CKKS方式準同型暗号におけるRescale演算のGPU実装と演算性能評価

井上紘太朗† 山田 健太†† 鈴木 拓也†† 石巻 優†† 山名 早人†††

† 早稲田大学 基幹理工学部 〒169–8555 東京都新宿大久保3-4-1
†† 早稲田大学 基幹理工学研究科 〒169–8555 東京都新宿大久保3-4-1
††† 早稲田大学 理工学術院 〒169–8555 東京都新宿大久保3-4-1
E-mail: †{kinoue,yamada,t-suzuki,yuishi,yamana}@yama.info.waseda.ac.jp

あらまし 近年，クラウドサービスの普及により，データの収集から活用までをクラウド上で完結させることが可能となった．しかし，クラウド上に保管されたデータは常に漏洩のリスクに晒されており，個人情報保護という観点で大きな問題となっている．これらのデータを安全に利活用するため，近年注目されている技術が準同型暗号である．準同型暗号は，暗号化した状態での演算が可能な反面，処理速度が遅いという問題を抱えている．準同型暗号の各演算を高速化する取り組みの一つとして，GPUやFPGAなどの外部機器へ処理を委譲する手法が提案されている．本研究では，準同型暗号の方式の一つであるCKKS方式に着目する．CKKS方式は，近似により準同型暗号上で実数を扱うことができ，機械学習分野への応用が期待されている．CKKS方式のGPU実装はすでに提案されているが，実装が未公開であり，十分な検証を行うことができない．本研究では，CKKS方式をナイーブにGPUへ実装し，実行速度の測定を行う．特に，CKKS方式において特徴的な*Rescale*と呼ばれる演算に着目し，GPU上での実装を行った．その後，CPU実装との性能比較を行い，実行速度において約20〜40倍程度の性能低下が確認された．これは，64bit整数演算のエミュレーションとグローバルメモリアクセスのオーバヘッドによるものと考えられ，今後，GPU上の高速なメモリ領域の活用を中心とした高速化を行っていく．

キーワード 準同型暗号，高速化，GPGPU

1 はじめに

近年，計算資源を自社で保有するオンプレミスではなく，インターネットを介して他者から提供される計算資源を用いたクラウドサービスが盛んに用いられている．しかし，クラウドを活用するにあたり，クラウド上に保管されているデータの取扱いが問題視されている．欧州では，GDPR（一般データ保護規則）[1]が施行され，企業がサービスにおける個人情報の取扱いについて再考する大きな要因となった．欧州に限らず，国内においても個人情報保護に関する法律が整備され[2]，カリフォルニア州でも2020年1月からCCPA（カリフォルニア州 消費者プライバシー法）[3]が施行されるなど，社会の個人情報保護に対する意識はより一層高まっている．社会意識が高まる中，近年データを保護した状態で利活用するための技術として注目されているものが，準同型暗号である．なお，ここで個人情報を暗号化しても個人情報であることには変わりないものの，GDPR等においては，個人情報を扱う上で暗号化することが推奨されている．

準同型暗号とは，データを暗号化した状態で演算を行うことができる技術である．特に，Gentryが2009年に発表した方式[4]は，従来の準同型暗号に存在した演算回数の制約をなくし，完全準同型暗号と呼ばれている．Gentryが完全準同型暗号に関する論文を発表して以来，準同型暗号の分野は，より盛んに研究が行われるようになった．特に，Cheonらが2017年に提案したCKKS方式[5]は，近似を用いて暗号文上での実数や複素数に対する演算を可能とし，機械学習の分野での応用が期待されている．しかし，準同型暗号の演算は，時間計算量と空間計算量ともに膨大であり，実用化という面で大きな障害となっている．この問題を解決するため，既存の方式を様々なアプローチで改善する研究が行われている．特に，RNS（剰余体系）と呼ばれる剰余表現[6]を用いると，準同型暗号の各演算の並列度を上げることができる[7], [8]．この並列化可能性を活かし，多くの方式について，準同型暗号の処理の一部[9]，あるいは全て[10], [11]を，GPUやFPGA等のハードウェアアクセラレータ上で行うアプローチが取られている．CKKS方式についても，FPGAを用いた高速化[9]については，すでに提案されている．しかし，GPUについては，現時点において広く公開された実装が存在していない．

CKKS方式では，「スケール」と呼ばれるパラメータを用いて，実数を表現している．スケールの値は，準同型乗算を行うたびに指数的に増加する．*Rescale*は，準同型乗算により増加したスケールを，適切に調整する関数である．*Rescale*は，準同型乗算を行う度に呼び出されるため，高速化による効果が大きい．

本研究では，CKKS方式に特徴的な演算である*Rescale*に着目し，*Rescale*をナイーブにGPU上へ実装した際，どの程度の実行速度の改善が見られるか，検証を行うことを目的とする．参考実装としてMicrosoft SEAL[12]を使用し，*Rescale*

と，*Rescale* に必要な各種関数を GPU 上に移植する．

本稿は，以下の構成をとる．2 節では準同型暗号の概要について述べる．続く 3 節では，CUDA を活用した GPU 上での汎用計算について述べる．その後，4 節では，関連研究を述べ，5 節で提案手法について説明する．そして，6 節で実験方法と評価について述べ，7 節でまとめる．

