 + a_1x^1 + \ ... + a_{n-1}x^{n-1}$
- 기본 연산
  - **Circular Convolution** : 순환 합성곱 : O(NlogN)
    - → **다항식 곱셈**에 사용 → 시간 소모가 가장 많은 과정 → 연산량 줄일 필요가 있음
  - RSA(modular multiplication), ECC(Elliptic Curve Addition), NTRU(Convolution)
    - → Convolution 연산은 기존의 공개키 암호의 연산보다 **암/복호화가 빠르고 효율적**
- 격자에서 짧은 벡터를 찾는 어려움(SVP)을 기반으로 안전성을 제공 & 복호화
  - 양자 컴퓨팅 공격에도 안전

➢ **빠른 연산 속도 / SW, HW 구현 용이 / 적은 메모리 사용 / 키 생성 쉬움 / 양자 알고리즘 공격에 안전**

CryptoCraft LAB 8

```
-- internals --

poly_Rq_mul:
median: 192436
average: 192921

poly_S3_mul:
median: 195147
average: 195848

poly_Rq_inv:
median: 8447773
average: 8454492

poly_S3_inv:
median: 6443604
average: 6438311

randombytes for fg:
median: 11357
average: 12535

randombytes for rm:
median: 9869
average: 10309

sample_iid:
median: 390
average: 395
```

```
sample_iid_plus:
median: 1764
average: 1768

poly_lift:
median: 10097
average: 10116

poly_Rq_to_S3:
median: 3690
average: 3676

poly_Rq_sum_zero_tobytes:
median: 1194
average: 1201

poly_Rq_sum_zero_frombytes:
median: 1698
average: 1693

poly_S3_tobytes:
median: 466
average: 469

poly_S3_frombytes:
median: 4028
average: 4028
```

1라운드 제출물

- Karatsuba 사용

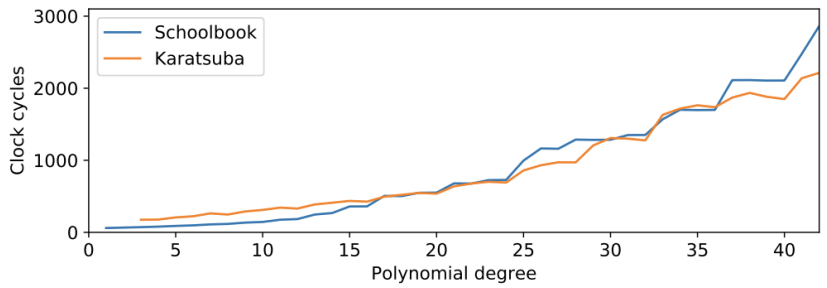
n/2 크기 4번의 곱셈

$$(a_0+a_1)\times(b_0+b_1)=\underbrace{a_0\times b_0}_{z_0}+\underbrace{a_1\times b_0+a_0\times b_1}_{z_1}+\underbrace{a_1\times b_1}_{z_2}$$

n/2 크기의 두 정수의 곱셈 3번, 덧셈 2번, 뺄셈 2

```
z2 = a1 * b1;
z0 = a0 * b0;
z1 = (a0 + a1) * (b0 + b1) - z0 - z2;
```

재귀 호출을 통해 그 곱을 계산
재귀 호출은 곱하는 수가 단번에 계산될 정도로 작아질 때까지 적용



schoolbook multiplication

			1	2	3	4
		X	5	6	7	8
			8	16	24	32
		7	14	21	28	
	6	12	18	24		
5	10	15	20			
5	16	34	60	61	52	32

code

```
void
ntru_ring_mult_coefficients(
    uint16_t const *a,          /*  in - pointer to polynomial a */
    uint16_t const *b,          /*  in - pointer to polynomial b */
    PARAM_SET const*param,
    uint16_t       *tmp,        /*  in - temp buffer of 3*padN elements */
    uint16_t       *c)          /* out - address for polynomial c */
{
    uint16_t i;
    uint16_t q_mask = param->q-1;

    memset(tmp, 0, 3*param->padN*sizeof(uint16_t));


    karatsuba(tmp, tmp+param->padN*2, a, b, param->N);

    for(i=0; i<param->N; i++)
    {
        c[i] = (tmp[i] + tmp[i+param->N]) & q_mask;
    }
    for(; i<param->padN; i++)
    {
        c[i] = 0;
    }
//    memset(tmp, 0, sizeof(uint16_t)*param->padN*3);
    return;
}
```

