

# 通信量を削減した浮動小数点演算のための マルチパーティ計算

天田 拓磨<sup>1,†1,a)</sup> 奈良 成泰<sup>1,†1</sup> 西出 隆志<sup>2</sup> 吉浦 裕<sup>1</sup>

受付日 2018年12月9日, 採録日 2019年6月11日

**概要:** ICTの進展にともない, データの重要性はますます高まっており, かつデータ量も大規模になっている. 高精度かつ多量の演算が要求される一方, データに個人情報や機密情報が含まれる場合にはデータの秘匿が重要な課題となる. マルチパーティ計算はデータを秘匿しながら各演算を実行できるため, 安全なデータ利活用を可能にする. 高精度の計算が要求される場面では, マルチパーティ計算の浮動小数点演算が有用であるが, 整数演算のプロトコルや固定小数演算のプロトコルなどに比べて処理コストが非常に大きいという問題がある. 本論文では既存の浮動小数点演算プロトコルをベースに計算結果の桁数に着目して, 小さい法の上で実行可能な浮動小数点加算と乗算プロトコルを提案し, これらのプロトコルの処理コスト削減を実現する.

キーワード:  $(t, n)$  閾値秘密分散, マルチパーティ計算, 浮動小数点演算

## Multiparty Computation for Floating Point Arithmetic with Less Communication over Small Fields

TAKUMA AMADA<sup>1,†1,a)</sup> MASAHIRO NARA<sup>1,†1</sup> TAKASHI NISHIDE<sup>2</sup> HIROSHI YOSHIURA<sup>1</sup>

Received: December 9, 2018, Accepted: June 11, 2019

**Abstract:** We propose new floating-point arithmetic protocols with reduced communication complexity based on previously proposed protocols. Secure computation of floating-point arithmetic is increasingly becoming important where highly precise calculation needs to be carried out without revealing sensitive information. However, the communication complexity of floating-point protocols is large compared with those on integer or fixed-point numbers. We propose new protocols by analyzing the direction for shift operation and decreasing the size of the field to reduce communication complexity of the protocol. When we compute addition of two significands, we shift the significand right and thus we can prevent the bit length from being larger. We also propose a multiplication protocol which can be executed over the small field similarly to our proposed addition protocol.

**Keywords:**  $(t, n)$ -secret sharing, multiparty computation, floating-point arithmetic

### 1. はじめに

クラウドコンピューティングによってビッグデータを解

析し, 解析結果を新規ビジネスの創出や社会問題の解決へと活用することへの期待が高まっている. たとえば, 個人の購買履歴を集積して解析することで商品のレコメンデーションシステムへと応用するといった事例や, 個人の移動履歴を集計して解析した人流データを都市計画やマーケティングへと活用するといった事例などがあげられる. データ解析の結果から新たな価値を生み出し, 社会問題の解決へとつなげることができる反面, 情報漏洩などの安全性の問題が発生する. 解析対象のデータが個人情報や企

<sup>1</sup> 電気通信大学  
University of Electro-Communications, Chofu, Tokyo 182-8585, Japan

<sup>2</sup> 筑波大学  
University of Tsukuba, Tsukuba, Ibaraki 305-8577, Japan

<sup>†1</sup> 現在, 日本電気株式会社  
Presently with NEC Corporation

<sup>a)</sup> t-amada@bk.jp.nec.com

業・国家の機密情報である場合、情報漏洩は重大なインシデントになる。したがって、データを外部に預託し解析する状況においては、データを秘匿しながら処理をする技術が重要になる。データを秘匿したままで解析を実現する手法として、秘密分散法とマルチパーティ計算 (MPC) がある。秘密分散法はデータを複数に分散して管理する手法であり、閾値未満の分散値からは元の秘密情報を復元することができない。また、このときにサーバ間で通信や計算を行うことをマルチパーティ計算 (MPC) という。ただし、秘密分散法では素数  $q$  を法とする整数体上のデータのみ扱うことができる。一方で、解析対象のデータが小数で表現されている場合や、精度の高い計算が要求される場合には、MPC においても小数値を扱う必要がある。MPC において小数値を扱うプロトコルとして、データを固定小数点の形式で表現して四則演算を実行するプロトコルを Catrina らが提案している [10], [11], [12]。また、Catrina らの固定小数点表示をベースにして、データを浮動小数点の形式で表現して四則演算を実行するプロトコルを Aliasgari らが提案している [5]。ただし、浮動小数点の四則演算は整数上での四則演算や固定小数点の四則演算と比較して処理コストが大きい。なかでも、基本演算である浮動小数点加算プロトコルの処理コストが非常に大きい。

本論文では Aliasgari らの浮動小数点表示のプロトコルをもとにして、通信コストを削減した浮動小数点のプロトコルを検討し、途中計算結果のサイズを既存方式よりも小さく設定する手法を 2 通り提案する。1 つは近い値の減算を実行することができないが通信量を約 61.1%削減できる手法で、もう 1 つは演算に制限がなく通信量を約 34.2%削減できる手法である。