2 準同型暗号

本節では，準同型暗号について述べる．準同型暗号とは，データを暗号化した状態で演算を行い，復号時に正しい演算結果が得られるように構成された暗号方式である．例えば，平文 $m1$，$m2$ があったとき，ある演算 $\cdot$ について，式 (1) の関係が成り立つような暗号方式である．なお，本稿で扱う CKKS 方式 [5] については，復号結果は一定の誤差を含む近似値であることに注意する．ただし，$Enc(m)$ は，平文 m を暗号化することにより得られる暗号文であり，$Dec(Enc(m))$ は，暗号文 $Enc(m)$ を復号して得られる平文を指す．

$$m1 \cdot m2 \approx Dec(Enc(m1) \cdot Enc(m2)) \tag{1}$$

本研究では，準同型暗号のライブラリとして，RNS 型 [6] の CKKS 方式が実装されている Microsoft SEAL [12] を用いる．ゆえに，本稿では，特に断りがない限り，Microsoft SEAL の実装を前提として説明する．また，記号については，原則 Cheon らの論文 [7] の表記に従う．

2.1 記号の定義

対数の底は，特に断りのない限り 2 とする．N を 2 べきの整数とし，$K = \mathbb{Q}[X]/(X^N + 1)$ を $2N$-円分体，およびその整数環を $R = \mathbb{Z}/(X^N + 1)$ とする．R 上の任意の元は，たかだか $(N-1)$ 次の多項式である．また，整数 q を法とする R の剰余環 R/qR を R_q と表記する．

多項式は，係数を要素とするベクトルで表される．ベクトルは，$\mathbf{a}$ のように太字で表記し，特に断りのない限り，行ベクトルとする．ベクトルの i 番目の要素は，$\mathbf{a}[i]$ と表記する．

整数 q について，$\mathbb{Z} \cap (-q/2, q/2]$ を $\mathbb{Z}_q$ と表記する．ただし，実装上は $\mathbb{Z}_q$ を $\mathbb{Z} \cap [0, q)$ としている．q を法として，a を Z_q の範囲へ換算する，すなわち剰余をとる演算を $[a]_q$ と表記する．なお，これらの演算は，多項式に対して係数ごとに行われる．

2.2 NTT(数論変換)

NTT は，特殊な法を用いた FFT（高速フーリエ変換）である [13]．FFT とは，DFT（離散フーリエ変換）を高速に計算するアルゴリズムである．多項式環上での乗算，すなわち多項式同士の乗算には，次数を N として $\mathcal{O}(N^2)$ の時間計算量を要する．多項式同士の乗算は，多項式の係数列同士の畳み込みとみなせる．係数列畳み込みの時間計算量も，多項式同士の乗算同様に $\mathcal{O}(N^2)$ である．

一方，多項式の係数列を NTT 形式に変換した場合，多項式同士の乗算（係数列の畳み込み）を $\mathcal{O}(N \log N)$ で得ることができる．RNS 型 CKKS 方式においても，NTT を用いて乗算の計算コストを抑えている [7]．

R_q 上の多項式の係数ベクトル $\mathbf{a} \in \mathbb{Z}_q^n$ の NTT 形式を $\tilde{\mathbf{a}}$ と表記する．$\mathbf{a}$ を $q \equiv 1 \mod 2N$ を満たす素数 q を法として NTT 形式 $\tilde{\mathbf{a}}$ へ変換する関数を $\mathrm{NTT}(\mathbf{a}, q)$ と定義する．また，$\mathrm{NTT}(\mathbf{a}, q)$ の逆変換を $\mathrm{INTT}(\tilde{\mathbf{a}}, q)$ と定義する．

2.3 RNS（剰余数系）

有限個の整数からなる順序集合 $\mathcal{B} = \{p_0, ..., p_{k-1}\}$ で，要素同士が互いに素である場合，$\mathcal{B}$ を基底と呼ぶ．さらに，$P = \prod_{i=0}^{k-1} p_i$ とおく．$[\cdot]_{\mathcal{B}} : \mathbb{Z}_p \mapsto \prod_{i=0}^{k-1} \mathbb{Z}_{p_i}$ を，$a \mapsto [a]_{\mathcal{B}} = ([a]_{p_i})_{0 \leqq i < k}$ と置いた時，$[a]_{\mathcal{B}}$ を，$a \in \mathbb{Z}_P$ の RNS 表現と呼ぶ．つまり，RNS を用いることで，$\mathbb{Z}_p$ 上の演算を，よりサイズの小さい Z_{p_i} に分割して計算を行うことができる．

2.4 CKKS 方式

RNS 型の CKKS 方式について述べる．CKKS 方式は，近似を用いることで，準同型暗号上で複素数や実数を扱うことのできる方式である．

CKKS 方式では，計算効率を上げるためにパッキングという手法を用いている．パッキングとは，複数の値を，一つの平文として扱う手法である．CKKS 方式において，パッキングを行う演算を *Encode*，およびその逆演算を *Decode* と呼ぶ．

