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

信学技報
TECHNICAL REPORT OF IEICE.

準同型暗号による行列積の高速化の検討

牧田　哲也† 　宍戸　哲平† 　和田　康孝†† 　木村　啓二†

† 早稲田大学
†† 明星大学

E-mail: †{makita,teppei14860623}@kasahara.cs.waseda.ac.jp, ††yasutaka.wada@meisei-u.ac.jp,
†††keiji@waseda.jp

あらまし　深層学習技術の発展と広がりに伴い，これを利用した多くのサービスが提供されつつある．これらのサービスの実現方式として，ユーザデバイスからクラウドにデータを送信し，その後クラウドで処理を行い出力を得るクラウド型サービスもある．クラウド型サービスとしてヘルスケアサービスのようなユーザの秘密情報を扱うサービスを提供する場合，クラウド上でこれら秘密情報を安全に扱えなければならない．このための技術として準同型暗号が注目されている．準同型暗号とは，暗号文を平文に戻すことなく計算結果を得ることのできる暗号であり，秘密を保持したままでのデータ処理が可能となる．しかしながら，準同型暗号を用いた演算は暗号化により大きくなったデータの演算になるため，非常に大きな時間がかかる．本稿では，深層学習の主要計算となる行列積に着目し，これを完全準同型暗号ライブラリ「HElib」上で実現した場合の高速化手法を検討した結果について報告する．HElib 上の行列積演算高速化の第一歩として KeySwitching における Horner 法計算と暗号文化された値の演算部に対して SIMD 化による高速化を行った．その結果，Horner 法計算部では約 3.4 倍，暗号文の演算部では加算部で最大 5.53 倍，乗算部で最大 3.73 倍の速度向上をそれぞれ得ることができた．

1. はじめに

計算機の能力の拡大や，ビッグデータの収集・活用を可能とする環境の広がりと共に，深層学習を利用したサービスの実用化が進められている．深層学習を利用するサービスの実現方式として，大きく分けて組み込み型やクラウド型の 2 つのシナリオが考えられる．組み込み型サービスではユーザデバイス内に組み込まれた推論部を利用するため処理がユーザデバイス内で完結するのに対し，クラウド型サービスでは処理を行うためにクラウド環境を利用する．

ヘルスケアや金融など，ユーザの秘密情報を扱うアプリケーションも深層学習を利用したサービスとして考えられる．このような秘密情報を扱うサービスをクラウド上で実現するためにはプライバシー保護が必須であり，そのための方法として準同型暗号が注目されている．

準同型暗号は暗号化したまま演算を行い最終結果を復号することで平文演算と同一の結果を得ることができる暗号技術である．準同型暗号を用いた機械学習を利用したサービスの例として，電子投票などの集計，医療データの統計分析や生体認証，スパムメール対策のような高度な分析サービスなどが存在する [1].

準同型暗号には加算のみが可能な型，乗算のみが可能な型，加算・乗算ともに可能だが演算の回数に制約がある Somewhat 準同型暗号 (SWHE)，演算の回数に制約がない完全準同型暗号 (FHE) が存在する．2009 年に Gentry が SWHE に bootstrapping と呼ばれる暗号化される度に増加するノイズを削減する手法を取り入れることで FHE の具体的な構成法を示した [2]．その後，様々な bootstrapping を用いた暗号手法が提案された．その後，bootstrapping を用いずに FHE を実現する BGV スキームが提案された [3]．また複数の要素を 1 つの暗号文として暗号化することで演算効率を向上させる SV パッキングと呼ばれるパッキング手法が Smart と Vecrauteren がによって提案された [4]．これらの技術により，基本的な整数演算や複数の要素に対する SIMD 型の演算が可能となり，様々なアプリケーションを実現するための枠組が整いつつある．すなわち，準同型暗号の利用によりサービス提供者にユーザの情報を開示せずに情報処理が可能となりつつある．

その一方で，準同型暗号による演算は非常に時間がかかるという問題がある．例えば，FHE ライブラリ「HElib」を用いて準同型暗号による行列–ベクトル積を実現した場合，次元数 682 の計算に Intel Xeon E5-2968 v3 (2.30GHz) 上で 2.22 秒もの時間を要するという報告もある [5]．準同型暗号を用いたサービスを実用的に運用するには更なる高速化が必要となる．