## 2. 準備

### 2.1 秘密分散法

秘密分散法は秘密情報を複数の参加者に分散して管理する技術である。秘密分散法の参加者の人数を  $n$  とし、それぞれの参加者を  $\mathcal{P}_1, \dots, \mathcal{P}_n$  とする。このとき、素数  $q$  を法とした有限体  $\mathbb{Z}_q$  上での線形秘密分散法を用いて秘密情報を分散することができる。線形秘密分散法には Shamir の  $(t, n)$  閾値秘密分散法 [24], [25], [27] が広く知られている。 $(t, n)$  閾値秘密分散法では、秘密情報  $s \in \mathbb{Z}_q$  を定数とし、各  $a_i$  を乱数とする多項式  $f(x) = s + a_1x + a_2x^2 + \dots + a_{t-1}x^{t-1} \pmod{q}$  を作り、それぞれの参加者  $\mathcal{P}_i$  には  $f(i)$  を配布する。ここで、任意の  $t$  人以上の参加者の集合からはラグランジュ補間を用いることで多項式  $f(x)$  の係数を一意に定めることができるため、秘密情報  $s = f(0)$  を復元することができる。したがって、参加者  $n$  人中、 $t$  人以上からシェアを集めれば秘密情報を復元することができるが、 $t$  人未満のシェアからは秘密情報を復元することができない。

本論文では秘密分散法に Shamir ベースの  $(t, n)$  閾値秘

密分散法を用い、 $s \in \mathbb{Z}_q$  のシェアを  $[s]_q$  と表す。また、本論文のプロトコルは semi-honest モデルを想定している。

### 2.2 マルチパーティ計算 (MPC)

秘密分散法により値を秘匿し、参加者間で計算や通信を行い、関数値などを求めることをマルチパーティ計算 (MPC) という。

本論文で扱う  $(t, n)$  閾値秘密分散ベースの MPC では有限体  $\mathbb{Z}_q$  上の算術演算を定義することができ、整数シェアの任意の線形演算は各参加者がローカルで計算できる。すなわち、 $[x]_q, [y]_q$  が参加者  $\mathcal{P}_1, \dots, \mathcal{P}_n$  でのシェアであり、 $c$  を定数としたとき、 $[x]_q + [y]_q, c + [x]_q, c[x]_q$  の計算は各  $\mathcal{P}_i$  がローカルに計算できる。一方で、 $[x]_q \times [y]_q$  の計算には参加者間で  $n(n-1)$  回の通信が必要になる。

加算プロトコルと乗算プロトコルを組み合わせ、大小比較や等号判定、ビット分解などの上位プロトコル [14], [23] を構成することができる。また、本論文ではシェア  $[\cdot]_q$  の  $q$  は省略するものとする。

### 2.3 処理コストの評価

乗算は各参加者がローカルで計算できる加算プロトコルなどの線形演算よりも処理コストがはるかに大きい。そこで、MPC プロトコルの処理コストは乗算プロトコルの実行回数により評価し、ローカルで計算できる線形演算は処理コストがないものと見なす [8]。

さらに処理コストには、通信量とラウンド数の 2 つの指標が用いられる。通信量とは、プロトコル中の乗算プロトコルの実行回数の中で、ラウンド数とは乗算プロトコルの実行をできる限り並列実行した際の段数のことである。たとえば、 $[a] \times [b] \times [c] \times [d]$  という処理において、左から順に乗算プロトコルを実行すると、処理コストは通信量 3 回、3 ラウンドとなる。一方、 $[a] \times [b]$  と  $[c] \times [d]$  を並列に実行し、それぞれの結果をかけ合わせると、処理コストは通信量 3 回、2 ラウンドとなる。ラウンド数の評価では、後者の方法で算出する。また、乱数共有はプロトコルの実行とは関係なく実行してもよく、事前計算することができる。プロトコルの処理コストを算出する際に、乱数共有の処理コスト除いた場合の処理コストを、事前計算を実行したときの処理コストとして示すことがしばしばある。

通信量は乗算の実行回数で表されるが、実際にネットワーク上を流れる通信量は秘密分散の法の大きさに比例するため、乗算回数として通信量が同じであっても実際の通信量は異なることがある。本論文では、通信量とラウンド数に加え、乗算回数としての通信量に実行における法をかけ合わせたものも処理コストの指標の 1 つとする。したがって、乗算回数としての通信量を“通信量 (乗算)”と定義し、法の大きさを考慮した通信量を“通信量 (ビット)”と定義する。

## 2.4 既存の MPC プロトコル

本論文で用いる既存の MPC プロトコルについて述べる.

- $[r] \leftarrow \text{RandBit}()$   
乱数ビット  $[r]$  を生成する. このプロトコルには 1 回の乗算を要する [14].
- $a \leftarrow \text{Reveal}([a])$   
シェア  $a$  を公開する. 公開には, ラグランジュ補間の式を用いる. たとえば, 各参加者が自分の保持しているシェアを  $\mathcal{P}_1$  に送信し,  $\mathcal{P}_1$  がラグランジュ補間により値を復元する. 処理コストは乗算 1 回と見なす.
- $[c] \leftarrow \text{OR}([a], [b])$   
 $[a] + [b] - [a][b]$  により, 2 つの入力ビット  $a, b$  の OR を計算する. このプロトコルには 1 回の乗算を要する.
- $[c] \leftarrow \text{XOR}([a], [b])$   
 $[a] + [b] - 2[a][b]$  により, 2 つの入力ビット  $a, b$  の XOR を計算する.
- $([y_1], \dots, [y_n]) \leftarrow \text{PreMul}([x_1], \dots, [x_n])$   
 $n$  個の自然数列  $x_1, \dots, x_n$  の入力について,

$$y_i = \prod_{j=1}^i x_j \quad (1)$$

により  $n$  個の自然数列  $y_1, \dots, y_n$  を出力する. 文献 [11] により, 通信量が  $3n - 1$  でラウンド数が 2 となる. また, 乱数共有を事前計算した際には, 通信量が  $n$  でラウンド数が 1 となる.