CKKS 方式は，固定小数点演算を暗号文上で行うことを想定した方式である．CKKS 方式では，演算の精度を保証するため，スケール値 Δ を平文に掛けることによって，固定小数点として扱っている．スケール値については，2.5 で詳しく述べる．

- $Encode(\mathbf{z}; \Delta)$．(N/2) 個の複素数からなる配列 $\mathbf{z} \in \mathbb{C}^{N/2}$ を入力にとり，各要素へ Δ でスケーリングを行った後，パッキングを行う．パッキングされた平文を $\mathbf{m}$ とする．
- $Decode(\mathbf{m}; \Delta)$．パッキングされた平文 $\mathbf{m}$ を入力にとり，各要素をスケール値 Δ で割り，$\mathbf{z}$ を返す．

RNS 型 CKKS 方式の演算を以下に示す．まず，CKKS 方式を使用するにあたって必要なパラメータの設定を行う．

- $Setup(q, L, \eta; 1^\lambda)$.

セキュリティパラメータ [1]λ として，q, L, η を与え，それぞれ整数，回路のレベル，精度（単位は bit）とする．回路のレベルとは，準同型乗算を評価できる回数を示している．

- $q_j/q \in (1 - 2^{-\eta}, 1 + 2^{-\eta}]$ $(1 \leqq j \leqq L)$ を満たすように，RNS の基底 $\mathcal{D} = \{p_0, ..., p_{k-1}, q_0, ..., q_L\}$ をとる．ここで，$0 \leqq l \leqq L$ について，$\mathcal{B} = \{p_0, ..., p_{k-1}\}, \mathcal{C}_l = \{q_0, ..., q_l\}, \mathcal{D}_l = \mathcal{B} \cup \mathcal{C}_l$ と表記する．また，$P = \prod_{i=1}^{k-1} p_i, Q_l = \prod_{i=0}^{l} q_i$ とする．
- 2 のべき乗であるような整数 N を選ぶ．

ここで，平文を $\mathbf{m}$，暗号文を $\mathbf{ct} = \left(\mathbf{ct}^{(j)} = (\mathbf{c}_0^{(j)}, \mathbf{c}_1^{(j)})\right)_{0 \leqq j \leqq l} \in \prod_{j=0}^{l} R_{q_j}^2$ と表記する．ここで，法が Q_l である暗号文を，レベル l の暗号文と呼ぶ．

- $KeyGen$.

公開鍵 pk，秘密鍵 sk，評価鍵 evk を生成.

- $Enc(\mathrm{pk}, \mathbf{m})$.

1：セキュリティパラメータとは，暗号強度についての指標である．

公開鍵 pk と平文 $\mathbf{m}$ をとり，暗号文 $\mathbf{ct}$ を返す．

- $Dec(\mathrm{sk}, \mathbf{ct})$.

秘密鍵 sk と暗号文 $\mathbf{ct}$ をとり，平文 $\mathbf{m}$ を返す．

- $Add(\mathbf{ct}, \mathbf{ct}')$.

暗号文 $\mathbf{ct}, \mathbf{ct}'$ をとり，暗号文 $\mathbf{ct}_{add} = \left(\mathbf{ct}_{add}^{(j)}\right)_{0 \leqq j \leqq l}$ を返す．ただし，$\mathbf{ct}_{add}^{(j)} \leftarrow \mathbf{ct}^{(j)} + \mathbf{ct}'^{(j)} \pmod{q_j}$.

- $Mult(\mathbf{ct}, \mathbf{ct}')$.

暗号文 $\mathbf{ct} = \left(\mathbf{ct}^{(j)} = (\mathbf{c}_0^{(j)}, \mathbf{c}_1^{(j)})\right)_{0 \leqq j \leqq l} \in \prod_{j=0}^{l} R_{q_j}^2$，$\mathbf{ct}' = \left(\mathbf{ct}'^{(j)} = (\mathbf{c}'^{(j)}_0, \mathbf{c}'^{(j)}_1)\right)_{0 \leqq j \leqq l} \in \prod_{j=0}^{l} R_{q_j}^2$ をとり，$\mathbf{ct}_{mult} = \left(\mathbf{ct}^{(j)} = (\mathbf{d}_0^{(j)}, \mathbf{d}_1^{(j)}, \mathbf{d}_2^{(j)})\right)_{0 \leqq j \leqq l} \in \prod_{j=0}^{l} R_{q_j}^3$ を返す．ただし，$\mathbf{ct}_{mul}$ は以下のように計算される．

$$
\begin{aligned}
\mathbf{d}_0^{(j)} &\leftarrow \mathbf{c}_0^{(j)} \mathbf{c}'^{(j)}_0 \pmod{q_j} \\
\mathbf{d}_1^{(j)} &\leftarrow \mathbf{c}_0^{(j)} \mathbf{c}'^{(j)}_1 + \mathbf{c}_1^{(j)} \mathbf{c}'^{(j)}_0 \pmod{q_j} \\
\mathbf{d}_2^{(j)} &\leftarrow \mathbf{c}_1^{(j)} \mathbf{c}'^{(j)}_1 \pmod{q_j}
\end{aligned}
$$

- $Relinearlization(\mathrm{evk}, \mathbf{ct}_{mult})$.