code

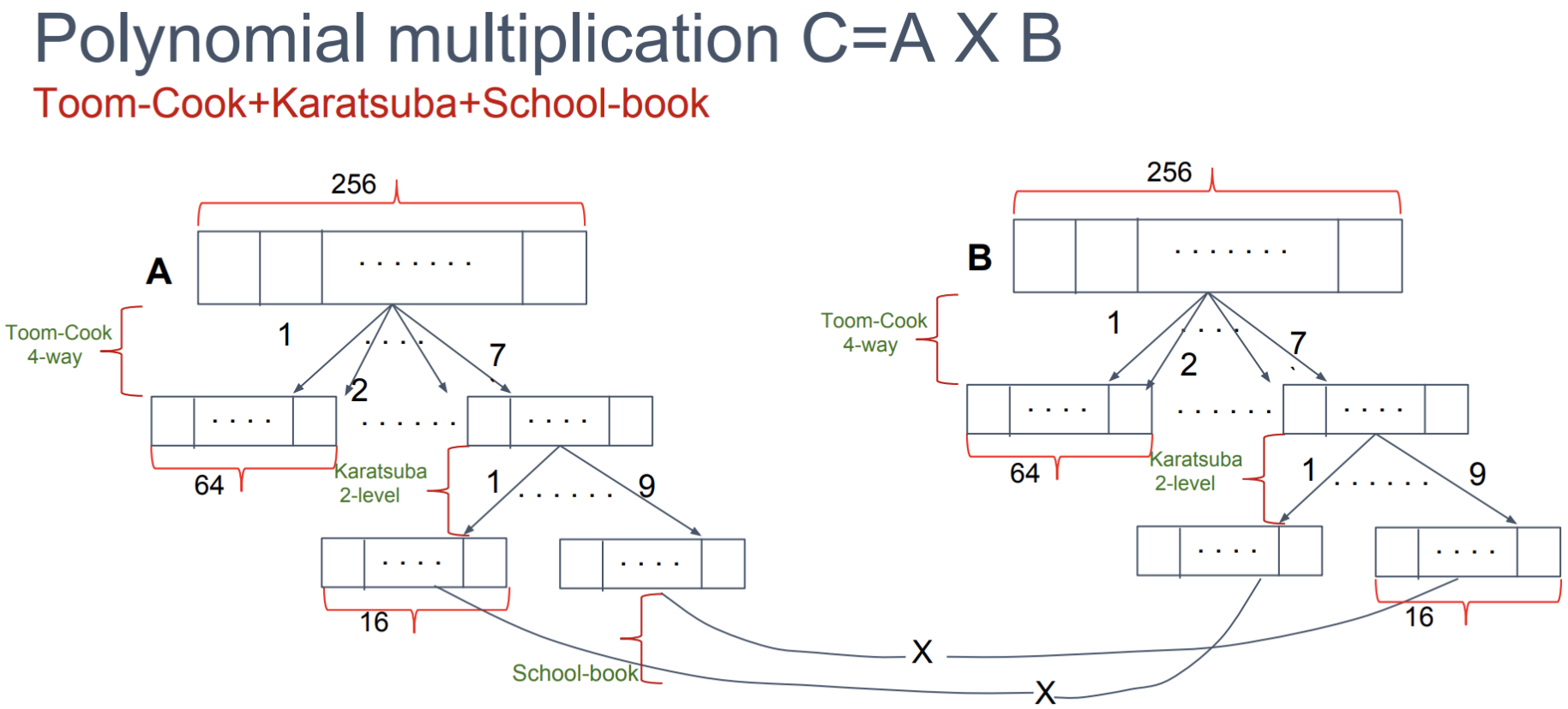
$$(a_0+a_1)\times(b_0+b_1)=\underbrace{a_0\times b_0}_{z_0}+\underbrace{a_1\times b_0+a_0\times b_1}_{z_1}+\underbrace{a_1\times b_1}_{z_2}$$

```
z2 = a1 * b1;
z0 = a0 * b0;
z1 = (a0 + a1) * (b0 + b1) - z0 - z2;
```

```
void
karatsuba(
    uint16_t        *res1,  /* out - a * b in Z[x], must be length 2k */
    uint16_t        *tmp1,  /*  in - k coefficients of scratch space */
    uint16_t const  *a,     /*  in - polynomial */
    uint16_t const  *b,     /*  in - polynomial */
    uint16_t const   k)     /*  in - number of coefficients in a and b */
{
    uint16_t i;

    uint16_t const p = k>>1;

    uint16_t *res2;
    uint16_t *res3;
    uint16_t *res4;
    uint16_t *tmp2;
    uint16_t const *a2;
    uint16_t const *b2;


    /* Grade school multiplication for small / odd inputs */
    if(k <= 32 || (k & 1) != 0)
    {
      grade_school_mul(res1,a,b,k);
      return;
    }

    res2 = res1+p;
    res3 = res1+k;
    res4 = res1+k+p;
    tmp2 = tmp1+p;
    a2 = a+p;
    b2 = b+p;

    for(i=0; i<p; i++)
    {
        res1[i] = a[i] - a2[i];
        res2[i] = b2[i] - b[i];
    }

    karatsuba(tmp1, res3, res1, res2, p);

    karatsuba(res3, res1, a2, b2, p);

    for(i=0; i<p; i++)
    {
      tmp1[i] += res3[i];
    }

    for(i=0; i<p; i++)
    {
        res2[i]  = tmp1[i];
        tmp2[i] += res4[i];
        res3[i] += tmp2[i];
    }

    karatsuba(tmp1, res1, a, b, p);

    for(i=0; i<p; i++)
    {
        res1[i]  = tmp1[i];
        res2[i] += tmp1[i] + tmp2[i];
        res3[i] += tmp2[i];
    }

    return;
}
```