本稿では，深層学習の推論処理の高速化を目標とし，まずその主要計算となる行列積の完全準同型暗号ライブラリ「HElib」による実装の高速化を検討した結果について報告する．その第一

歩として，HElib による行列積を構成要素である KeySwitching における Horner 法計算，及び暗号化された値の演算部を，それぞれ SIMD 化することで高速化した．

本稿の構成は以下の通りである．まず，第 2. 節で準同型暗号技術の関連研究を紹介する．第 3. 節で HElib に対する調査結果を述べ，第 4. 節で提案する高速化手法の説明を行い，第 5. 節でその評価結果について報告する．最後に第 6. 節でまとめる．

2. 関 連 研 究

本節では本稿の前提となる完全準同型暗号（FHE）の概要について，特にこれまでに提案された高速化手法の観点から説明する．

HElib による準同型暗号は，誰でも入手可能な公開鍵 pk を用いてデータを暗号化し，復号の権限を持つもののみが所持する秘密鍵 sk で復号する公開鍵暗号の一種である．

準同型暗号の演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．特にノイズが増加した結果ある閾値を超えると復号不可能になるため，bootstrapping と呼ばれるノイズを減らす処理が必要となる [2]．bootstrapping はノイズを蓄積した暗号文を再び暗号化し暗号化された秘密鍵を用いて復号処理を行うことで，ノイズが減少した新たな暗号文を得る操作である．Gentry は bootstrapping 手法を提案し，これにより初めて FHE の具体的な構成法が示された．しかし，この bootstrapping の実行には非常に時間がかかる．そのため，これまでに乗算のノイズの増加を緩やかにして bootstrapping の回数を減らす手法 [6] [7]，計算回路のデータフローグラフ上で効率よく bootstrapping を行う箇所を求める手法 [8] [9] などが提案されている．

Brakerski, Gentry, Vaikuntanathan らは bootstrapping を用いずに FHE を実現する BGV スキームを提案した [3]．また，従来 1 要素ごとに暗号化していたのに対し，Smart, Vecrauteren らは，中国余剰定理 (CRT) を用いたパッキングにより要素をまとめて暗号化して扱うことが可能な SV パッキングを提案した．これらの手法によって暗号文の数，暗号演算の回数を共に削減することができる．

本稿で用いた FHE ライブラリ「HElib」上で列積演算を行うにあたり BGV スキーム [3]，SV パッキング [4] が重要となるのでここでより詳しく説明する．

2.1 BGV スキーム [3]

BGV スキームでは Ring-LWE 問題に基づいて平文，暗号文を多項式環で表現する．

$$\text{平文} : R_{p^r} = \mathbb{Z}[X]/(\Phi_m(X), p^r)$$

$$\text{暗号文} : R_q = \mathbb{Z}[X]/(\Phi_m(X), q)$$

$\Phi_m(X)$ は円分多項式である．平文空間，暗号文空間は上記のように多項式環として表される．平文空間は $\Phi_m(X)$ で割り，p^r（p は素数）を係数の法とすることで制限されている．暗号文空間も $\Phi_m(X)$ で割り，q を係数の法とすることで制限されている．BGV スキームには modulus chain と呼ばれる暗号文の法の組 $q_L > q_{L-1} > \cdots > q_0$ が存在し，準同型暗号演算を繰り返す度に小さな法にスイッチしていくようになっている．これを modulus switching と呼ぶ．暗号文の法を q_i から q_{i-1} にスイッチすると暗号文のノイズはおおよそ q_{i-1}/q_i の比率で削減できる．これによって bootstrapping を用いずに FHE を実現している．

2.2 SV パッキング [4]

SV パッキングにより，n 個の暗号化対象の平文要素 $a_i (0 \leq i < n)$ を $[a_0, a_1, \ldots, a_{n-1}]$ のベクトル（まとまり）として暗号化する過程を下記に記す．まず各要素を $[a_0, a_1, \ldots, a_{n-1}]$ を多項式変換したもの $f(X)$ を $F_i(X)(0 \leq i < n)$ で割った余りであると考える．$F_i(X)(0 \leq i < n)$ は

$$\Phi_m(X) = F_0(X) \times F_1(X) \times \cdots \times F_{n-1}(X) \quad [mod \quad p]$$

