Vol.2022-SLDM-199 No.7
2022/11/11

高位合成を用いたGentleman-Sande型NTTの高速化

佐々木 智大[1,a)]　原 祐子[1,b)]

概要：耐量子暗号として注目されている格子暗号では，多項式乗算の処理がボトルネックである．この多項式乗算の計算量は，多項式の次数 n に対して通常の実装では $O(n^2)$ のところを，Number Theoretic Transform (NTT) を用いることで $O(n \log n)$ に削減できる．本研究は，高位合成を用いて格子暗号のアクセラレータを FPGA に実装する手法を確立することを目的とする．C 言語による格子暗号の実装に，ループ展開やパイプラインなどのプラグマを挿入し，柔軟に NTT アクセラレータを実現できる．高位合成を用いた NTT 実装の先行研究の多くは Cooley-Tukey 型の NTT（CT 型 NTT）を対象としているが，本研究では Gentleman-Sande 型の NTT（GS 型 NTT）を高位合成向けに実装し，既存実装の CT 型 NTT と比較する．また，多項式乗算において，NTT の処理と Inverse NTT の処理のいずれかを CT 型，もう一方を GS 型で実装することでビット反転のステップを省くことができることが知られている．本研究では，ビット反転を高位合成によって実装し，CT 型と GS 型を併用する場合のリソース削減量の見積もりを示す．

1. はじめに

現在，公開鍵暗号として用いられている RSA 暗号や楕円曲線暗号は，量子アルゴリズムである Shor のアルゴリズムによって多項式時間で解読できてしまうことが知られている．そのため 2016 年にアメリカ国立標準技術研究所 (National Institute of Standards and Technology; NIST) は耐量子暗号の標準化計画を立ち上げ，現在 4 段階目の審査が行われている[*1]．その候補として CRYSTALS-KYBER や SABER，Frodo といった Lerning With Error (LWE) 問題の困難性を利用した格子暗号ベースのアルゴリズムが複数挙げられている [1].

格子ベースの暗号・復号アルゴリズムでは，多項式乗算の計算量が主なボトルネックとなっている．多項式乗算に Number Theoretic Transform (NTT) を用いることで，計算量を $O(n^2)$ から $O(n \log n)$ に削減し，高速化することができる．近年は，NTT のアルゴリズムを対象に，FPGA のアクセラレータ実装を行うことで，さらなる高速化を目指す研究が行われている [2], [3].

NTT のアクセラレータ設計の研究初期は RTL 設計によるものが多かったが，設計生産性を上げ，最適化指示子で様々な構成を柔軟に探索するため，C ベース高位合成を使った研究に着目され始めている．しかし，既存手法の多くは CT 型 NTT を主に対象としており，GS 型 NTT の検討が未だ不十分である．本研究では GS 型 NTT を高位合成向けに実装し，CT 型 NTT との比較を行う．また，GS 型 NTT を活用することによりビット反転のステップを省くことができる [4] ため，本研究ではビット反転を高位合成によって実装し，削減することのできるリソース量を見積もる．

[1] 東京工業大学
Tokyo Institute of Technology
[a)] sasaki.c.ab@m.titech.ac.jp
[b)] hara@cad.ict.e.titech.ac.jp

[*1] https://csrc.nist.gov/projects/post-quantum-cryptography

2. 準備

2.1 Number Theoretic Transfrom (NTT)

Number Theoretic Transform (NTT) とは i 次の係数が $f_i \in \mathbb{Z}_q$ の n 次多項式に対して，フーリエ変換と同様の操作を行うもので，以下のように定義される．

$$\hat{f}_i = \sum_{j=0}^{n-1} f_j \omega_n^{ij} \pmod q$$

ただし n は 2 の冪で $i, j \in \mathbb{Z}$ かつ $\forall k \in \mathbb{Z}\ (1 \leq k < n), \omega_n^k \neq 1 \wedge \omega_n^n = 1$ である．また，NTT の逆変換である Inverse Number Theoretic Transform (INTT) は，NTT の各項に $\frac{1}{n}$ を掛け，ω_n^{ij} の Inverse をとることで定義される．

NTT は以下のように展開することができる．

$$\hat{f}_i = \sum_{j=0}^{n/2-1} f_{2j} (\omega^2)^{ij} + \omega^i \sum_{j=0}^{n/2-1} f_{2j+1} (\omega^2)^{ij}$$

$$\hat{f}_{i+n/2} = \sum_{j=0}^{n/2-1} f_{2j} (\omega^2)^{ij} - \omega^i \sum_{j=0}^{n/2-1} f_{2j+1} (\omega^2)^{ij}$$