현재 제출물

- Toom/Karatsuba 사용



현재 제출물

- Toom/Karatsuba 사용

Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates

Matthias J. Kannwischer, Joost Rijneveld and Peter Schwabe*

Radboud University, Nijmegen, The Netherlands
matthias@kannwischer.eu, joost@joostrijneveld.nl, peter@cryptojedi.org

Abstract. In this paper we optimize multiplication of polynomials in $\mathbb{Z}_{2^m}[x]$ on the ARM Cortex-M4 microprocessor. We use these optimized multiplication routines to speed up the NIST post-quantum candidates RLizard, NTRU-HRSS, NTRUEncrypt, Saber, and Kindi. For most of those schemes the only previous implementation that executes on the Cortex-M4 is the reference implementation submitted to NIST; for some of those schemes our optimized software is more than factor of 20 faster. One of the schemes, namely Saber, has been optimized on the Cortex-M4 in a CHES 2018 paper; the multiplication routine for Saber we present here outperforms the multiplication from that paper by 37%, yielding speedups of 17% for key generation, 15% for encapsulation and 18% for decapsulation. Out of the five schemes optimized in this paper, the best performance for encapsulation and decapsulation is achieved by NTRU-HRSS. Specifically, encapsulation takes just over 430 000 cycles, which is more than twice as fast as for any other NIST candidate that has previously been optimized on the ARM Cortex-M4.