である．$f(X) \equiv a_i [mod \quad F_n(X)](0 \leq i < n)$ を満たす $f(X)$ が CRT よりただ一つ存在する．この $f(X)$ を暗号化することによって $[a_0, a_1, \ldots, a_{n-1}]$ の複数の要素をまとめて暗号化できる．$\Phi_m(X), F_i(X)(0 \leq i < n)$ それぞれの可能な最大の次数を l, d とすると $d = l/n$ であり，$\mathbb{F}_i = f(X)/F_i(X) \quad [mod \quad p^d, 0 \leq i < n]$ をスロットと呼ぶ．まとめて暗号化する複数の要素同士は全て異なるスロットに入っている．ここで注意すべきは，異なるスロット位置に配置されている要素同士は直接には演算できないことである．

異なるスロット位置に配置された要素同士を演算する方法として，hypercube 構造のネットワークを導入し，要素間の入れ替えを行う手法が提案されている [10]．ネットワークを経由して構成するとスロット位置の違う値を入れ替えることにより，元々異なるスロット位置に配置されていた要素同士の演算も可能になる．暗号化したスロット位置の異なる値を移動させる操作を automorph といい，このとき秘密鍵 sk が変化する．この操作は第 3. 節と特に関連する．

3. HElib を用いた行列積演算 [5]

本稿では行列積の高速化手法の検討対象として，HElib のサンプルとして提供されている行列–ベクトル積を用いた．本節ではこの行列–ベクトル積の方式，プログラムの実装とその性能上のボトルネックについて説明する．

3.1 HElib による行列–ベクトル積の説明

本稿では HElib のテストプログラムである Test_matmul.cpp プログラムを利用した．このプログラムでは [行列]×[暗号化されたベクトル] を行いそれを復号化した値と，[行列]×[ベクトル] の値を比較することで準同型暗号を用いた行列積演算の正答確認を行うプログラムである．本プログラムで調整可能な主要パラメータを表 1 に示す．本プログラムでは，行列及びベクトルのサイズを直接指定することはできず，表 1 中のパラメータ m，p，r からプログラム内で発見的に求められる．以降，デフォルト値と異なるパラメータを用いる場合のみ，その旨示すものとする．

3.2 行列積演算の方法

本プログラムでは SV パッキング，スロットを有効利用する

表 1　Test_matmul.cpp の主要パラメータ

変数	説明	デフォルト値
m	円分多項式 $\Phi(X)$ を決定する	2047
p^r	平文空間の法（p:素数）	2^1
L	modulus chain の積 $< 2^L$ となるよう制限	200
$full$	処理に全ての空間を使う：1，使わない：0	1
nt	スレッド数	1

ために行列積演算を特殊な方法で行なっている．2. 節で説明した通りスロット位置が同じもの同士でのみ演算が可能であることに注意する．下記の行列の要素数を $n \times n$，ベクトルの要素数を n とした行列積演算を用いて説明する．

$$\begin{pmatrix} a_{0,0} & \cdots & a_{0,i} & \cdots & a_{0,n-1} \\ \vdots & \ddots & & & \vdots \\ a_{i,0} & & a_{ii} & & a_{i,n-1} \\ \vdots & & & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,i} & \cdots & a_{n-1,n-1} \end{pmatrix} \times \begin{pmatrix} b_0 \\ \vdots \\ b_i \\ \vdots \\ b_{n-1} \end{pmatrix}$$

この演算で求めたいベクトル c は $c_i = a_{i,0} \times b_0 + a_{i,1} \times b_1 + \cdots + a_{i,n-1} \times b_{n-1} (0 \leq i < n)$ である．行列の要素をどのような組みに分けて考えればスロットが活用できるかを考える．まず行列積演算は次のように変形することができる．簡単のため 3×3 行列，ベクトルを想定する．

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

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

このように行列を斜め方向に分けて取り出した対角行列とベクトルの要素をずらしたベクトルの乗算の和として変形できる．これを元に行列を下のように組みに分けて考える．

$$A_0[a_{0,0}, a_{1,1}, \ldots, a_{n-1,n-1}], A_1[a_{0,n-1}, a_{1,0}, \ldots, a_{n-1,n-2}], \ldots$$

$$\ldots, A_{n-1}[a_{0,1}, a_{1,2}, a_{2,3}, \ldots, a_{n-1,0}]$$

次にベクトルを下のような組みに分けて考える．

$$B_0[b_0, b_1, \cdots_{n-1}], B_1[b_{n-1}, b_0, \ldots, b_{n-2}], \ldots, B_{n-1}[b_1, b_2, \ldots, b_0]$$

