一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2021-60,DC2021-94(2022-03)

演算ビット数削減による準同型暗号ライブラリ SEAL の高速化

宍戸　哲平† 　西　　将暉† 　李　　欣怡† 　木村　啓二†

† 早稲田大学

E-mail: †{teppei14860623,masaki170401}@kasahara.cs.waseda.ac.jp, ††keiji@waseda.jp

あらまし　コンピュータで様々な秘密情報を扱う機会が増えており，データを暗号化したまま計算できる準同型暗号に注目が集まっている．しかしながら，準同型暗号による演算コストは大きく，これまでにも並列化をはじめとするいくつかの高速化手法が提案されている．本稿では，代表的な準同型暗号ライブラリである SEAL とその演算高速化ライブラリ HEXL を対象に，演算で使用する基本データ構造をオリジナルの 64bit 整数から 32bit 整数に削減し，SIMD 演算幅を拡張することで準同型暗号演算を高速化する手法を提案する．今回は特に実数・複素数の処理が可能な CKKS 方式に着目して 32bit 化を実施し，準同型暗号の主要なアプリケーションの一つと目される深層学習の主要計算である行列積を評価した結果を報告する．評価の結果，オリジナルのライブラリに対し，32 ビット化したライブラリでは，最大で 3.27 倍の速度向上を得ることができた．

キーワード　準同型暗号 SIMD 並列化 OpenMP 並列化 NTT

Acceleration of Homomorphic Encryption Library SEAL by Reducing the Number of Arithmetic Bits

Teppei SHISHIDO†, Masaki NISHI†, Xinyi LI†, and Keiji KIMURA†

† Waseda University

E-mail: †{teppei14860623,masaki170401}@kasahara.cs.waseda.ac.jp, ††keiji@waseda.jp

1. はじめに

コンピュータが社会生活の多くの場面で利用されるようになり，それに伴い様々な種類のデータがコンピュータ上で扱われるようになっている．これらのデータの中には，金融や医療などの情報漏洩に伴う個人的あるいは社会的損失の大きいものも含まれる．このようなデータがシステムから漏洩しない，あるいは漏洩したとしても暗号化により内容が秘匿されたままにするための技術が重要となっている．

データを秘匿する方法として準同型暗号が注目されている．準同型暗号とは暗号化したまま演算を行い得られた結果を復号したものが平文演算の結果と一致する暗号技術である．準同型暗号を用いたアプリケーションの例として病院および医療機関の間で医療データの安全な共有を可能とするプラットフォーム，履歴データを使用して将来の傾向を予測する分析ツールなどが挙げられる [1]．しかしながら，準同型暗号は暗号により大きくなったデータを演算に利用するため実行時間が非常に大きくなってしまう問題がある．この問題を解決するために，これまでに並列化をはじめとする準同型暗号の高速化に関する研究がある．

一方，準同型暗号の主要アプリケーションの一つと目される深層学習の分野では，その推論処理で多くのビット数を必要としなくても精度を保てるという報告がある [2]〜[4]．準同型暗号の代表的なオープンソースのライブラリである SEAL と，その演算高速化ライブラリ HEXL は内部の演算が 64bit で行われているが，このような実装で深層学習の推論処理のような多くのビット数を必要としないアプリケーションを扱う場合，過剰にメモリ領域を消費してしまい，また SIMD 演算による高速化を行う場合にその演算幅が不必要に制限されてしまう．これに対して Intel による準同型暗号対応深層学習用グラフコンパイラ nGraph-HE2 では 32bit 演算可能な場合の演算最適化が言及され，部分的な評価が行われている [5]．しかしながら，部分的に 32bit 化しても他の処理部分との間でのデータ変換オーバーヘッドが課される．

本稿では，SEAL と HEXL の内部演算を 32bit 化することによる SIMD 並列幅拡張による高速化を提案する．準同型暗号による実数及び複素数の演算処理が可能な CKKS 方式に着目し，これに必要な処理を 32bit 化した．さらに，32bit 化した