Keywords: ARM Cortex-M4, Karatsuba, Toom, lattice-based KEMs, NTRU

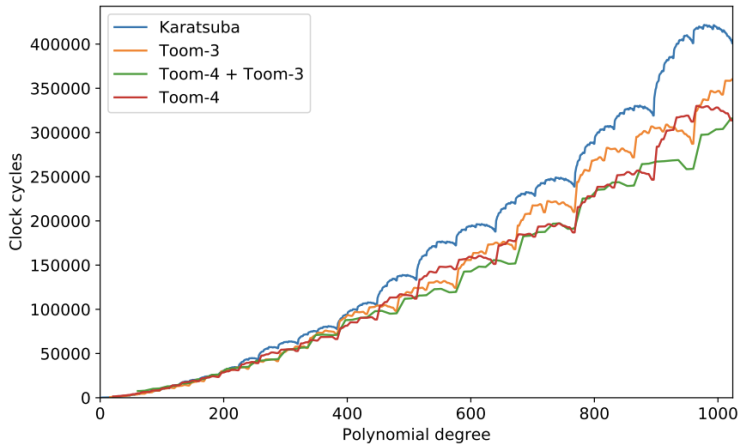
1 Introduction

In November 2017 the NIST post-quantum project [NIS16b] received 69 "complete and proper" proposals for future standardization of a suite of post-quantum cryptosystems. By September 2018, five of those 69 have been withdrawn. Out of the remaining 64 proposals, 22 are lattice-based public-key encryption schemes or key-encapsulation mechanisms (KEMs)[1]. Most of those lattice-based schemes use structured lattices and, as a consequence, require fast arithmetic in a polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/f$ for some n-coefficient polynomial $f \in \mathbb{Z}_q[x]$. Typically the largest performance bottleneck of these schemes is multiplication in $\mathcal{R}_q$.

Many proposals, for example NewHope [ADPS16, AAB+17], Kyber [ABD+17], and LIMA [SAL+17], choose q, n, and f such that multiplication in $\mathcal{R}_q$ can be done via very fast number-theoretic transforms. However, six schemes choose $q = 2^k$ which requires using a different algorithm for multiplication in $\mathcal{R}_q$. Specifically those six schemes are Round2 [GMZB+17], Saber [DKRV17], NTRU-HRSS [HRSS17b], NTRUEncrypt [ZCHW17], Kindi [Ban17], and RLizard [CPL+17]. Round2 recently merged with Hila5 [Saa17] into Round5 [BGML+18] and the Round5 team presented optimized software for the ARM Cortex-M4 processor in [SBGM+18]; the multiplication in Round5 has more structure, allowing for a specialized high-speed routine. In this paper we optimize the other five

*This work has been supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE) and by COST (European Cooperation in Science and Technology) through COST Action IC1403 (CRYPTACUS). Date: October 19, 2018

[1] see https://www.safecrypto.eu/pqclounge/



code

$$X(t) = x3*t^3 + x2*t^2 + x1*t + x0$$
$$Y(t) = y3*t^3 + y2*t^2 + y1*t + y0$$

Point	Value
t=0	*x0 * y0*, which gives w0 immediately
t=1/2	*(x3+2*x2+4*x1+8*x0) * (y3+2*y2+4*y1+8*y0)*
t=-1/2	*(-x3+2*x2-4*x1+8*x0) * (-y3+2*y2-4*y1+8*y0)*
t=1	*(x3+x2+x1+x0) * (y3+y2+y1+y0)*
t=-1	*(-x3+x2-x1+x0) * (-y3+y2-y1+y0)*
t=2	*(8*x3+4*x2+2*x1+x0) * (8*y3+4*y2+2*y1+y0)*
t=inf	*x3 * y3*, which gives w6 immediately

```
static void toom4_k2x2_mul(uint16_t ab[2*L], const uint16_t a[L], const uint16_t b[L])
{
  uint16_t tmpA[9*K];
  uint16_t tmpB[9*K];
  uint16_t eC[63*2*K];

  toom4_k2x2_eval_0(tmpA, a);
  toom4_k2x2_eval_0(tmpB, b);
  toom4_k2x2_basemul(eC+0*9*2*K, tmpA, tmpB);

  toom4_k2x2_eval_p1(tmpA, a);
  toom4_k2x2_eval_p1(tmpB, b);
  toom4_k2x2_basemul(eC+1*9*2*K, tmpA, tmpB);

  toom4_k2x2_eval_m1(tmpA, a);
  toom4_k2x2_eval_m1(tmpB, b);
  toom4_k2x2_basemul(eC+2*9*2*K, tmpA, tmpB);

  toom4_k2x2_eval_p2(tmpA, a);
  toom4_k2x2_eval_p2(tmpB, b);
  toom4_k2x2_basemul(eC+3*9*2*K, tmpA, tmpB);

  toom4_k2x2_eval_m2(tmpA, a);
  toom4_k2x2_eval_m2(tmpB, b);
  toom4_k2x2_basemul(eC+4*9*2*K, tmpA, tmpB);

  toom4_k2x2_eval_p3(tmpA, a);
  toom4_k2x2_eval_p3(tmpB, b);
  toom4_k2x2_basemul(eC+5*9*2*K, tmpA, tmpB);

  toom4_k2x2_eval_inf(tmpA, a);
  toom4_k2x2_eval_inf(tmpB, b);
  toom4_k2x2_basemul(eC+6*9*2*K, tmpA, tmpB);

  toom4_k2x2_interpolate(ab,eC);
}
```