$A_0[a_{0,0}, a_{1,1}, \ldots, a_{n-1,n-1}] \times B_0[b_0, b_1, \ldots b_{n-1}]$ を行うと $C_0[a_{0,0} \times b_0, a_{1,1} \times b_1, \ldots, a_{n-1,n-1} \times b_{n-1}]$ を得る．この乗算は同じスロット同士の乗算結果である．同様にして $A_i \times B_i (0 < i < n)$ を行って $C_i (0 < i < n)$ を得る．求めた $C_i (0 \leq i < n)$ を足し合わせると

$$[a_{0,0} \times b_0 + a_{0,1} \times b_1 + \cdots + a_{0,n-1} \times b_{n-1}, \ldots] = [c_0, c_1, \ldots, c_{n-1}]$$

このように求めたいベクトル c をスロットを活用して求めることができる．

行列は $A_0, A_1, \ldots, A_{n-1}$ をそれぞれ多項式変換を用いて求めるのに対して，ベクトルは順番は違うが要素の値は同じなのでまず最初に $B_0[b_0, b_1, \ldots b_{n-1}]$ を求めた後に automorph を行うことで $B_1[b_{n-1}, b_0, \ldots, b_{n-2}], \ldots, B_{n-1}[b_1, b_2, \ldots, b_0]$ を求める．

3.3　HElib による行列積演算の予備評価・解析

ここでは，HElib 付属のテストプログラム Test_matmul.cpp を用いた行列積演算の予備評価，およびその解析結果について述べる．なお，本予備評価で用いた環境を表 2 に記す．

表 2　予備評価環境

マシン名	Intel(R) Xeon(R) E5-2650
動作周波数	2.20GHz
キャッシュ	L1i: 32KB, L1D: 32KB（コアあたり）
	L2: 256KB（コアあたり）
	L3: 30720KB（チップあたり）

3.3.1　ボトルネックの解析

Test_matmul.cpp プログラムの実行時間プロファイルを計測した結果を表 3 に示す．表 3 のように GenKeySWmatrix 関数に約 50–80[%]，DoTest 関数に約 10–30[%] の時間がかかっていることがわかる．GenKeySWmatrix 関数では KeySwitching に用いられる行列の生成が行なわれている．DoTest 関数では行列・ベクトルの encode（多項式化）・暗号化，演算，復号化が行われている．GenKeySWmatrix 関数を 1 回実行することで同じ要素数の行列積演算は何度も行うことができる．

表 3　Test_matmul.cpp のプロファイル結果

(m, p, r)	GenKeySWmatrix [%]	DoTest [%]
(511, 2, 1)	69.22	26.16
(4095, 2, 1)	57.48	34.22
(15709, 2, 1)	67.42	27.78
(15709, 3, 1)	80.83	10.70
(15709, 2, 59)	54.58	35.30
(24295, 2, 1)	68.89	26.39

ここで GenKeySWmatrix 関数と DoTest 関数の詳しい時間計測結果を表 4 及び表 5 にそれぞれ示す．表 4 より，GenKeySWmatrix において complexEvalPoly 関数がほとんどの割合を占めていることが分かる．表 5 では，行列の多項式変換を m_e，ベクトルの多項式変換・暗号化を v_ee, 暗号文の演算を e_exec，結果の復号化を de と表し，DoTest 関数全体の実行時間を 100% としている．

表 3，表 4，及び表 5 より 1 回あたりの実行時間が大きい GenKeySWmatrix 関数の中の complexEvalPoly 関数，複数回実行する DoTest 関数内の暗号文の基本演算を行う演算部を，本稿における高速化対象とした．

3.3.2　GenKeySWmatrix 関数

この関数では，秘密鍵が sk' の暗号文を秘密鍵が sk の暗号文に変換する Key-Switching と呼ばれる操作に必要となる

表 4　GenKeySWmatrix 関数のプロファイル結果

(m, p, r)	complexEvalPoly[%]
(511, 2, 1)	59.3
(4095, 2, 1)	82.9
(15709, 2, 1)	97.9
(15709, 3, 1)	99.7
(15709, 2, 59)	99.4
(24295, 2, 1)	97.8