実装により行列積演算を評価した結果について報告する．

本稿の構成は以下の通りである．まず，2. 節で準同型暗号について説明し，3. 節で準同型暗号技術の関連研究を紹介する．4. 節で SEAL ライブラリ上で行列積演算を実装した方法や行列積演算のボトルネックについて述べ，5. 節で提案する高速化手法について説明を行い，6. 節で提案した高速化手法の評価を報告する．最後に 7. 節でまとめる．

2. 準同型暗号

準同型暗号とは公開鍵 pk で暗号化したデータを用いて演算を行い，最終的に得られた結果を秘密鍵 sk で復号することで出力を得る公開鍵暗号方式の一種である．本稿では準同型暗号の演算を示す時，以下の例のように表記することとする．

$$Enc(A) + Enc(B) = Enc(A + B)$$

$$Enc(A) \times Enc(B) = Enc(A \times B)$$

準同型暗号は LWE 問題と呼ばれる計算量困難性を安全性の基点とした暗号である．SEAL ライブラリでは Ring-LWE 問題と呼ばれる LWE 問題を有限体上の多項式環に限定した問題をベースとしている．

準同型暗号では乗算を行うと暗号文のサイズ・暗号文に内在するノイズが増加する．暗号文のサイズが大きくなるほど実行時間が大きくなり，ノイズがある閾値を超えると復元ができなくなってしまう．乗算により大きくなったノイズを削減する処理が relinearize(relin とする) である．またノイズを削減するために bootstrapping と呼ばれる手法が Gentry によって提案された [6]．bootstrapping を利用することで暗号化した状態を保ちつつ, 暗号文内のノイズを削減することにより任意の回数演算を行うことが可能となった．このように回数制限なく演算が行えるものを完全準同型暗号 (FHE) と呼び，Gentry によって初めて FHE の具体的な構成が示された．しかし，bootstrapping は非常に大きな実行時間がかかる．

その後も準同型暗号の分野は発展し続けており, BGV 方式 [7] や SV パッキング [8] という高速化手法をはじめとする様々な手法が提案されている．以下で本研究で用いる準同型暗号ライブラリ SEAL に実装されている Cheon らによって提案された CKKS 方式 [9], 平文・暗号文空間とパッキング, 及び NTT(数論変換) について詳しく説明する．

2.1 CKKS 方式

入力値が整数でなければならない BFV 方式に対して，CKKS 方式とは入力値が実数または複素数である場合でも演算を行うことができる方式である．CKKS 方式では平文に対してパラメータ scale を利用したスケーリングを行い固定小数点として扱う．scale は暗号文乗算毎に増加するため，複数回乗算を行う場合は rescale と呼ばれる操作が必要となる．CKKS 方式を利用した演算の復号結果は，実数を整数に近似することに起因する誤差が生じることに注意しなければならない．

2.2 暗号文空間とパッキング

SEAL ライブラリにおいて暗号文は多項式環として表現される．この多項式環を表現する上で必要となる次数と法について説明する．次数は多項式の長さを制限するものであり，長さは 2 の正の冪乗である必要がある．次数を大きくすると暗号文が大きくなり操作が遅くなるが，より大きなサイズの行列積演算が可能となる．法は多項式の係数の大きさを制限する．SEAL ライブラリでこの法は複数の素数を組み合わせて表現されており，これをモジュラススイッチングチェーンと呼ぶ．

次にパッキングについて説明する．パッキングとは一つの暗号文に複数要素を埋め込む技術であり，これを利用することで複数要素に対する同一演算の大幅な高速化が可能となる．複数の値を一つの平文にパッキングする操作を encode，その逆の操作を decode と呼ぶ．一般に，パッキングを用いた暗号処理は encode，暗号化，暗号演算，復号，decode の順で行われる．

パッキングでは暗号化前の各要素をスロットと呼ばれる値格納用の一種の領域に配置する．一つの暗号文に格納できる要素数をスロット数と呼び，CKKS 方式の場合は (暗号文を表現する多項式の長さ/2) がスロット数となる．