評価鍵 evk と，乗算結果として得られた暗号文 $\mathbf{ct}_{mult}$ をとり，$\mathbf{ct}' = \left(\mathbf{ct}'^{(j)} = (\mathbf{d}'^{(j)}_0, \mathbf{d}'^{(j)}_1)\right)_{0 \leqq j \leqq l} \in \prod_{j=0}^{l} R_{q_j}^2$ を返す．

- $Rescale(\mathbf{ct})$.

暗号文 $\mathbf{ct} = \left(\mathbf{ct}^{(j)} = (\mathbf{c}_0^{(j)}, \mathbf{c}_1^{(j)})\right)_{0 \leqq j \leqq l} \in \prod_{j=0}^{l} R_{q_j}^2$ をとり，$\mathbf{c}'^{(j)}_i \leftarrow q_l^{-1} \cdot \left(\mathbf{c}_i^{(j)} - \mathbf{c}_i^{(l)}\right) \pmod{q_j}$ を計算する．結果として，$\mathbf{ct}' = \left(\mathbf{ct}'^{(j)} = (\mathbf{c}'^{(j)}_0, \mathbf{c}'^{(j)}_1)\right)_{0 \leqq j \leqq l-1} \in \prod_{j=0}^{l-1} R_{q_j}^2$ を返す．

2.5 Rescale

CKKS 方式の演算のうち，*Rescale* について詳しく述べる．*Rescale* とは，レベルが l であるような $\mathbf{m}$ の暗号文に対して，レベルが $l-1$ であるような $q_l^{-1} \cdot \mathbf{m}$ の暗号文を返す関数である．CKKS 方式では，固定小数点を整数係数多項式で表現するため，スケーリングを行っている．例えば，2.4 を 16bit のスケールで表現すると，2.4 を 16bit 左シフト，すなわち $2.4 \cdot 2^{16} \approx 157286$ となる．逆に，スケーリングされた値から元の値を得る場合は，16bit 右シフト，すなわち $157286/2^{16} \approx 2.4$ となる．CKKS 方式では，スケールの値によって，結果の精度を調整することが可能となる．しかし，スケールの値は，準同型乗算を行うごとに，指数的に増加するため，演算の前後で適切に調整されなければならない．そこで，CKKS 方式では，*Rescale* を用いて，準同型乗算により増加したスケールを適切に調整している．例えば，z_1, z_2 を固定小数点で表現された実数，q を整数とする．CKKS 方式において，z_1, z_2 を暗号化する際，平文としてそれぞれ $q \cdot z_1, q \cdot z_2$ が用いられる．ここで，q がスケールを表している．$q \cdot z_1$ と $q \cdot z_2$ の暗号文同士で準同型乗算を行うと，結果は $q^2 \cdot z_1 z_2$ となり，スケールは q^2 となる．ここで，乗算結果に対して *Rescale* を行うことにより，暗号文のスケールを，演算前のスケールである q へ削減することができる．

アルゴリズム 2.1 は，SEAL 内部で実装されている *Rescale* のアルゴリズムである．ただし，説明の都合上，$0 \leq j \leq N-1$ の範囲について，一時変数 $\mathbf{u}, \mathbf{t}$ を，それぞれ $\mathbf{u} = (\mathbf{u}_0, \mathbf{u}_1) \in R_{q_j}^2, \mathbf{t} \in R_{q_l}$ とする．また，剰余演算については，除算を使わず高速に剰余演算を行うことができる Barrett 法 [14] を用いている．

アルゴリズム 2.1 Rescale

Input: $\tilde{\mathbf{ct}} = \left(\tilde{\mathbf{ct}}^{(j)} = (\tilde{\mathbf{c}}_0^{(j)}, \tilde{\mathbf{c}}_1^{(j)})\right)_{0 \leqq j \leqq l} \in \prod_{j=0}^{l} R_{q_j}^2$

Output: $\tilde{\mathbf{ct}}' = \left(\tilde{\mathbf{ct}}'^{(j)} = (\tilde{\mathbf{c}}'^{(j)}_0, \tilde{\mathbf{c}}'^{(j)}_1)\right)_{0 \leqq j \leqq l-1} \in \prod_{j=0}^{l-1} R_{q_j}^2$