表 5　DoTest 関数の時間計測結果

(m, p, r)	m_e[%]	v_ee[%]	e_exec[%]	de[%]
(511, 2, 1)	33	3.4	60.3	2.3
(4095, 2, 1)	40.1	1.9	56.1	1.1
(15709, 2, 1)	62.8	6.7	28.3	1.0
(15709, 2, 59)	86.8	1.7	8.0	1.7
(15709, 3, 1)	40.3	18.1	17.9	22.9
(24295, 2, 1)	58.3	6.5	33.6	0.8

Key-Switching-Matrix を生成している．

本プログラムの行列積演算はスロットを活用するために特殊な演算方法（3.2 節参照）を用いる．このとき暗号化したベクトルを automorph を行うことで暗号化したままスロット間で要素の移動を行う．このときに秘密鍵も変化してしまうため元の秘密鍵に戻すために Key-Switching が必要となる．この情報は公開鍵に埋め込まれる．

3.3.3　暗号文の演算

この関数では暗号文の automorph，加算・乗算が行われており，本稿では加算・乗算に着目した．暗号文の加算部分の最下層で行われている演算は，(64bit)long 型 a，b，n を利用して表すと $a+b$ を行い結果が閾値 n を超えていたら $(a+b)-n$ を行うものである．同様に，暗号文の乗算部分の最下層で行われている演算は $a \times b\ (mod\quad n)$ と表わされる．剰余を求める演算子 % を利用せず専用の処理を行う．これを 4.2 節に記す．

3.4　HElib による行列積の評価

準同型暗号による行列積の高速化手法を検討するにあたり，HElib の行列積プログラムに予め実装してあるマルチスレッド化の評価を行った．パラメータは nt のみを変更した．結果を図 1 に示す．図の横軸はスレッド数を示し，縦軸は各スレッド数による実行時間をミリ秒で表す．

図 1 より，スレッド数の増加に伴い実行時間が減少し，8 スレッド使用時には 1 スレッドの時に比べ約 5 倍の性能向上を得られることがわかる．このことから，実装済みのマルチスレッドの効果は十分に得られることを確認した．またキャッシュのプロファイルを測定した結果，L2 キャッシュでのキャッシュミス率が 0.21[%] であった．しかしながら，現状の実装では 3.3 節で述べたボトルネックとなっている箇所で SIMD 化が行われておらず，本稿では GenKeyMatrix と暗号文演算の高速化のアプローチとして SIMD 化を採用した．

4.　HElib 行列積の高速化手法

ここでは，Test_matmul.cpp のボトルネックとなることを

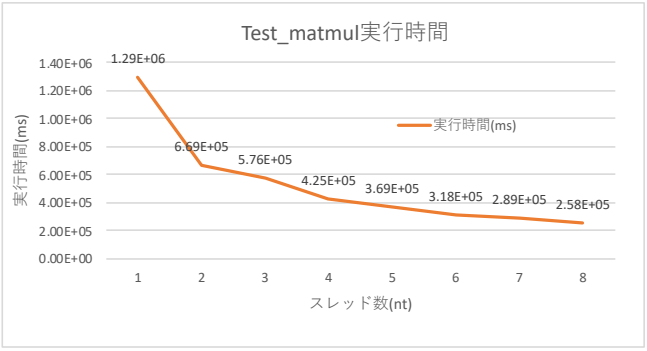


図 1　行列積のマルチスレッドによる測定結果

3.3 節で示した，GenKeySWmatrix 関数と暗号文の加算・乗算関数に用いた高速化手法についてそれぞれ述べる．

4.1　GenKeyMatrix 関数の高速化手法

前節までの測定結果をもとに，GenKeySWmatrix の計算の大部分を占める complexEvalPoly 関数で行われている Horner 法による多項式演算に着目し高速化を行った．Horner 法とは以下の左辺のような多項式を計算するとき，これを右辺のように変形を施す方法のことである．

$$a_n \times\ x^n + a_{n-1} \times\ x^{n-1} + a_{n-2} \times\ x^{n-2} + \cdots + a_1 \times\ x + a_0$$

$$= (\ldots((a_n \times\ x + a_{n-1}) \times\ x + a_{n-2}) \times\ x + \cdots + a_1) \times\ x + a_0$$