パッキング処理が行われた暗号文を演算する場合，スロット毎の演算が一度に行われる．例えば，スロット数を N として，要素数 N のベクトル $a([a_1, a_2, .., a_N])$，$b([b_1, b_2, .., b_N])$ をそれぞれ暗号化した暗号文の加算・乗算を考える．スロット $i(1 \leqq i \leqq N)$ にはそれぞれ a_i，b_i の値が格納されていると考える．

$$Enc([a_1, a_2, .., a_N]) + Enc([b_1, b_2, .., b_N])$$
$$= Enc([a_1 + b_1, a_2 + b_2, .., a_N + b_N])$$
$$Enc([a_1, a_2, .., a_N]) \times Enc([b_1, b_2, .., b_N])$$
$$= Enc([a_1 \times b_1, a_2 \times b_2, .., a_n \times b_N])$$

スロットを利用した演算はこのように同一スロットに格納された値同士を演算した結果が得られる．

スロットに関する操作として, 以下のような暗号文のスロットに格納されている値の順番をずらす rorate 処理が利用できる．

$$Enc([a_1, a_2, .., a_N]) \rightarrow rotate(1) \rightarrow Enc([a_N, a_1, .., a_{N-1}])$$

rotate の併用により異なるスロット間の演算が可能となる．

2.3 NTT(数論変換)

多項式の項数が 2 の冪乗である場合，FFT を利用して多項式乗算の高速化が可能である．準同型暗号計算では素数の法を考慮した多項式乗算を多用するが，特定の素数を法とした剰余体上で行われる FFT を数論変換（Number Theoretic Transform: NTT）といい，SEAL でも利用されている．本研究では，AVX512 によって高速化された NTT を備えるオープンソースライブラリである Intel HEXL [10] を利用する．

3. 関 連 研 究

Riazi 等は CKKS 方式の主要処理である暗号乗算，NTT，逆 NTT，鍵交換関数を FPGA 上で実装した [11]．本実装では，64bit 乗算は FPGA の 27bit 乗算器を組み合わせて実現している．Mert 等は BFV 方式の暗号化，復号，NTT などを FPGA

上で実装した [12]．本実装では 32bit の多項式を想定しているが，CPU-FPGA 間のデータ転送は 64bit 単位で行なっている．

これらの研究ではハードウェアアクセラレータに一部の処理を実装することで高速化を検討しているが，準同型暗号全体の 32bit 以下での実装は行われていない．本稿では SEAL，HEXL の CKKS 方式の 32bit 化による SIMD 幅拡張をはじめとする高速化を実施した．

4. SEAL 上の行列積演算の実装

4.1 想定するシナリオ

ある学習済みモデルを暗号化した状態でクラウド上に置き，そのモデルに暗号化されたデータを入力して出力を得ることを想定する．このシナリオに従い，行列積演算は暗号文同士の演算を行うこととする．

4.2 SEAL 及び HEXL

SEAL は準同型暗号の代表的オープンソースライブラリ実装の一つである．入力値が整数のみに対応した BFV 方式及び実数・複素数に対応した CKKS 方式が実装されており，本稿では CKKS 方式を利用する．SEAL 内部の演算は 64bit 整数により行われる．本稿で利用する SEAL のバージョンは 3.6.6 である．CKKS 方式での実行で利用されるパラメータを表 1 に示す．

表 1　SEAL の CKKS 方式に利用されるパラメータ

パラメータ名	概要
poly_modulus_degree	多項式の次数 (スロット数× 2)
coeff_modulus	暗号文空間の法
scale	実数を整数に近似するための値

poly_modulus_degree は多項式の次数，つまり暗号文の長さを表すパラメータである．coeff_modulus は暗号文の多項式係数の法となる素数の組を表すパラメータであり，構成する素数の bit 数と個数を指定する必要がある．例えば $\{30, 30, 30, 30\}$ であれば暗号文多項式の係数の法は 30bit の素数 4 つで構成されることになる．この素数の個数によって乗算可能回数が決定され，これをレベルという．scale は実数を整数にスケーリングするためのパラメータである．