- $([y_1], \dots, [y_n]) \leftarrow \text{PreOR}([x_1], \dots, [x_n])$   
 $n$  個のビット列  $x_1, \dots, x_n$  の入力について,

$$y_i = \bigvee_{j=1}^i x_j \quad (2)$$

により  $n$  個のビット列  $y_1, \dots, y_n$  を出力する. 文献 [11] により, 通信量が  $5n - 1$  でラウンド数が 3 となる. また, 乱数共有を事前計算した際には, 通信量が  $2n - 1$  でラウンド数が 2 となる.

- $[c] \leftarrow \text{EQ}([a], [b], \ell)$   
EQ は  $\ell$  ビットの入力  $a, b$  に対する等号判定を行うプロトコルである.  $a = b$  のときに  $c = 1$  となる.  $[c] \leftarrow \text{EQZ}([a'], \ell)$  は  $a' = 0$  のときに  $c = 1$  となるプロトコルであるが,  $[c] \leftarrow \text{EQZ}([a] - [b], \ell)$  と実行することで  $a$  と  $b$  の等号を判定する. 文献 [11] により, EQ プロトコルの処理コストは通信量が  $\ell + 4\lceil \log(\ell) \rceil$  でラウンド数が 4 となる. また, 乱数共有を事前計算した場合には, 通信量が  $\lceil \log(\ell) \rceil + 2$  でラウンド数が 3 となる.
- $[c] \leftarrow \text{LT}([a], [b], \ell)$   
LT は  $\ell$  ビットの入力  $a, b$  に対する大小比較を行うプロトコルである.  $a < b$  のときに  $c$  は 1 となる.  $[c] \leftarrow \text{LTZ}([a'], \ell)$  は  $a' \geq 0$  のとき  $c = 1$  となるプロト

コルであるが,  $[c] \leftarrow \text{LTZ}([a] - [b], \ell)$  と実行することで  $a$  と  $b$  の等号を判定する. 文献 [11] により, LT プロトコルの処理コストは通信量が  $4\ell - 2$  でラウンド数が 4 となる. また, 乱数共有を事前計算した場合には, 通信量が  $\ell + 1$  でラウンド数が 3 となる.

- $([x_{m-1}], \dots, [x_0]) \leftarrow \text{BitDec}([x], \ell, m)$   
BitDec は  $\ell$  ビットの入力  $x$  に対して,  $x$  の下位  $m$  ビットをビットごとのシェアに分解するプロトコルである. 文献 [12] によると, このプロトコルの処理コストは通信量が  $m\lceil \log(m) \rceil$  でラウンド数が  $\lceil \log(m) \rceil$  となる. また, 通信量とのトレードオフでラウンド効率の良いプロトコル [10] も提案されており, この方式を用いると通信量が  $6m$  でラウンド数が 3 である. 乱数共有を事前計算した場合には, 通信量が  $2m + 1$  でラウンド数が 3 となる.
- $[2^a] \leftarrow \text{Pow2}([a], \ell)$   
Pow2 は  $\ell$  ビットの入力  $a$  に対して,  $[2^a]$  を計算するプロトコルである. このプロトコルの内部で BitDec プロトコルが実行されており, 文献 [12] の BitDec プロトコルを用いると, Pow2 プロトコルの処理コストは通信量が  $3\lceil \log \ell \rceil + \lceil \log \ell \rceil \lceil \log \lceil \log \ell \rceil \rceil - 1$  でラウンド数が  $\log \log \ell + 2$  となる. 一方, 文献 [10] の BitDec プロトコルを用いると, Pow2 プロトコルの処理コストは  $9\lceil \log \ell \rceil - 1$  でラウンド数が 5 となる.
- $[y] \leftarrow \text{Trunc}([x], \ell, m)$   
Trunc は  $\ell$  ビットの入力  $x$  に対して,  $\lfloor [x]/2^m \rfloor$  を計算するプロトコルである. 文献 [11] により, このプロトコルの処理コストは通信量が  $4m + 1$  でラウンド数が 4 となる. また, 乱数共有を事前計算した場合には, 通信量が  $m + 2$  でラウンド数が 3 となる.
- $[y] \leftarrow \text{Trunc}([x], \ell, [m])$   
Trunc は  $\ell$  ビットの入力  $x$  に対して,  $\lfloor [x]/2^{[m]} \rfloor$  を計算するプロトコルである. このプロトコルでは前述の  $\text{Trunc}([x], \ell, m)$  プロトコルとは異なり,  $m$  が秘匿されている. プロトコル内部で BitDec プロトコルが実行されており, 文献 [12] の BitDec プロトコルを用いると, この Trunc プロトコルの処理コストは通信量が  $12\ell + \lceil \log \ell \rceil \lceil \log \lceil \log \ell \rceil \rceil + 3\lceil \log \ell \rceil$  でラウンド数が  $\lceil \log \lceil \log \ell \rceil \rceil + 9$  となる. 一方, 文献 [10] の BitDec プロトコルを用いると, Trunc プロトコルの処理コストは  $12\ell + 9\lceil \log \ell \rceil$  でラウンド数が 12 となる.