for $i = 0$ to 1 **do**
  for $j = 0$ to l **do**
    $\mathbf{c}_i^{(j)} \leftarrow \mathrm{INTT}(\tilde{\mathbf{c}}_i^{(j)}, q_j)$ ▷ NTT 形式から逆変換する
  end for
end for
for $i = 0$ to 1 **do**
  for $j = 0$ to N **do**
    $\mathbf{t}[j] \leftarrow \left[\mathbf{c}_i^{(l)}[j] + q_l/2\right]_{q_l}$
  end for
  for $j = 0$ to $l-1$ **do**
    $\mathbf{u}_i \leftarrow [\mathbf{t}]_{q_j}$
    for $k = 0$ to $N-1$ **do**
      $\mathbf{u}_i[k] \leftarrow [\mathbf{u}_i[k] - q_l/2]_{q_j}$
    end for
    $\mathbf{c}'^{(j)}_i \leftarrow \mathbf{c}_i^{(j)} - \mathbf{u}_i \pmod{q_j}$
    $\mathbf{c}'^{(j)}_i \leftarrow q_l^{-1} \cdot \mathbf{c}'^{(j)}_i \pmod{q_j}$
  end for
end for
for $i = 0$ to 1 **do**
  for $j = 0$ to l **do**
    $\tilde{\mathbf{c}}_i^{(j)} \leftarrow \mathrm{NTT}(\mathbf{c}_i^{(j)}, q_j)$ ▷ NTT 形式に戻す
  end for
end for

3 GPU の構造と CUDA

本節では，実験で使用する NVIDIA 製 GPU のアーキテクチャと，NVIDIA 製 GPU で GPGPU（GPU による汎用計算）を行う際のツールである CUDA について述べる．

3.1 GPU の構造 [15]

GPU は，複数の SM（ストリーミング・マルチプロセッサ）と，各 SM からアクセスできるグローバルメモリから構成されている．各 SM は複数の CUDA コアとレジスタ，キャッシュ，共有メモリから構成されている．共有メモリは，SM 内の CUDA コアからのみアクセス可能なメモリ領域である．また，GPU 側からの読み出し専用ではあるが，高速にアクセスが可能なコンスタントメモリと呼ばれる領域が存在する．

3.2 CUDA プログラミングモデル [16]

CUDA とは，NVIDIA 社が提供している，NVIDIA 製 GPU 向けの汎用的な並列計算基盤，およびプログラミングモデルである．CUDA は，C を拡張した「CUDA C」という専用の言語と，nvcc と呼ばれる専用のコンパイラを用いて開発を行う．CUDA では，GPU 上で実行される関数を「カーネル」と

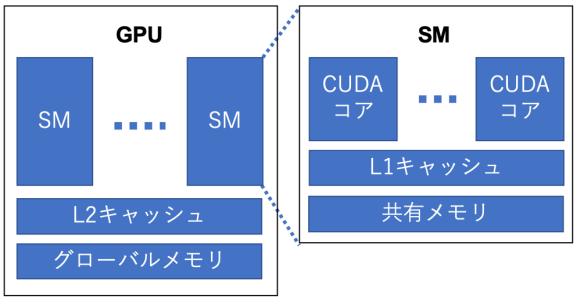

図 1　GPU のアーキテクチャ

呼ぶ．カーネルは，呼び出し時に指定した並列度で実行される．CUDA において，最小の実行単位は「スレッド」である．スレッドは，「ブロック」という実行単位にまとめられ，各 SM 上で実行される．ブロックは，さらに「グリッド」と呼ばれる単位でまとめて管理されている．このグリッドが，CUDA におけるカーネルにあたる．各 SM に割り当てられたブロック内のスレッドは，32 スレッドごとに「ワープ」という単位にまとめられ，命令レベルで同時に実行される．各ワープは，ワープスケジューラによってスケジュールされ，実行される．

4　関 連 研 究

本節では，ハードウェアアクセラレータを用いた準同型暗号の実装について，先行研究を紹介する．先行研究では，ハードウェアアクセラレータとして GPU や FPGA を採用し，準同型暗号の各種演算を実装している．

4.1　GPU を用いたもの

準同型暗号の GPU 実装は，2012 年に Wang らによって初めて提案された [17]．Wang らは，Gentry-Halevi 方式 [18] を GPU 上で実装し，CPU 実装と比較して 7 倍程度の性能向上を実現した．以降，準同型暗号の様々な方式に対して，GPU 実装による高速化に関する研究が盛んに行われている．

CKKS 方式の GPU 実装は，Badawi らによって，2019 年に初めて提案されている [19]．Badawi らは，Badawi らが提案した FV 実装 [11], [20] に，CKKS 方式に特有の演算である *Rescale* を加えることによって，CKKS 方式の GPU 実装を行っている．一方，CKKS 方式特有の演算のうち，*Encode* と *Decode* については，計算量的に低コストであるとして，GPU による実装を行っていない．

4.2　GPU 以外を用いたもの

GPU 以外を用いた研究として，Riazi らの研究 [9] を紹介する．これは，ハードウェアアクセラレータとして，FPGA を用いた研究である．準同型暗号の方式として，筆者らと同様に RNS 型の CKKS 方式を採用している．Riazi らの研究では，特に計算量が大きい演算として，準同型乗算と *Relinearization*，*Rotaion* を挙げ，それぞれ FPGA 上で実装を行っている．また，これらの演算を実装するにあたって必要な，多項式同士の乗算を行う MULT モジュール，NTT と INTT を行うモジュール，*Modulus Switching* を行うモジュールをそれぞれ FPGA 上へ実装している．一方，CKKS 方式の演算のうち，鍵生成や *Encode*・*Decode*，暗号化・復号については，前述の演算と比較して計算量的に低コストであるとしている．ゆえに，これらの演算については，FPGA 上での実装を行っていない．