HEXL は NTT 等の演算を AVX512 により高速化したオープンソースの準同型暗号高速化ライブラリである [10]．本稿で利用する HEXL のバージョンは 1.1.0 であり，内部の演算は SEAL と同様に 32bit 整数で行われる．バージョン 1.2.0 以降でも基本的に 64bit で演算が行われているが，値が 32bit 以下の場合に命令を変更することによる高速化を行う箇所が存在する．

4.3 行列積演算の実装

SEAL の CKKS 方式での行列積演算はパッキングと rotate を利用する．本稿の行列積演算は行列-ベクトル積演算の実装を拡張して実装しているため，まず行列–ベクトル積演算の実装を述べる [13]．簡単のため 3×3 行列と 3 要素ベクトルの積として説明する．この演算は以下のように変形できる．

$$\begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} A & 0 & 0 \\ 0 & E & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$$+ \begin{pmatrix} C & 0 & 0 \\ 0 & D & 0 \\ 0 & 0 & H \end{pmatrix} \begin{pmatrix} z \\ x \\ y \end{pmatrix} + \begin{pmatrix} B & 0 & 0 \\ 0 & F & 0 \\ 0 & 0 & G \end{pmatrix} \begin{pmatrix} y \\ z \\ x \end{pmatrix}$$

これに基づき，行列は斜めに分割してそれぞれをベクトルとして暗号化し，ベクトルは元ベクトルのみを暗号化し rotate を利用することで元ベクトルから値の順番をずらしたベクトルの暗号文を得る．それらの暗号文を利用して以下のような演算を行う．

$$Enc([A, E, I]) \times Enc([x, y, z]) = Enc([Ax, Ey, Iz])$$

$$Enc([C, D, H]) \times Enc([z, x, y]) = Enc([Cz, Dx, Hy])$$

$$Enc([B, F, G]) \times Enc([y, z, x]) = Enc([By, Fz, Gx])$$

これらの演算結果を足し合わせると

$$Enc([Ax + By + Cz, Dx + Ey + Fz, Gx + Hy + Iz])$$

となり，パッキングを活用しつつ正しい結果が得られる．実際はスロットの余った部分に 0 を代入する必要があるので元ベクトルの暗号文は以下のように表現される．

$$Enc([x, y, x]) \rightarrow Enc([x, 0, 0, 0, .., y, 0, 0, 0, .., z, 0, 0, 0, ..0])$$

ここでベクトルを行列にする，つまり行列積演算を実行する場合，0 で埋めている部分に行列要素を埋め込むことで演算が可能となる．例えば今回の 3 要素ベクトルを以下のような 3×3 の行列とする場合

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x & u & r \\ y & v & s \\ z & w & t \end{pmatrix}$$

$$Enc([x, u, r, 0, .., y, v, s, 0, .., z, w, t, 0, ..0])$$

とすればよい．このように行列積演算は行列–ベクトル積演算と実行処理はほぼ同じであるため，実行時間は概ね一致する．そして行列積演算において rotate がボトルネックになり得ることが知られている．

5. 準同型暗号演算の高速化

本節では行列積演算プログラムの並列化，HEXL の 32bit 化，SEAL の 32bit 化，その他の SEAL 高速化について述べる．

5.1 行列積演算プログラムの並列化

SEAL，HEXL ライブラリによる行列積演算は，行列を斜めに分割してそれぞれを encode・暗号化するループ処理，暗号文を rotate により生成し対応する暗号文と掛け合わせるループ処理という複数のループから構成される．これらのループは並列化可能であり．本稿では OpenMP により並列化した．

5.2 HEXL の 32bit 化

SEAL を 32bit 化する場合, 組み込まれている HEXL も 32bit 化する必要がある．そのため HEXL の 32bit 化を行なった．表 2 に 32bit 化した HEXL の関数を，関数名を簡略化して示す．表中，$A[i]$，$B[i]$ は配列，B は変数をそれぞれ表す，また，q は法である．