## 2.5 既存プロトコルの安全性

有限集合  $\Omega$  上の 2 つの確率変数を  $X, Y$  とし, これらの統計的距離を以下のように定義する.

$$\Delta(X, Y) = \frac{1}{2} \sum_{w \in \Omega} |\Pr[X = w] - \Pr[Y = w]| \quad (3)$$

また, 関数  $f(n) : \mathbb{N} \rightarrow \mathbb{R}$  が  $n$  について無視できるとは,

どのような正の多項式  $p(\cdot)$  に対しても、ある整数  $n_p \in \mathbb{N}$  が存在し、 $n > n_p$  であるすべての  $n$  に対して  $f(n) < \frac{1}{p(n)}$  が成り立つことである。  $\Delta(X, Y) = 0$  のときに、確率変数  $X, Y$  の確率分布は完全識別不可能であるといい、  $\Delta(X, Y)$  がセキュリティパラメータ  $\kappa$  について無視できるときに、確率変数  $X, Y$  の確率分布は統計的識別不可能であるという [26]。プロトコルの実行において、攻撃者によって観測されるすべての値を *view* といい、実際に観測される *view* の確率分布と、シミュレーションによって得られる *view* の確率分布が統計的識別不可能なとき、プロトコルは統計的安全という [13]。

本論文では秘密分散の方式に  $(t, n)$  閾値秘密分散法を用いているため、各プロトコルは情報理論的安全 (perfect privacy) または統計的安全 (statistical privacy) を確保する。前節で述べた既存の MPC プロトコルのうち、EQ, LT, Trunc, BitDec については、統計的安全を確保するものである。

### 3. 関連研究

#### 3.1 Aliasgari らの浮動小数点演算

Aliasgari らは浮動小数点演算の MPC プロトコルを提案している [5]。このプロトコルでは、浮動小数を  $(v, p, z, s)$  の4つの変数で表現する。それぞれの変数については、 $v$  を仮数部、 $p$  を指数部、 $z$  をゼロフラグ、 $s$  を符号ビットとしている。 $v$  は  $\ell$  ビット、 $p$  は  $k$  ビットであり、 $z$  は  $v = 0$  のときに  $z = 1$  となり、 $s$  は浮動小数の値が負のときに  $s = 1$  となる。4つの変数  $v, p, z, s$  をそれぞれ  $\mathbb{Z}_q$  上で秘密分散する。ただし、 $v$  の2進数表現を  $(v_{\ell-1}, \dots, v_1, v_0)_2$  としたとき、 $v_{\ell-1} = 1$  となっている。すなわち、 $v$  の  $\ell-1$  ビット目が必ず最上位ビット (MSB) になるように固定されているので、 $v \in [2^{\ell-1}, 2^\ell)$  となる。また、指数部  $p$  は符号付き整数で表されるので  $p \in (-2^{k-1}, 2^{k-1})$  の値である。このとき基数を2とし、浮動小数  $u$  は以下のように表現できる。

$$u = \langle v, p, z, s \rangle = (1 - 2s)(1 - z)v \cdot 2^p \quad (4)$$

##### 3.1.1 浮動小数点加算プロトコル

Aliasgari らが提案している浮動小数点加算プロトコル (FLAdd) の詳細をプロトコル 1 に示す。

浮動小数点加算プロトコルの手順の概略を以下に示す。入力  $u_1 = \langle v_1, p_1, z_1, s_1 \rangle$ ,  $u_2 = \langle v_2, p_2, z_2, s_2 \rangle$  とする。

- (1) 入力  $\langle v_1, p_1, z_1, s_1 \rangle$ ,  $\langle v_2, p_2, z_2, s_2 \rangle$  に対し、EQ プロトコルおよび LT プロトコルを用いて  $v_{max}, v_{min}, p_{max}, p_{min}$  を決定する。このとき、指数部の大小により  $v_{max}$  と  $v_{min}$  を決定する。指数部が同じときは、仮数部の大小により  $v_{max}$  と  $v_{min}$  を決定する。
- (2)  $s = \text{XOR}(s_1, s_2)$  を計算する。 $s = 0$  のとき  $u_1$  と  $u_2$  の和を計算し、 $s = 1$  のとき差を計算する。