よって，一項目の総和と二項目の総和の部分が次数 $n/2$ の

Algorithm 1 NTT

Require: $\forall i\ (0 \leq i < n), a[i] \in \mathbb{Z}_q$
Require: $\omega_n \in \mathbb{Z}_q$, $n = 2^l$. ただし ω_n は q の n 乗根.
Ensure: $\hat{a} = NTT(a)$.

```
1: â ← bitreverse(a)
2: for i = 1 : i < l : i++ do
3:     m ← 2^{l−i}
4:     ω_m ← ω_n^{n/m}
5:     for j = 0 : j < n : j++ do
6:         ω ← 1
7:         for k = 0 : k < m/2 : m++ do
8:             t_1 ← ω · â[k + j + m/2] (mod q)
9:             t_2 ← â[k + j]
10:            â[k + j] ← t_1 + t_2 (mod q)
11:            â[k + j + m/2] ← t_1 − t_2 (mod q)
12:            ω ← ω · ω_m (mod q)
13:        end for
14:    end for
15: end for
```

NTT となっており，これを再帰的に行うことで NTT の計算量は $O(n \log n)$ となる．

2.2 NTT を用いた多項式乗算

NTT を用いた n 次多項式 f と g の乗算は

$$f \cdot g = INTT(NTT(f) * NTT(g))$$

で実行される．NTT, INTT は前項で述べた通り $O(n \log n)$ なので，多項式乗算全体の計算量も $O(n \log n)$ となる．

3. NTT の実装

3.1 既存研究における NTT の実装

通常 NTT のソフトウェア実装では Algorithm 1 のようなループ構造で実装する．この実装では，最内ループのトリップカウントが m の値によって変わる．一方，高位合成は，ループ展開やパイプライン化などの最適化プラグマでループ最適化を行う場合，トリップカウントは一定である必要がある．そのため高位合成向けにループ構造を工夫する必要があり，高位合成向け CT 型 NTT の実装が [2] や [3] で提案されている．[2] では最内ループの回数を B というパラメータで定めて，トリップカウントを一定にしている．[3] では [2] の実装のパラメータ B を 2 に固定し，ループ依存性を取り除くプラグマを使用することでさらにレイテンシを削減している．

3.2 Gentleman-Sande 型 NTT の実装

本研究では，[3] をもとに Gentleman-Sande 型 NTT (GS 型 NTT) を Algorithm 2 に示す通り実装した．また，プラグマの構成は表 1 のように設定した．ループ構造は [3] に従い，最内ループはコンパイラが最適化しやすいように，インデックス計算をする IDX LOOP，メモリ読み出しをする MEM READ LOOP，実際の演算を一次保存配列に保存する OP LOOP，メモリ書き込みをする MEM WRITE LOOP に分けている．

表 1 プラグマの構成

擬似コード	高位合成のプラグマ
$i_e, i_o, i_w, w, U, V, W, E, O$	array_partition complete
$\hat{a}$	array_partition block factor=8
BUTTERFLY_LOOP	pipeline
most inner loops	unroll
BUTTERFLY_LOOP: $\hat{a}$	dependence inter false

表 2 高位合成結果

	CT 型 [3]	GS 型（2 章）
レイテンシ	2,114	2,094
DSP48E	2	2
FF	9,276	8,568
LUT	7,351	7,426

GS 型 NTT と CT 型 NTT の違いは，バタフライ演算の順序である．具体的には IDX LOOP 内のインデックス計算と，OP LOOP の演算の内容に相当する．[3] では IDX LOOP 内で i だけ左巡回シフトしているのに対し，Algorithm 2 では GS 型 NTT のバタフライ順序にするため $l - i + 1$ だけ左巡回シフトしている．また，OP LOOP 内では GS 型 NTT のバタフライ順序に合わせて演算を行っている．

3.3 ビット反転の実装

NTT と INTT のいずれかを CT 型，もう一方を GS 型にすることでビット反転のステップを省略できる [4]．通常のビット反転は，事前計算の結果をテーブルに保存する方法で実装されるが，ビット数が 2 の冪乗の場合は，ビット反転にマスキングを利用した逐次計算で実現することができる．前者は事前計算でテーブルを $T[N]$（Nbit のビット反転）として，$T[i]$ に i をビット反転したものを格納しておくことで実装できる．後者の実装方法を Algorithm 3 に示す．

4. 実験結果

4.1 CT 型 NTT と GS 型 NTT の比較

既存手法の CT 型 NTT 実装 [3] と 2 章で述べた GS 型 NTT 実装を Xilinx 社 Vivado HLS 2018.3 を用いて高位合成を行った．FPGA は ZYNQ-7 ZC702 を指定し，周波数制約は 100MHz とした．パラメータは $B = 2, q = 3329, n = 256$ と設定した．

まず，CT 型 NTT と GS 型 NTT の合成結果を比較する．それぞれのレイテンシおよび FPGA のリソース使用量を表 2 に示す．CT 型 NTT と GS 型 NTT を比較すると，GS 型 NTT は CT 型 NTT とほぼ同じレイテンシ，リソー