表 2　HEXL で 32bit 化した関数

関数名	概要
fwd_NTT・inv_NTT	NTT と逆 NTT
AddMod	A[i]+B mod q ・ A[i]+B[i] mod q
SubMod	A[i]-B mod q ・ A[i]-B[i] mod q
MultMod	A[i] × B mod q ・ A[i] × B[i] mod q
ReduceMod	A[i] mod q

HEXL には AVX512 により実装された，複数要素に対するシフト，比較，乗算などの基本処理を行う関数 (HEXL 基本関数とする) が用意されており，表 2 の関数をはじめとする様々な関数で利用されている．以降，HEXL の 32bit 化について HEXL 基本関数の 32bit 化，及びそれらを利用した表 2 の関数の 32bit 化に分けて順に説明する．

はじめに HEXL 基本関数の 32bit 化について説明する．これらは基本的に AVX512 の 64bitSIMD 命令を 32bitSIMD 命令に置き換える．例えば乗算を行い下位 bit を求める関数では内部に利用される_mm512_mullo_epi64 命令を _mm512_mullo_epi32 命令に置き換える．しかし置き換えだけで対応できない場合もある．例えば乗算を行い上位 bit を求める関数では，利用されている_mm512_shuffle_epi32 命令を置き換えるだけでは対応できず，この場合は複数命令を組み合わせるなどして対応する必要がある．

次に表 2 の関数の 32bit 化について説明する．NTT, 逆 NTT 以外の関数は，対応する HEXL 基本関数を利用し，SIMD 演算幅が 8（512bit/64bit）から 16（512bit/32bit）になったことを考慮して実装する．

NTT の 32bit 化に関しては以下のようになる．NTT では配列を分割し，分割された配列に対するバタフライ演算を再帰的に実行する．図 1 のように最初に配列を 2 つに分割し，それぞれの配列の先頭から順番に 16 要素毎にバタフライ演算を適用していく．そして次に配列を 4 分割して同様の処理を行う．

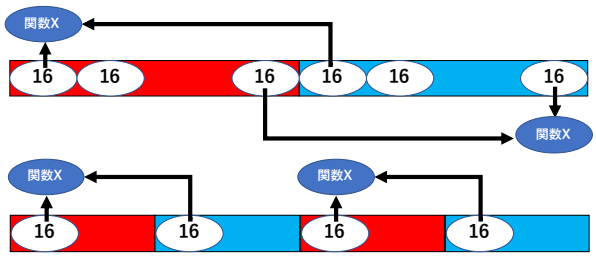

図 1　NTT における処理の挙動 (2 分割・4 分割)

配列を 4 分割して処理した後，8 分割，16 分割のように分割を再帰的に繰り返し最終的にはブロック毎の要素数が 1 になるまで続ける．ここでブロック毎の要素数が 8，4，2，1 の場合は SIMD 命令を活用するために配列要素を並び替える必要がある．例えばブロック毎の要素数が 8，4 の場合は図 2 のように並び替えることで SIMD 幅を活かすことができる．このよ

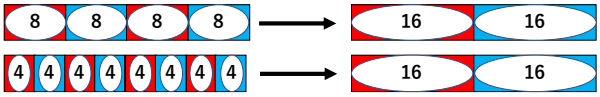

図 2　NTT における配列要素の並び替え

うに SIMD 幅が 8 から 16 に変わったことによる影響を考慮して，32bit 用のバタフライ演算，配列の並び替え関数などを用意することで NTT を 32bit 化した．逆 NTT は NTT の処理を逆から行うものであり，同様にして 32bit 化した．

5.3 SEAL の 32bit 化

SEAL の内部演算の 32bit 化を実現する上で暗号文を表現する多項式の係数型を 64bit 型から 32bit 型にすることが必要である．ここでまず，暗号文の生成方法について説明する．暗号文は入力値 (行列の部分要素) を encode(FFT と NTT を含む) し，次に暗号化を適用することで得られる．NTT と暗号化は多項式係数の法が値の上限値を制限するため，coeff_modulus パラメータを 32bit 以内に設定することで出力値を 32bit 以内に収めることができる．以上のように暗号化パラメータの調整により暗号文の係数を 32bit 型に修正した．