#### プロトコル 1 FLAdd

---

**Input:**  $\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle$

**Output:**  $\langle [v], [p], [z], [s] \rangle$  ←

---

- 1:  $[a] \leftarrow \text{LT}([p_1], [p_2], k)$ ;
- 2:  $[b] \leftarrow \text{EQ}([p_1], [p_2], k)$ ;
- 3:  $[c] \leftarrow \text{LT}([v_1], [v_2], \ell)$ ;
- 4:  $[p_{max}] \leftarrow [a][p_2] + (1 - [a])[p_1]$ ;
- 5:  $[p_{min}] \leftarrow (1 - [a])[p_2] + [a][p_1]$ ;
- 6:  $[v_{max}] \leftarrow (1 - [b])([a][v_2] + (1 - [a])[v_1]) + [b]([c][v_2] + (1 - [c])[v_1])$ ;
- 7:  $[v_{min}] \leftarrow (1 - [b])([a][v_1] + (1 - [a])[v_2]) + [b]([c][v_1] + (1 - [c])[v_2])$ ;
- 8:  $[s_3] \leftarrow \text{XOR}([s_1], [s_2])$ ;
- 9:  $[d] \leftarrow \text{LT}(\ell, [p_{max}] - [p_{min}], k)$ ;
- 10:  $[2^\Delta] \leftarrow \text{Pow2}((1 - [d])([p_{max}] - [p_{min}]), \ell + 1)$ ;
- 11:  $[v_3] \leftarrow 2([v_{max}] - [s_3]) + 1$ ;
- 12:  $[v_4] \leftarrow [v_{max}][2^\Delta] + (1 - 2[s_3])[v_{min}]$ ;
- 13:  $[v] \leftarrow ([d][v_3] + (1 - [d])[v_4])2^{\ell} \text{Inv}([2^\Delta])$ ;
- 14:  $[v] \leftarrow \text{Trunc}([v], 2\ell + 1, \ell - 1)$ ;
- 15:  $([u_{\ell+1}], \dots, [u_0]) \leftarrow \text{BitDec}([v], \ell + 2, \ell + 2)$ ;
- 16:  $([h_{\ell+1}], \dots, [h_0]) \leftarrow \text{PreOR}([u_{\ell+1}], \dots, [u_0])$ ;
- 17:  $[p_0] \leftarrow \ell + 2 - \sum_{i=0}^{\ell+1} [h_i]$ ;
- 18:  $[2^{p_0}] \leftarrow 1 + \sum_{i=0}^{\ell+1} 2^i(1 - [h_i])$ ;
- 19:  $[v] \leftarrow \text{Trunc}([2^{p_0}][v], \ell + 2, 2)$ ;
- 20:  $[p] \leftarrow [p_{max}] - [p_0] + 1 - [d]$ ;
- 21:  $[v] \leftarrow (1 - [z_1])(1 - [z_2])[v] + [z_1][v_2] + [z_2][v_1]$ ;
- 22:  $[z] \leftarrow \text{EQZ}([v], \ell)$ ;
- 23:  $[p] \leftarrow ((1 - [z_1])(1 - [z_2])[p] + [z_1][p_2] + [z_2][p_1])(1 - [z])$ ;
- 24:  $[s] \leftarrow (1 - [b])([a][s_2] + (1 - [a])[s_1]) + [b]([c][s_2] + (1 - [c])[s_1])$ ;
- 25:  $[s] \leftarrow (1 - [z_1])(1 - [z_2])[s] + (1 - [z_1])[z_2][s_1] + (1 - [z_2])[z_1][s_2]$ ;
- 26: **return**  $\langle [v], [p], [z], [s] \rangle$

---

- (3)  $\Delta = p_{max} - p_{min}$  とし、Pow2 プロトコルを用いて  $2^\Delta$  を計算する。 $\Delta$  は指数部の差分である。
- (4)  $v = v_{max} \cdot 2^\Delta + (1 - 2s)v_{min}$  を計算する。これは指数部の大きいほうの仮数部を  $\Delta$  だけ左シフトして、仮数部どうしの和を計算している。
- (5)  $v$  をビット分解し、MSB を  $\ell$  ビット目に調節する。
- (6)  $s, p, z$  を決定する。

文献 [5] で Aliasgari らが算出している FLAdd プロトコルの処理コストについては、通信量が  $14\ell + 9k + (\ell + 9)\lceil \log \ell \rceil + \lceil \log \ell \rceil \lceil \log \lceil \log \ell \rceil \rceil + 4\lceil \log k \rceil + 37$  でラウンド数が  $\lceil \log \ell \rceil + \lceil \log \lceil \log \ell \rceil \rceil + 27$  となる。通信量のオーダーが  $\mathcal{O}(\ell \log \ell + k)$ 、ラウンド数のオーダーが  $\mathcal{O}(\log \ell)$  であることに注意されたい。また、秘密分散における法  $q$  は  $q > 2\ell + \kappa + 1$ \*1 である。

一方、BitDec プロトコルに文献 [10] の方式を用いた場合、FLAdd プロトコルの処理コストは通信量が  $20\ell + 9k + 15\lceil \log \ell \rceil + 4\lceil \log k \rceil + 37$  でラウンド数が  $\lceil \log \ell \rceil + 32$  となる。この場合は通信量のオーダーが  $\mathcal{O}(\ell)$  でラウンド数のオーダーが  $\mathcal{O}(\log \ell)$  である。