5　Rescale 処理の GPU 実装提案

本節では，CKKS 方式のうち，*Rescale* の GPU 実装を提案する．Badawi らの CKKS 方式の実装 [19] は，実装の詳細については公開されていない．本研究では，既存の CPU 実装をもとに，実際に GPU 上へナイーブに実装することにより，実行速度の改善が可能かどうかを検証する．なお，本稿において，「ホスト」と「デバイス」は，それぞれ CPU と GPU を指す．

GPU 上で実装を行うにあたり，参考実装の選定を行った．CKKS 方式を実装したライブラリとして，HEAAN[2] や HElib[3]，Microsoft SEAL が挙げられる．本研究では，参考実装として Microsoft SEAL を選択した．HEAAN や HElib は，多倍長整数演算の実装を NTL[4] に依存しているため，GPU への移植に不向きである．対して，Microsoft SEAL は，外部ライブラリに依存していないため，GPU への移植が容易であると判断した．

GPU 上で実行できるよう，CUDA を用いて実装を行った．*Rescale* の処理中に呼ばれるすべての関数は GPU 上で実行され，ホストとデバイス間で無駄な通信が発生しないよう実装を行った．*Rescale* の実装に伴い，GPU 上への実装が必要な処理を以下に示す．

- NTT，および INTT
- Barrett 法
- mod 上での加算，減算，乗算

Rescale 処理全体と NTT，および INTT は，それぞれ独立の CUDA カーネルとして実装を行った．対して，Barret 法や mod 上の演算については，デバイス側から呼び出す関数として実装を行った．

さらに，各カーネル内は，各多項式ごとにスレッドを割り当てた．すなわち，並列度は，回路のレベルを L とすると，$2 * L$ である．

5.1　計算の流れ

一連の処理の流れを以下に示す．また，処理全体の概要図を図 2 に示す．デバイスで実際に実行される処理は (iii) のみであり，その他の処理についてはホスト上で実行される．また，(i) は，SEAL によって行われる．事前計算された値や，計算対象の暗号文については，SEAL 内部で定義された独自データ型で表現されている．デバイス上で計算を行う際，これらの独自データ型から，対応する C++のプリミティブ型へ変換を行う必要がある．プリミティブ型へ変換されたデータは，デバイス

2：https://github.com/snucrypto/HEAAN
3：https://github.com/homenc/HElib
4：https://www.shoup.net/ntl/

上のメモリへ転送され，*Rescale* 処理が実行される．得られた計算結果は，デバイス上のメモリからホスト上のメモリへ転送される．

(i) 以下の値を事前計算
- NTT，および INTT で用いる定数
- Barrett 法で用いる定数

(ii) 以下の値を，ホストからデバイスへ転送
- *Rescale* 対象の暗号文 **ct**
- 暗号文の法 $(q_j)_{0 \leqq j \leqq l}$
- (i) で事前計算した値

(iii) *Rescale* 実行

(iv) 以下の値を，デバイスからホストへ転送
- (iii) の結果として得られた暗号文 $\mathbf{ct}'$

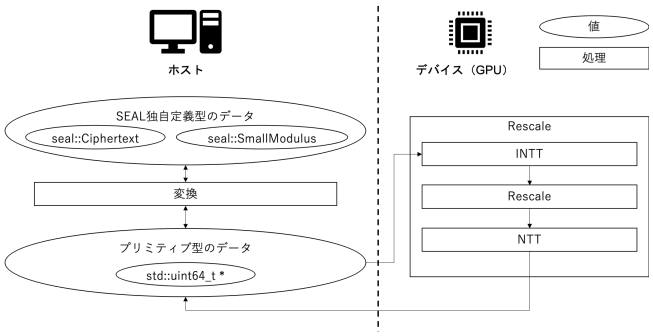

図 2　実装の概要図

6 実　　験

本節では，提案手法の性能評価を行う．測定は，1 種類の CPU 上と，世代の異なる 2 種類の GPU 上で行い，結果を比較する．さらに，得られた結果についての考察を行う．

6.1 実験の準備

実験環境について述べる．実験は，表 1 に示したホストマシンと，ホストマシンに導入した GPU カードを用いて行う．実験に使用する GPU カードは，Tesla P100 と Tesla V100-PCIe を用いる．それぞれの GPU カードの仕様を，表 2 に示す．

6.2 結　　果

測定結果を表 3 に示す．また，測定結果を，実行時間を対数軸にとり，グラフとして表したものを図 3 に示す．さらに，CPU の実行時間に対する，GPU の実行時間の割合を図 4 に示す．表中の SEAL-CPU，SEAL-GPU は，それぞれ CPU，GPU 上で実行した際の実行時間を表している．また，N を多項式の次数，L を回路の深さ（レベル）とする．実行時間は，10 回実行した結果の平均として算出した．実験では，デバイスへのデータ転送が完了し，*Rescale* が呼び出される瞬間から，