暗号文を表す多項式係数が 32bit であるという条件の下，encode・decode 関数や暗号文に関わる演算や rotate などの関数を 32bit 化することにより SEAL の内部演算の 32bit 化を実現した．これらの関数は引数の型やローカル変数の型を変更するだけでなく，例えば事前計算を行うために用意された 64bit 用の値を 32bit に対応させるなど，周辺の処理も 32bit 化した．

5.4 その他の SEAL 高速化

その他，galois 鍵生成関数，鍵交換関数を高速化したので本節で説明する．

galois 鍵生成関数は rotate 処理で利用される鍵である galois 鍵の生成関数である．この関数内のループ処理において例外処理を呼び出す可能性のある妥当性確認が行われていた．この処理を外に出し事前に行うようにした上で，ループ処理を OpenMP により並列化した．

鍵交換関数は rotate や relinearize 内部で実行される関数であり，それぞれの関数の実行時間の 9 割以上を占める．この関数に対して 2 つの高速化を行った．1 つ目は，ループの中の if 文を外に移動できる場合，移動させたことである．2 つ目は，4 重ループになっている箇所の乗算と加算を AVX512 による SIMD 化を行ったことである．32bit の変数同士の乗算を行うと結果が 64bit になるため，32bit の変数を 512bit の SIMD レジスタに代入した後，SIMD レジスタの内部の 32bit の変数を 64bit に変換してから乗算を行っている．

6. 評　　価

本節では 32bit 化した SEAL 及び HEXL により，鍵交換関

数，SEAL の基本処理，及び SEAL の行列積演に対して実行時と演算精度を評価した結果について述べる．

6.1 評 価 環 境

評価は Intel Xeon W-2145 を搭載したサーバ上で実施した．本 CPU は 3.70GHz で動作する Skylake コアを 8 基搭載し，各コアが L1 命令キャッシュ 32KiB, L1 データキャッシュ 32KiB, 及び L2 キャッシュ 1MiB を持つ．さらに，これらのコアごとのキャッシュに加えコア共有の L3 キャッシュ 11MiB を搭載する．また，命令セットは AVX512 をサポートする．

6.2 実行時のパラメータ設定

評価を行う上で，表 1 のパラメータを決定する必要がある．

poly_modulus_degree は 2 の冪乗である必要があり 1024, 2048，4096，8192，16384，32768 が推奨されている．このパラメータは大きいほど実行時間が大きくなる一方で，より多くの行列要素を利用可能である．本評価では 32768 を採用する．

coeff_modulus は SEAL の 32bit 化にあわせて暗号文が 32bit 以内になるように素数の bit 数，個数を設定する．素数の bit 数は 30bit を超えると内部演算で 32bit を超えるため 29bit とする．本稿では深層学習の特に CNN での計算を想定しており，CNN で消費されるレベルは 10 前後であるため個数を 11 個とした．これより coeff_modulus を $\{29 \times 11\}$ とする．

scale は rescale 時に利用する coeff_modulus の素数と対応させるために値を 2^{29} とする．

次に時間測定を行う際に利用するライブラリを表 3 で示す．SEAL64 と HEXL64，SEAL32 と HEXL32 をそれぞれ組で使用する．

表 3　利用するライブラリ一覧

表記	概要
SEAL64	バージョン 3.6.6 の SEAL
SEAL32	SEAL64 を高速化 (32bit 化とその他高速化) したもの
HEXL64	バージョン 1.1.0 の HEXL
HEXL32	HEXL64 を 32bit 化したもの