##### 3.1.2 浮動小数点乗算プロトコル

Aliasgari らが提案している浮動小数点乗算プロトコルを

\*1  $\kappa$  はセキュリティパラメータ。

---

**プロトコル 2** FLMul
 

---

**Input:**  $\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle$ **Output:**  $\langle [v], [p], [z], [s] \rangle$  ←

```

FLMul( $\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle$ )
1:  $[v] \leftarrow [v_1][v_2]$ ;
2:  $[v] \leftarrow \text{Trunc}([v], 2\ell, \ell - 1)$ ;
3:  $[b] \leftarrow \text{LT}([v], 2^\ell, \ell + 1)$ ;
4:  $[v] \leftarrow \text{Trunc}(2[b][v] + (1 - [b])[v], \ell + 1, 1)$ ;
5:  $[z] \leftarrow \text{OR}([z_1], [z_2])$ ;
6:  $[s] \leftarrow \text{XOR}([s_1], [s_2])$ ;
7:  $[p] \leftarrow ([p_1] + [p_2] + \ell - [b])(1 - [z])$ ;
8: return  $\langle [v], [p], [z], [s] \rangle$ 

```

---

プロトコル 2 に示す。

浮動小数乗算プロトコルの手順の概略を以下に示す。

入力は  $u_1 = \langle v_1, p_1, z_1, s_1 \rangle$ ,  $u_2 = \langle v_2, p_2, z_2, s_2 \rangle$  とする。

- (1) 入力  $\langle v_1, p_1, z_1, s_1 \rangle$ ,  $\langle v_2, p_2, z_2, s_2 \rangle$  に対し,  $v = v_1 \cdot v_2$  で仮数部の積を計算する. このとき,  $v$  のビット長は  $2\ell$  ビットまたは  $2\ell - 1$  ビットになっている.
- (2) Trunc プロトコルにより,  $v$  の下位  $\ell - 1$  ビットを切り捨てる.
- (3)  $v$  と  $2^\ell$  との LT プロトコルにより,  $v$  が  $\ell$  ビットなのか  $\ell + 1$  ビットなのか求め, その結果に従って Trunc プロトコルで下位ビットを切り捨てる. この操作では, 出力の  $v$  のビット長を  $\ell$  に調節している.
- (4)  $s, p, z$  を決定する.

FLMul プロトコルの処理コストについて, 通信量は  $8\ell + 10$  でラウンド数は 11 となる.

また, Aliasgari らは文献 [5] で FLAdd, FLMul を提案し, さらに浮動小数大小比較 (FLLT), 浮動小数と固定小数の変換などの上位プロトコルも提案している.

### 3.2 その他の浮動小数点演算

Franz らは 2 パーティで実数の近似値を計算する秘密計算を提案している [15]. データの秘匿には準同型暗号を用いており, 対数ベースでデータを表現しているため, 加算や減算に比べて乗算と除算を簡単に実行できる. Kamm らや Liu らはデータマイニングや機械学習などの様々なアプリケーションへと応用可能なデータ分析に向けた浮動小数点の加算, 乗算, 除算などの四則演算の MPC を提案している [17], [22]. データは IEEE 754 の標準化をベースとして浮動小数を表現し, 秘密分散のフレームワークには加法的秘密分散を用いている. Kamm らはマルチパーティ, Liu らは 2 パーティで実行するプロトコルである. また, Aliasgari らは自身の提案したプロトコルをもとにして, 浮動小数点演算の MPC を Malicious 対応させるプロトコルについても提案している [4]. Kerik らは実世界にスケラブルで頑強性のあるマルチパーティ計算の整数演算および浮動小数点演算を提案している [18]. Kerik らの提案したマルチパーティ計算は秘密分散の方式に  $2^n$  を法とする

環  $\mathbb{Z}_{2^n}$  上の加法型秘密分散を用いている.  $q$  を法とする体  $\mathbb{Z}_q$  上の秘密分散である  $(t, n)$  閾値秘密分散法とは異なり, 乗法逆元を計算できるとは限らないが, 高速にマルチパーティ計算を実行できる [6], [7]. Krips らは固定小数演算と浮動小数点演算を組み合わせ, 高速かつ高精度な実数のマルチパーティ計算を提案している [19]. Krips らはマルチパーティ計算における固定小数演算は高速に実行でき, 浮動小数点演算は高精度かつ柔軟な計算ができるという特徴に着目し, データ保持には浮動小数点の形式をとりつつ可能な限り固定小数の演算を実行することでプロトコルの効率化を図るアプローチをとっている.

## 4. 提案する浮動小数点加算プロトコル

本章では, Aliasgari らの手法 [5] を基にして, 通信量を削減した FLAdd プロトコルについて述べる. FLAdd プロトコルでは, 設定する仮数部のビット長と内部で用いる BitDec プロトコルの種類によって, 効率化可能な方法が異なる. そこで, 2 通りの浮動小数点加算プロトコルを提案する. 従来手法である FLAdd プロトコルの内部で BitDec プロトコルに文献 [12] の手法を用いる場合をケース 1 とし, BitDec プロトコルに文献 [10] の手法を用いる場合をケース 2 とし, それぞれについて通信量を削減する方法について言及する.

### 4.1 ケース 1

BitDec プロトコルに文献 [12] の手法を用いるという前提のもとでは, BitDec プロトコルの実行を回避することで, 通信量を削減したプロトコルを設計することができる. このとき, 通信量を削減できるプロトコルをプロトコル 3 に示す.

#### 4.1.1 手順

プロトコル 3 で示した, 提案プロトコルの実行手順の概略について以下に説明する.