表 1　実験環境（ホストマシン）

名称	値
CPU	Intel(R) Xeon(R) E5-2690 v4
動作周波数 [GHz]	2.60
コア数	14
ソケット数	2
RAM サイズ [GB]	512
OS	CentOS Linux release 7.6.1810 (Core)
g++(GCC) version	6.3.1
nvcc version	9.0.176
CUDA version	9.0

表 2　実験環境（GPU）[15]

名称	値	
型名	Tesla P100	Tesla V100-PCIe
CUDA コア数	3,584	5,120
SM 数	56	80
動作周波数 [GHz] ※ GPU Boost 時	1.480	1.530
VRAM [GB]	16	32

Rescale の処理が終了し，デバイスのメモリへ結果が書き込まれるまでの時間を測定する．したがって，処理の際に発生するホスト—デバイス間のデータ転送時間は，実行時間には含まれない．なお，CPU については，*Rescale* が呼び出された瞬間から，*Rescale* の処理が完了するまでの時間を測定する．処理時間の計測には C++の標準ライブラリに含まれている chrono を使用した．ここで，$Q_L = \prod_{i=0}^{L} q_i$ を暗号文の法とする．

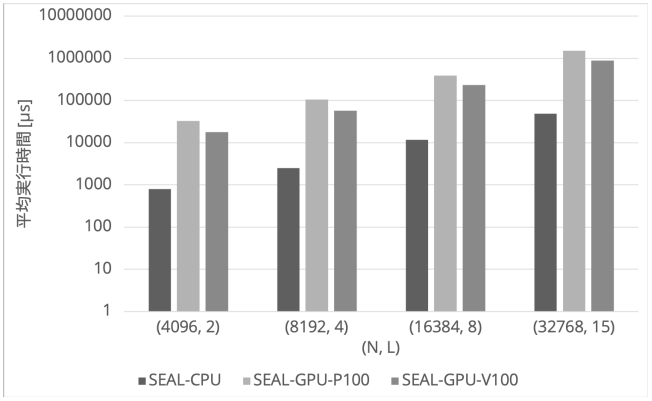

図 3　多項式の次数と *Rescale* の処理時間の変化

図 4 によると，GPU 上での実行速度は，CPU 上での実行速度に比べ，約 20〜40 倍程度低下した．

6.3 考　　察

速度低下の要因として，以下の 3 つが挙げられる．
- 64bit 整数演算の多用
- 高頻度なグローバルメモリへのアクセス
- 低い並列度

表 3　*Rescale* の処理時間比較

N	L	$\lceil \log Q_L \rceil$	SEAL-CPU [μs]	SEAL-GPU-P100 [μs]	SEAL-GPU-V100[μs]
4,096	2	72	800	32,476	17,733
8,192	4	174	2,520	105,550	57,447
16,384	8	389	11,770	386,873	231,750
32,768	15	825	49,137	1,501,470	877,117

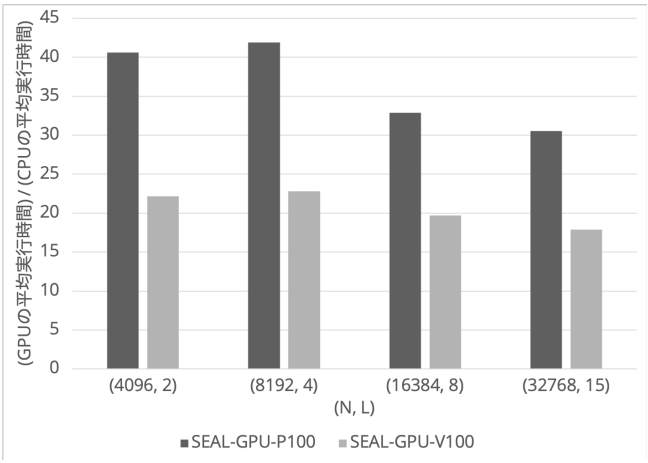

図 4　*Rescale* の CPU 実装に対する GPU 実装の実行時間割合

NVIDIA 製 GPU に搭載されている CUDA コアにおいて，整数演算は 32bit で動作する [16]．ゆえに，64bit の整数演算命令は，32bit 整数演算としてエミュレートされる．SEAL は，64bit アーキテクチャの CPU 上での実行を想定して設計されているため，内部では 64bit 整数を使用している．本研究では，SEAL の処理をナイーブに GPU へ移植したため，64bit 整数演算のエミュレーションによるボトルネックが影響していると考えられる．

また，グローバルメモリへの頻繁なアクセスも，パフォーマンス低下の大きな要因となっていると考えられる．本研究で行った実装では，あらゆる値をグローバルメモリに格納している．アルゴリズム 2.1 において，多項式同士の計算を行う際，多項式の係数の読み出しが発生する．そのため，計算の度に，グローバルメモリへの頻繁なアクセスが発生する．グローバルメモリへのアクセスは，GPU 上に存在するメモリ領域の中で最も低速である．ゆえに，グローバルメモリへの頻繁なアクセスが，パフォーマンスへ影響していると考えられる．これは，グローバルメモリよりも高速にアクセスできる共有メモリを適切に使用することにより，改善すると考えられる．