このような変形により，乗算の回数を n 回に抑えることができる．GenKeySWmatrix にではこの手法を用いて多項式演算が実装されており，これを SIMD 化により高速化する．

まず，Horner 法による計算をある複数個の項毎に行うことを考える．例として，4 項ずつ行うととし，第 0 項から第 i までの計算結果を A_{0-i} と表記すると，Horner 法による計算は以下のように分割できる:

$$A_{0-3} = a_n \times\ x^3 + a_{n-1} \times\ x^2 + a_{n-2} \times\ x^1 + a_{n-3} \times\ x^0$$

$$A_{0-6} = A_{0-3} \times\ x^3 + a_{n-4} \times\ x^2 + a_{n-5} \times\ x^1 + a_{n-6} \times\ x^0$$

$$A_{0-9} = A_{0-6} + x^3 + a_{n-7} + x^2 + a_{n-8} + x^1 + a_{n-9} + x^0$$

$$\cdots$$

このように計算を分割することにより，各 A_{0-i} を SIMD 化可能となる．

なお，HElib における多項式の係数 a 及び変数 x は複素数で与えられているが，本稿で SIMD 命令セットとして利用した AVX512 には複素数同士の演算を直接行う命令が存在しない．そこで，SIMD 化にあたってそれぞれの実部と虚部を切り分けて実装を行った．さらに，HElib では上記演算を倍精度浮動小数点で行っているが，入力データの性質から単精度で十分な場面が多いと判断し，本稿では単精度で演算することで SIMD 幅を多く確保することとした．

4.2　暗号文の加算・乗算関数の高速化手法

暗号文の加算と乗算はそれぞれ 64 ビット整数型 (long) の配列に対して行われる．暗号文の加算では，与えられた 2 つの配列要素を加算し，結果がある閾値を超えたらその値を減じる．

本稿では，これをマスク演算を用いることで SIMD 化した．

乗算は，long 型は二つの配列の各要素 a，b と閾値 n，及び unsinged int 型の n の逆数 $ninv$ を用いて以下のように処理される．ここで，n 及び $ninv$ は演算前に決定されている．

（1） 精度を保障するために b 及び n を左に 6bit シフトする

（2） $(unsigned\ long)a \times (unsigned\ long)b$ を行い，演算結果を上位 64bit（hi）と下位 64bit（lo，lo'）に分ける

（3） hi を左に 6bit，lo を右に 58bit シフトしてその論理和 H をとる

（4） $H \times ninv$ の結果の上位 64bit（Q）を得る

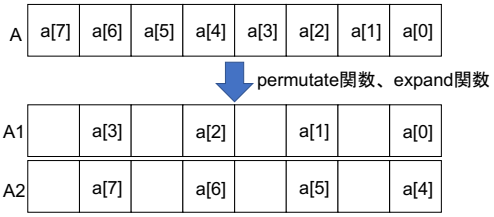
（5） $lo' - Q \times (unsigned\ long)n$ の結果を long 型にキャストする

（6） 最後に得られた結果を 6bit 右にシフトする

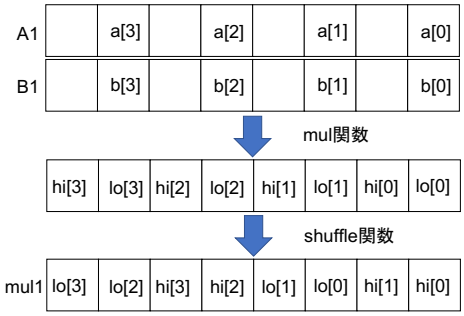
現在の Test_matumul.cpp では上記のような演算が行われているものの，GenKeyMatrix と同様に実際の入力データから 1 要素に 64bit を必要とする場面は少ないと考えられる．このことから，SIMD 幅を確保するため，32bit 整数により SIMD 化を行った．SIMD 化の方法を以下に説明する．実装では AVX512 による SIMD 幅を 16（512/32）としたが，本節では簡単のため SIMD 幅を 8 として説明する．

本手法では配列 $a[0:7]$ と $b[0:7]$ の各要素同士を乗算，これらを法 n のもとでそれぞれの剰余を求める処理を SIMD 化する．ここで重要となるのは SIMD レジスタのどこに $a[i]$ および $b[i]$（$0 \leq i < n$）に関する処理中の値が格納されているかである．