code

```c
static inline void k2x2_eval(uint16_t r[9*K])
{
  /* Input:  e + f.Y + g.Y^2 + h.Y^3                          */
  /* Output: [ e | f | g | h | e+f | f+h | g+e | h+g | e+f+g+h ]  */

  size_t i;
  for (i=0; i<4*K; i++) {
    r[4*K+i] = r[i];
  }
  for (i=0; i<K; i++) {
    r[4*K+i] += r[1*K+i];
    r[5*K+i] += r[3*K+i];
    r[6*K+i] += r[0*K+i];
    r[7*K+i] += r[2*K+i];
    r[8*K+i] = r[5*K+i];
    r[8*K+i] += r[6*K+i];
  }
}
```

```c
static void toom4_k2x2_basemul(uint16_t r[18*K], const uint16_t a[9*K], const uint16_t b[9*K])
{
  schoolbook_KxK(r+0*2*K, a+0*K, b+0*K);
  schoolbook_KxK(r+1*2*K, a+1*K, b+1*K);
  schoolbook_KxK(r+2*2*K, a+2*K, b+2*K);
  schoolbook_KxK(r+3*2*K, a+3*K, b+3*K);
  schoolbook_KxK(r+4*2*K, a+4*K, b+4*K);
  schoolbook_KxK(r+5*2*K, a+5*K, b+5*K);
  schoolbook_KxK(r+6*2*K, a+6*K, b+6*K);
  schoolbook_KxK(r+7*2*K, a+7*K, b+7*K);
  schoolbook_KxK(r+8*2*K, a+8*K, b+8*K);
}
```

a13 구현

- Multiplication instruction

VMUL

Vector Multiply.

Syntax

`VMUL{cond}.datatype {Qd}, Qn, Qm`

`VMUL{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, `F32`, or `P8`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

PMUL (vector)

Polynomial Multiply.

Syntax

`PMUL Vd.T, Vn.T, Vm.T`

Where:

Vd

Is the name of the SIMD and FP destination register.

T

Is an arrangement specifier, and can be either `8B` or `16B`.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Usage

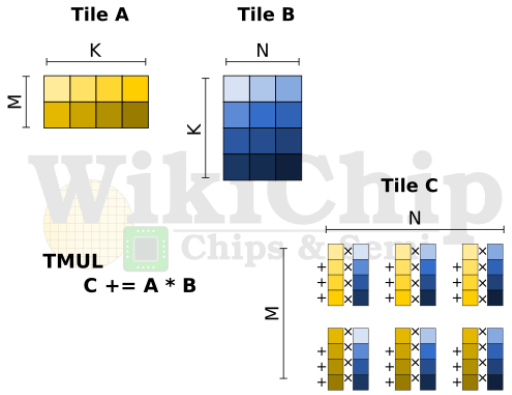
Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD and FP registers, places the results in a vector, and writes the vector to the destination SIMD and FP register.

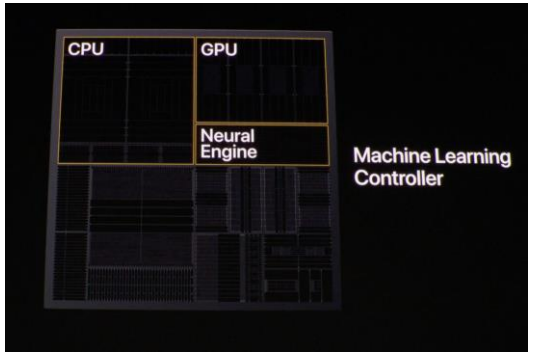
For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* in the *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

a13 구현

- AMX insturction Intel Advanced Matrix Extension (AMX)





Dougall
@dougallj

I started reversing Apple AMX, an undocumented 64-bit ARM instruction set extension for a matrix coprocessor, used by the Accelerate framework on the M1 (and A13+). My (incomplete) IDA/Hex-Rays plugin and notes so far:

aarch64_amx.py
GitHub Gist: instantly share code, notes, and snippets.
gist.github.com

Q & A