さらに，処理全体における並列度が低いことも，パフォーマンス低下の要因として考えられる．本研究では，多項式ごとにスレッドを割り当てており，並列度は，回路のレベルを L として，$2 * L$ であった．対して，各多項式内部の処理については，スレッド内で逐次実行されている．*Rescale* 処理や NTT については，多項式の係数ごとに処理を行っている箇所が存在する．多項式単位でなく，多項式の係数単位でスレッドを割り当てることによって，並列度を上げることができると考えられる．

次に，図 4 の結果に注目する．図 4 を見ると，多項式の次数が増加するにつれて，CPU の実行時間との差が小さくなっている．これは，多項式の次数が増えることにより，逐次実行の CPU に比べて，GPU の並列性能が効いているためだと考えられる．

7 おわりに

本稿では，GPU 上で CKKS 方式特有の演算の 1 つである *Rescale* をナイーブに実装することにより，実行速度の改善が見られるか，検証を行った．その結果，既存の CPU 実装より，GPU に移植した実装は，実行速度において約 20〜40 倍程度の性能低下が見られた．速度低下の要因として，グローバルメモリへの頻繁なアクセス，64bit 整数演算のエミュレーションによるオーバヘッド，低い並列度が考えられる．

今後の課題として，先行研究で用いられている様々な高速化手法を適用し，引き続き GPU 上の実装について検討を進めていくことが考えられる．また，Badawi らの先行研究 [19] と同様に，CKKS 方式の全ての演算を GPU 上へ実装し，包括的な性能測定を行うことが挙げられる．また，本稿で行った評価実験は，演算単位での性能測定にとどまっている．ゆえに，x^{-1} などの具体的な回路を CKKS 上へ実装し，性能測定を行うことも，今後の課題である．

謝　辞

本研究の一部は，JST（CREST，JPMJCR1503）及び NICT（Japan–US Network Opportunity 2 (JUNO2)）の支援を受けたものである．

文　献

[1] The European Parliament and the Council of the European Union, “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation),” https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679. accessed on Feb. 13. 2020.

[2] 数藤雅彦，川野智弘，“個人情報保護の最新動向 〜GDPR を中心に〜，” 二弁フロンティア，vol.12，pp.10–20，2018. accessed on Dec. 22. 2019. http://niben.jp/niben/books/frontier/frontier201812/2018_NO12_10.pdf

[3] State of California, “California Consumer Privacy Act of 2018,” http://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?lawCode=CIV&division=3.&title=1.81.5.&part=4.&chapter=&article=. accessed on Feb. 13. 2020.

[4] C. Gentry, "Fully homomorphic encryption using ideal lattices," Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing, pp.169–178, ACM, New York, NY, USA, Jan. 2009.

[5] J.H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol.10624, pp.409–437, Springer Verlag, 2017.

[6] C. Gentry, S. Halevi, and N.P. Smart, "Homomorphic evaluation of the AES circuit," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol.7417, pp.850–867, 2012.

[7] J.H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A Full RNS Variant of Approximate Homomorphic Encryption," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol.11349, pp.347–368, Springer Verlag, 2019.

[8] J.C. Bajard, J. Eynard, M.A. Hasan, and V. Zucca, "A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol.10532, pp.423–442, Springer Verlag, 2017.

[9] M.S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: High-Performance Architecture for Computation on Homomorphically Encrypted Data in the Cloud," arXiv:1909.09731, 2019.

[10] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol.9540, pp.169–186, Springer Verlag, 2016.

[11] A.A. Badawi, B. Veeravalli, C.F. Mun, and K.M.M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," IACR Transactions on Cryptographic Hardware and Embedded Systems, vol.2018, no.2, pp.70–95, May 2018.

[12] "Microsoft SEAL (release 3.3)," https://github.com/Microsoft/SEAL, June 2019. Microsoft Research, Redmond, WA.

[13] J.W. Cooley and J.W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," Mathematics of Computation, vol.19, pp.297–301, 1965.

[14] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol.263, pp.311–323, Springer Verlag, 1987.

[15] NVIDIA Corporation, "Nvidia tesla v100 gpu アーキテクチャ," https://images.nvidia.com/content/pdf/tesla/Volta-Architecture-Whitepaper-v1.1-jp.pdf, Aug. 2017. accessed on Dec. 31. 2019.

[16] NVIDIA Corporation, "Cuda c programming guide," https://docs.nvidia.com/cuda/archive/9.0/pdf/CUDA_C_Programming_Guide.pdf, June 2018. accessed on Jan. 2. 2020.

[17] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using gpu," Proceedings of IEEE Conference on High Performance Extreme Computing, pp.1–5, Sept. 2012.

[18] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol.6632, pp.129–148, 2011.

[19] A.A. Badawi, L. Hoang, C.F. Mun, K. Laine, and K.M.M. Aung, "Privft: Private and fast text classification with homomorphic encryption," arXiv:1908.06972, 2019.

[20] A.A. Badawi, Y. Polyakov, K.M.M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme," Cryptology ePrint Archive, Report 2018/589, 2018. https://eprint.iacr.org/2018/589.