6.3 鍵交換関数の実行時間評価

行列積演算及びその内部の鍵交換関数の測定結果を表 4 に示す．32bit 化した SEAL 内で鍵交換関数を SIMD 化しているため，32bit 化した SEAL 上での測定となっている．なお，本評価では HEXL を使用している．また，行列積演算には鍵生成や暗号化及び復号を含まない．表の sw_k は鍵交換関数を，sw_k_mul はその中の SIMD 化した乗算部をそれぞれ示す．表 4 より鍵交換関数は行列積演算の実行時間の 9 割以上を占めることがわかる．また，SIMD 化により乗算部が 3.08 倍，鍵交換関数全体で 1.41 倍，行列積では 1.39 倍の速度向上がそれぞれ得られた．

表 4　鍵交換関数の実行時間 [s]

環境	SIMD	行列積演算	sw_k	sw_k_mul
SEAL32	OFF	5.11	4.93	2.03
	ON	3.67	3.50	0.66

6.4 SEAL の基本処理の実行時間評価

SEAL の基本処理関数 1 回あたりの測定結果を表 5 と表 6 に分けてそれぞれ示す．なお、本評価では HEXL を使用している．ここで rotate_r は 0~(スロット数-1) までの乱数分の回転，galois_key は galois 鍵生成である．また galois_key は OpenMP 並列化による高速化を行ったので，本処理のみ 8 スレッドで測定した．

表 5，6 より全ての関数で 32bit 化により速度向上したことがわかる．特に rotate，rotate_r，relin は 32bit 化に加え鍵交換関数の高速化の影響を受けるため更に大きな 2.8~3.3 倍の速度向上となっている．また galois_key は 32bit 化に加え OpenMP 並列化により 8.6 倍の速度向上を得ることができた．

表 5　SEAL の基本処理の実行時間（1）[ms]

環境	encode	decode	encrypt	decrypt	add	mul
SEAL64	2.85	13.3	17.8	1.42	0.97	1.74
SEAL32	2.20	12.5	12.7	0.72	0.48	1.35

表 6　SEAL の基本処理の実行時間（2）[ms]

環境	rotate	rotate_r	rescale	relin	galois_key
SEAL64	60.0	260.0	5.13	59.0	4,210
SEAL32	18.2	94.2	3.15	17.7	490

6.5 SEAL の行列積演算プログラムの実行時間評価

図 3 に SEAL による行列積プログラムの流れと今回実施した高速化を示す．

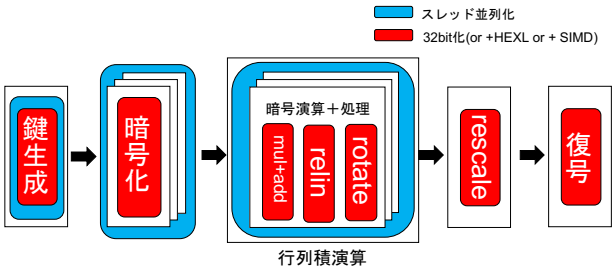

図 3　行列積演算プログラムの高速化箇所

本節では繰り返し行われると想定される行列積演算（add, mul，relin，rotate）のみ測定対象とする．ここで行列積演算の入力値は CNN を想定し，行列 1 を画像データに見立てて 0~255，行列 2 をカーネルに見立てて 0.01~1 としている．まず 1 スレッドでの測定結果を表 7 に示す．ここで add は実行時間が小さいため省略している．表中，使用メモリ量は鍵生成，暗号化，及び復号の処理を含む行列積演算プログラム全体を通してのものである．

表 7 より，HEXL の利用により SEAL32，SEAL64 ともに速度向上していること，HEXL を利用した場合，SEAL32 の方が SEAL64 より行列積演算内部の処理が高速であることを確認できた．また HEXL 使用による速度向上は SEAL32 の方が大きく，relin で 2.13 倍，rotate で 2.09 倍となった．これは 32bit

化に加え鍵交換関数の SIMD 化によるものである．また 32bit 化により確保されたメモリ量が SEAL32 では SEAL64 の約半分となっていることがわかる．

表 7　1 スレッドでの行列積演算の実行時間 [s]

環境	HEXL	行列積演算			使用メモリ量
		mul	relin	rotate	[GB]
SEAL64	OFF	0.25	5.35	11.7	2.28
	ON	0.081	3.70	8.10	
SEAL32	OFF	0.20	2.36	5.15	1.15
	ON	0.067	1.11	2.47	