Algorithm 2 Gentleman-Sande 型 NTT

Require: $\omega[i] = \omega^i \pmod q$, $(0 \le i \le n/2)$.
Ensure: $\hat{a} = NTT(a)$

```
STAGE LOOP:
for i = 0 : i < l : i++ do
    BUTTERFLY LOOP:
    for s = 0 : s < n/2 : s = s + b do
        IDX LOOP:
        for b = 0 : b < B : b++ do
            i_e[b] ← rotateLeft(((s + b) << 1) + 0, l − i + 1)
            i_o[b] ← rotateLeft(((s + b) << 1) + 1, l − i + 1)
            i_w[b] ← (1 << i) + ((s + b) >> i)
            parity[b] ← calc_parity(i_e[b])
        end for
        MEM READ LOOP:
        for b = 0 : b < B : b++ do
            U[b] ← â[i_e[b] >> 1][not(parity[b])]
            V[b] ← â[i_o[b] >> 1][parity[b]]
            W[b] ← ω[i_w[b]]
        end for
        OP LOOP:
        for b = 0 : b < B : b++ do
            E[b] ← (U[b] + V[b]) (mod q)
            O[b] ← (U[b] − V[b]) · W[b] (mod q)
        end for
        MEM WRITE LOOP:
        for b = 0 : b < B : b++ do
            â[i_e[b] >> 1][not(parity[b])] ← E[b]
            â[i_o[b] >> 1][parity[b]] ← O[b]
        end for
    end for
end for
```

Algorithm 3 8 ビットのビット反転

Require: n(n は 8bit の正の整数)
Ensure: n のビット反転を計算.

```
t ← n
n ← n & (0x55555555)
t ← t ⊕ n
n ← n << 2
n ← n|t
t ← n
n ← n & (0x33333333 << 1)
t ← t ⊕ n
n ← n << 4
n ← n|t
t ← n
n ← n & (0x0f0f0f0f << 3)
t ← t ⊕ n
n ← n << 8
n ← n|t
n ← n >> 7
```

ス量で実現できる．よって，3.3 章で述べた手法による多項式乗算で，GS 型 NTT を使用することによるデメリットは発生しない．若干のリソース差は Algorithm 2 の OP LOOP 部分の計算ステップの差によるものと考えられる．

表 3 ビット反転の高位合成結果

	8 ビット (Algorithm 3)	8 ビット	9 ビット
レイテンシ	不定	30	89
FF	57	1,165	3,202
LUT	172	2,169	5,379

4.2 ビット反転ステップの見積もり

本研究では，8 ビットのビット反転 (Algorithm 3) と，事前計算テーブルを利用したビット反転で 8 ビットと 9 ビットのビット反転 を実装し，それを使った **bitreverse** に高位合成を行った．その結果を表 3 に示す．8 ビットなど 2 の冪乗ビットでのビット反転の場合，リソース使用量を大幅に削減できるが，トリップカウントは一定ではない．一方，任意ビットの場合（本実験では 8 ビットと 9 ビットの場合），ビット数に比例して回路規模が肥大化する．本実験結果から，CT 型と GT 型を併用することでこれらのビット反転操作自体が不要になるため，回路規模やレイテンシの増加を大幅に削減するできることが分かる．

5. おわりに

本研究では，GS 型 NTT アクセラレータを高位合成を用いて実装し，既存研究によって実装されていた CT 型 NTT アクセラレータと比較を行った．多項式乗算に対して NTT と INTT のいずれかを GS 型に置き換えても，リソース量の肥大化やレイテンシの増加が起きないことを示した．また，ビット反転も高位合成で実装することで，CT 型 NTT と GS 型 NTT を併用した際のリソース削減の見積もりを示した．今後の課題として，ARM のプロセッサを用いたソフトウェア実装に対する，NTT の FPGA アクセラレータの高速化の効果を定量的に示すことが挙げられる．

謝辞

本研究は，JSPS 科研費 JP20H00590 の支援を受けたものである．

参考文献

[1] T. Zijlstra, K. Bigou, A. Tisserand, “Lattice-based Cryptosystems on FPGA: Parallelization and Comparison using HLS,” *IEEE Transactions on Computers*, 2021.
[2] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas¸, and A. Aysu, “An extensive study of flexible design methods for the Number Theoretic Transform,” *IEEE Transactions on Computers*, 2020.
[3] A. El-Kady, P. Fournaris, T. Tsakoulis, E. Haleplidis, V. Paliouras, “High-Level Synthesis design approach for Number-Theoretic Transform Implementations,” *International Conference on Very Large Scale Integration*, 2021.
[4] P. Longa, M. Naehrig, “Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography,” *International Conference on Cryptology and Network Security*, 2016.