- (1) ステップ 1 からステップ 7 では, 仮数部と指数部の大小を決定し  $[v_{max}], [v_{min}], [p_{max}], [p_{min}]$  とする. ただし, 仮に指数部が同じ値である場合,  $[v_{max}], [v_{min}]$  は仮数部の絶対値の大小により決定する.
- (2) ステップ 8 では仮数部の和を計算するか差を計算するか決定する. 入力の 2 つの浮動小数の符号が同じならば仮数部の和を計算し, 異なるならば仮数部の差を計算する.
- (3) ステップ 9 では指数部の差  $\Delta$  を計算する.
- (4) ステップ 10 では Trunc プロトコルにより  $[v_{min}]$  を  $\Delta - 1$  だけ右シフトする. このとき, Trunc プロトコルの実行において計算される  $2^{[\Delta]}$  についても残しておく.
- (5) ステップ 11 からステップ 12 では  $[v_{min}]$  の下位  $\Delta$  ビットを調べて, 誤差が生じるかをチェックする. この詳

プロトコル 3 Proposed FLAdd1

**Input:**  $\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle$

- 1:  $[a] \leftarrow \text{LT}([p_1], [p_2], k);$
- 2:  $[b] \leftarrow \text{EQ}([p_1], [p_2], k);$
- 3:  $[c] \leftarrow \text{LT}([v_1], [v_2], \ell);$
- 4:  $[p_{max}] \leftarrow [a][p_2] + (1 - [a])[p_1];$
- 5:  $[p_{min}] \leftarrow (1 - [a])[p_2] + [a][p_1];$
- 6:  $[v_{max}] \leftarrow (1 - [b])([a][v_2] + (1 - [a])[v_1]) + [b]([c][v_2] + (1 - [c])[v_1]);$
- 7:  $[v_{min}] \leftarrow (1 - [b])([a][v_1] + (1 - [a])[v_2]) + [b]([c][v_1] + (1 - [c])[v_2]);$
- 8:  $[s_3] \leftarrow \text{XOR}([s_1], [s_2]);$
- 9:  $[\Delta] \leftarrow [p_{max}] - [p_{min}];$
- 10:  $[v'_{min}], 2^{[\Delta]-1} \leftarrow \text{Trunc}([v_{min}], \ell, (1 - [b])([\Delta] - 1));$
- 11:  $[e] \leftarrow [v_{min}] - 2^{[\Delta]} \cdot \text{Trunc}([v'_{min}], \ell, 1);$
- 12:  $[f] \leftarrow \text{EQZ}([e], \ell);$
- 13:  $[v_3] \leftarrow (1 - [b])(2[v_{max}] + (1 - 2[s_3])[v'_{min}]) + [b]([v_{max}] + (1 - 2[s_3])[v'_{min}]);$
- 14:  $[v_3] \leftarrow [v_3] - [s_3](1 - [f]);$
- 15:  $[v_{smb}] \leftarrow \text{Trunc}([v_3], \ell + 2, \ell);$
- 16:  $[v_{amb}] \leftarrow \text{Trunc}([v_{smb}], 2, 1);$
- 17:  $[g] \leftarrow 2^2 - 2[v_{smb}];$
- 18:  $[h] \leftarrow 2^1 - [v_{amb}];$
- 19:  $[v_5] \leftarrow [s_3][g][v_3] + (1 - [s_3])[h][v_3];$
- 20:  $[v_6] \leftarrow \text{Trunc}([v_5], \ell + 2, 2);$
- 21:  $[p_3] \leftarrow [p_{max}] + [s_3](1 - [v_{smb}]) - (1 - [s_3])[v_{amb}];$
- 22:  $[z] \leftarrow \text{EQZ}([v_6], \ell);$
- 23:  $[p] \leftarrow ((1 - [z_1])(1 - [z_2])[p_3] + [z_1][p_2] + [z_2][p_1])(1 - [z]);$
- 24:  $[s] \leftarrow (1 - [b])([a][s_2] + (1 - [a])[s_1]) + [b]([c][s_2] + (1 - [c])[s_1]);$
- 25:  $[s] \leftarrow (1 - [z_1])(1 - [z_2])[s] + (1 - [z_1][z_2])[s_1] + (1 - [z_2])[z_1][s_2];$
- 26: **return**  $\langle [v], [p], [z], [s] \rangle$

細については 4.1.2 および 4.1.5 項に記述している。

- (6) ステップ 13 からステップ 14 では仮数部どうしの加算または減算を実行し、誤差が生じる場合にはキャンセルする。
- (7) ステップ 15 からステップ 20 では仮数部の MSB の位置を調べて MSB を  $\ell$  ビットに設定する。この詳細については 4.1.2 項に記述している。
- (8) 最後に、ステップ 21 からステップ 25 では、仮数部の桁数に合わせて指数部の大きさを決定し、演算結果の符号ビット、ゼロフラグを決定する。

4.1.2 処理コスト削減の基本アイデア

プロトコル 3 において処理コスト削減のポイントは、仮数部の和もしくは差を計算する際に右シフトするという点である。従来プロトコルでは 3.1 節の手順 4 より、 $[v_{max}]$  に  $2^\Delta$  をかけ合わせて  $[v_{max}]$  と  $[v_{min}]$  を足し合わせる。すなわち、 $[v_{max}]$  を左シフトしてから仮数部の和を計算する。一方で、提案プロトコルでは  $v_{min}$  の下位  $\Delta$  ビットを切り捨てて、 $[v_{max}]$  と  $[v_{min}]$  を足し合わせる。すなわち、 $[v_{min}]$  を右シフトしてから仮数部の和を計算する。下位ビットの切り捨てには、Trunc プロトコルを用いる。

仮数部の和を計算する際に、右シフトしてから和を計算することで計算途中の値のビット長が大きくなることを抑制することができる。従来プロトコル中で値のビット長が

表 1 演算と計算結果のビット長との関係

Table 1 Relationship between operation and result.