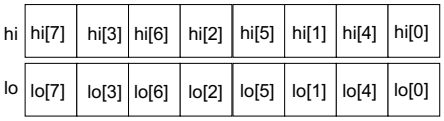
最初に SIMD レジスタ A，B に $a[0:7]$ と $b[0:7]$ の値を代入する．A に permutate 関数及び expand 関数を行い，1 要素おきに SIMD レジスタ $A1$ 及び $A2$ に格納する．B に関しても同様の操作を行い，結果を $B1$ 及び $B2$ に格納する．



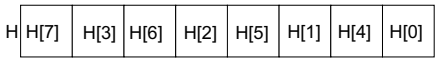
$A1 \times B1$ を行いさらに要素の順番を下図のように入れ替え SIMD レジスタ $mul1$ に格納する．$A2$ と $B2$ にも同じ処理を行い結果を $mul2$ に格納する．



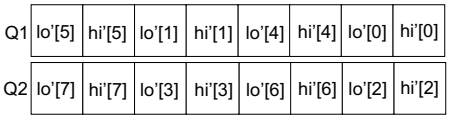
$mul1$ と $mul2$ に unpakedlo 関数，unpakedhi 関数をそれぞれ行い結果を SIMD レジスタ hi，lo に格納する．



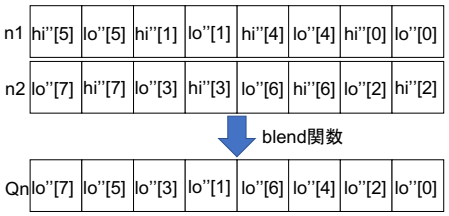
hi を左に 3bit，lo を右に 29bit シフトし，その後 hi と lo の論理積を SIMD レジスタ H に格納する．



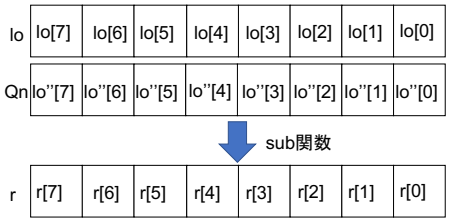
H に A，B のレジスタに行った処理を行い，値を 1 要素おきに $H1$，$H2$ に格納する．そして $H1$ と全ての要素が $ninv$ である SIMD 用ジスタ $ninv'$ の乗算を行い，その結果を shuffle 関数で入れ替えた結果を SIMD レジスタ $Q1$ に格納する．$H2$ にも同様の処理を行い結果を $Q2$ に格納する．



$Q1$，$Q2$ それぞれと全ての要素が n である SIMD レジスタ n' の乗算を行い結果をそれぞれ $n1$，$n2$ に格納する．$n2$ にのみ shuffle 関数を適用し要素を入れ替える．そして $n1$，$n2$ に blend 関数を行い格納したい要素を自分で選びその結果を Qn に格納する．



以前求めた lo と Qn を permutate 関数によって先頭から $a[0]$，$b[0]$ 由来の値，$a[1]$，$b[1]$ 由来の値，.... となるように並び替える．そして lo から Qn を減算して結果を r に格納する．この r に格納された要素が求めたい値となる．



以上のようにして，暗号文同士の加算・乗算を SIMD 化し高速化することが可能となる．

5. 評　　価

ここでは，4. 節で述べた高速化手法について性能評価を行なった結果について述べる．

5.1 性能評価環境

表 6 に，本性能評価で用いた評価環境を記す．本評価では，Intel C++ コンパイラ（Parallel Studio XE 2018）を用いてコンパイルを行なった．本コンパイラによる自動ベクトル化機能

表 6　性能評価環境

マシン名	Intel(R) Xeon(R) W-2145 CPU（8 コア）
動作周波数	3.70GHz
キャッシュ	L1i: 32KB, L1D: 32KB（コアあたり）
	L2: 1024KB（コアあたり）
	L3: 11264KB（チップあたり）
(avx)flags	avx, avx2, avx512f, avx512cd