次にスレッド数 1，4，8 での測定結果を表 8 に示す．いずれも HEXL を使用している．

表 8　各スレッド数での行列積演算の実行時間評価 [s]

環境	1 スレッド	4 スレッド	8 スレッド
SEAL64	12.0	3.84	2.55
SEAL32	3.67	1.29	0.917

表 8 より，1 スレッドと比較して SEAL64 では 4 スレッドで 3.13 倍，8 スレッドで 4.71 倍の速度向上，SEAL32 では 4 スレッドで 2.84 倍，8 スレッドで 4.00 倍の速度向上が得られた．また SEAL32 は同一スレッド数の SEAL64 と比較して，1 スレッドで 3.27 倍，4 スレッドで 2.98 倍，8 スレッドで 2.78 倍の速度向上をそれぞれ得ることができた．SEAL32 では SEAL64 に比べてスレッド数増加に伴う速度向上率が低いが，これは SEAL32 の SIMD 演算による要求メモリバンド幅の増加，及び各スレッドの部分和の総和処理オーバーヘッドの相対的な増加のためである．

6.6 演 算 精 度

SEAL32 での行列積演算の演算精度について検討した．本稿では行列積の精度を以下の式のように定義する．ここで Ans は通常の演算により求めた行列積演算結果，Res は SEAL での行列積演算結果である．

$$(\text{精度}) = 100 \times \left(1 - \left(\sum_{i=1}^{N}\sum_{j=1}^{N}\left|\frac{Ans_{i,j} - Res_{i,j}}{Ans_{i,j}}\right|\right)/N^2\right) \qquad (Ans_{i,j} \neq 0)$$

上記定義に基づき SEAL32 の演算精度を評価した結果，SEAL64 の場合と同様の 99% となり演算の精度に問題はないことがわかった．さらに基本データ型のビット数を 16bit とした場合ついても検討した．この場合，coeff_modulus の素数を 16bit 以下にする必要があり，そのように設定するには暗号文の長さを非常に小さくしなければならず行列積演算を行うことはできないことが分かった．

7. ま と め

本稿では，準同型暗号ライブラリのオープンソース実装である SEAL とその演算高速化ライブラリ HEXL で利用されている基本データ型を 32bit 化し SIMD 演算幅増加による高速化を提案した．32bit 化した SEAL と HEXL を評価して従来の SEAL と比較した結果，基本処理を 1.1~3.3 倍高速化できた．32bit 化と共に OpenMP 並列化を適用し 8 スレッドで実行した galois 鍵生成では 8.6 倍の性能向上が得られた．また，行列積演算では 32bit 化前後の同一スレッド数で比較すると，1 スレッド実行で 3.27 倍，8 スレッドで 2.78 倍の高速化をそれぞれ得ることができた．また使用メモリ量も半減可能なことを確認できた．演算精度に関しても，入力データを 32bit で表現できる範囲であれば，32bit 化した SEAL の結果が問題ないことを確認した．

文　　献

[1] Ayantika Chatterjee and Khin Mi Mi Aung. *Fully homomorphic encryption in real world applications*. Springer, 2019.

[2] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[3] Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights +1, 0, and − 1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 1–6, 2014.

[4] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[5] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pp. 45–56, 2019.

[6] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169–178, 2009.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, Vol. 6, No. 3, pp. 1–36, 2014.

[8] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, Vol. 71, No. 1, pp. 57–81, 2014.

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 409–437. Springer, 2017.

[10] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. *arXiv preprint arXiv:2103.16400*, 2021.

[11] B. Pelton M.S. Riazi, K. Laine and W. Dai. Heax: High-performance architecture for computation on homomorphically encrypted data in the cloud. *arXiv:1909.09731*, 2019.

[12] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 28, No. 2, pp. 353–362, 2019.

[13] Shai Halevi and Victor Shoup. Algorithms in helib. In *Annual Cryptology Conference*, pp. 554–571. Springer, 2014.