により，複素数要素の演算において SIMD 命令が用いられていることを確認している．

5.2　GenKeySWmatrix 内 Horner 法部の評価

GenKeySWmatrix の Horner 法部に対して SIMD 化を適用し実行時間を測定した結果を表 7 に示す．本評価では入力パラメータとして $(m, p, r, full)$=(15709, 2, 1, 1) を想定している．Horner 法による多項式演算のプログラムに対して SIMD 化を行った結果，約 3.4 倍の速度向上を得た．前述の通りオリジナルは倍精度浮動小数点を用いて演算していたものを，提案手法では単精度で処理している．準同型暗号による行列積演算の結果に演算精度の変更が及ぼす影響については，今後より詳細に評価・検討を行う必要がある．

表 7　Horner 法部の測定結果

	SIMD 化無し	SIMD 化有り
実行時間 [ms]	0.168	0.050
速度向上率	1	3.4

5.3　暗号化演算部の評価

暗号文化演算に関して，まず加算部を SIMD 化して測定した．測定結果を表 8 に示す．元のコードの測定結果と比較したところ，演算数の多いパラメータ (m, p, r)=(15709, 3, 1) の場合で最大 5.53 倍，演算数の少ないパラメータ (m, p, r)=(511, 2, 1) の場合においても 3.64 倍の速度向上を得た．また SIMD 化した演算結果は正しいことを確認した．

表 8　加算部の測定結果

(m, p, r)	元コード [ms]	SIMD 化 [ms]	速度向上率
(511, 2, 1)	0.804	0.221	3.64
(4095, 2, 1)	7.70	1.73	4.45
(15709, 2, 1)	96.6	20.3	4.76
(15709, 2, 59)	105	20.8	5.05
(15709, 3, 1)	35.5	6.42	5.53
(24295, 2, 1)	132	27.1	4.87

同様に暗号文の乗算部を SIMD 化して計測を行った．ただし，演算精度を SIMD を用いるために long 型から int 型に落としている．測定結果を表 9 に示す．元コードの演算関数を int 型に修正した場合の測定結果と SIMD 化した場合の測定結果を比較したところ，パラメータ (m, p, r)=(24295, 2, 1) の場合に最大で 3.73 倍，パラメータ (m, p, r) = (511, 2, 1) の場合においても最小 2.74 倍の速度向上を得た．乗算においても SIMD 化した演算結果は元コードの演算関数を int 型に修正した場合と同一であることを確認した．

表 9　乗算部の測定結果

(m, p, r)	元コード [ms]	SIMD 化 [ms]	速度向上率
(511, 2, 1)	1.82	0.665	2.74
(4095, 2, 1)	17.8	5.54	3.21
(15709, 2, 1)	179	56.5	3.17
(15709, 2, 59)	191	59.4	3.22
(15709, 3, 1)	65.2	23.5	2.77
(24295, 2, 1)	273	73.1	3.73

6.　ま　と　め

本稿では完全準同型暗号ライブラリ「HElib」を用いた準同型暗号の行列積の演算における高速化のために SIMD 化を提案した．SIMD 化によって準同型暗号の行列積演算における KeySwitchingMatrix を生成する関数の Horner 法計算部では約 3.4 倍，暗号文の演算部の加算部で最大 5.53 倍，乗算部で最大 3.73 倍の速度向上を得ることができ，準同型暗号による行列積高速化の可能性を確認した本手法を HElib 本体に組み込み，行列積全体の評価を行うことが今後の課題である．

謝辞　本研究の一部は科研費挑戦的研究（萌芽）18K19786 の助成により行われた．本研究を行うにあたり助言をいただいた E-JUST の Ahmed El-Mahdy 教授に感謝します．

文　　献

[1] Mitsunari Shigeo. 準同型暗号の実装と Montgomery, Karatsuba, FFT の性能. `https://www.slideshare.net/herumi/x86-64study-shimov6`. Accessed on 2020-01.

[2] 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.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, New York, NY, USA, 2012. Association for Computing Machinery.

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

[5] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in helib. In *Annual International Cryptology Conference*, pp. 93–120. Springer, 2018.

[6] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pp. 97–106, Oct 2011.

[7] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pp. 868–886, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[8] Marie Paindavoine and Bastien Vialla. Minimizing the number of bootstrappings in fully homomorphic encryption. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography – SAC 2015*, pp. 25–43, Cham, 2016. Springer International Publishing.

[9] Fabrice Benhamouda, Tancrède Lepoint, Claire Mathieu, and Hang Zhou. Optimization of bootstrapping in circuits. *CoRR*, Vol. abs/1608.04535, , 2016.

[10] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *Cryptology ePrint Archive, Report 2011/566*.