演算*2	計算後のビット長
加算	$\ell$ ビットまたは $\ell + 1$ ビット
減算 $\Delta > 2$	$\ell$ ビットまたは $\ell - 1$ ビット
$\Delta \leq 1$	0~ $\ell$ ビット

最大になる箇所は、3.1 節の手順 4 で仮数部の和を計算した後に BitDec プロトコルを実行するときであり、 $2\ell + 1$  ビットの仮数部に対して  $2\ell + \kappa + 1$  ビットの乱数を生成する必要がある。MPC においてはプロトコル中のとりうる値が法  $q$  を超えないように設定する必要があるため、従来プロトコルを実行する際には法  $q$  の大きさを  $|q| > 2\ell + \kappa + 1$  と設定しなければならない。

一方、提案プロトコルで値のビット長が最大になるのは、プロトコル 3 のステップ 15 である。このとき、Trunc プロトコルに  $\ell + 2$  ビットの値を入力するが、Trunc プロトコル内部では  $\ell + \kappa + 2$  ビットの乱数を生成する。したがって、提案プロトコルを実行する際には法の大きさを  $|q| > \ell + \kappa + 2$  以上に設定すればよい。また、MPC の通信量は法の大きさに比例するため、法の大きさを考慮した通信量である“通信量 (ビット)”の指標において、提案プロトコルは通信量を削減できる。

また、BitDec プロトコルに文献 [12] の手法を用いると、BitDec プロトコルの通信量が  $\mathcal{O}(\ell \log \ell)$  となり、浮動小数点の加算演算全体の処理コストとしてみて最もオーダの大きい箇所となっている。したがって、ケース 1 における提案プロトコルでは BitDec プロトコルの実行を回避する。

従来手法では、仮数部の和や差を計算する際に左シフトしていたため、3.1 節の手順 4 で計算した  $v$  は 0 から  $2\ell$  ビットまでのすべての値をとりうる。プロトコルの入出力の仮数部ビットはつねに MSB が  $\ell - 1$  ビット目に設定されている必要があるため、 $v$  の MSB を  $\ell - 1$  ビット目に設定する必要があり、そのために  $\ell$  ビットの  $v$  に対して BitDec プロトコルを実行する必要がある。一方で、提案手法では提案プロトコルではポイント 1 により仮数部は  $\ell$  ビットの値と  $\ell - \Delta$  ビットの値の和または差を計算する。仮数部の加算を実行する場合および  $\Delta > 1$  で仮数部の減算を実行する場合には、計算結果のビット長を予測することができる。各演算と計算結果のビット長との関係を表 1 に示す。

例として、仮数部の加算実行時に BitDec プロトコルを実行せずに MSB の位置を求める方法について説明する。 $\ell$  ビットの値と  $\ell - \Delta$  ビットの値の加算実行結果を  $v$  とすると、表 1 より  $v$  のビット長は  $\ell$  ビットまたは  $\ell + 1$  ビットになっている。このとき、 $v$  に Trunc プロトコルを実行することで MSB を調べることができる。すなわち、 $\text{Trunc}([v], \ell + 1, \ell) = [1]$  のとき  $v$  は  $\ell + 1$  ビットであり、

\*2 入力  $\ell$  ビットの  $v_{max}$  と  $\ell - \Delta$  ビットの  $v_{min}$ 。

表 2 ケース 1 浮動小数点加算プロトコルにおける処理コスト比較  
**Table 2** Case 1: Complexity comparison of floating-point addition protocol.

	通信量	ラウンド
従来	$14\ell + 9k + (\ell + 9)\lceil \log \ell \rceil$ $+ \lceil \log \ell \rceil \lceil \log \lceil \log \ell \rceil \rceil$ $+ 4\lceil \log k \rceil + 37$	$\lceil \log \lceil \log \ell \rceil \rceil +$ $\lceil \log \ell \rceil + 27$
提案	$22\ell + 5k + 11\lceil \log \ell \rceil$ $+ \lceil \log \ell \rceil \lceil \log \lceil \log \ell \rceil \rceil$ $+ 4\lceil \log k \rceil + 59$	$\lceil \log \lceil \log \ell \rceil \rceil + 44$

表 3 ケース 1 浮動小数点加算プロトコルにおける処理コスト比較 (事前計算あり)

**Table 3** Case 1: Complexity comparison of floating-point addition protocol after precomputation.

	通信量	ラウンド
従来	$6\ell + 2k + (\ell + 4)\lceil \log \ell \rceil$ $+ \lceil \log \ell \rceil \lceil \log \lceil \log \ell \rceil \rceil$ $+ \lceil \log k \rceil + 72$	$\lceil \log \lceil \log \ell \rceil \rceil$ $+ \lceil \log \ell \rceil + 27$
提案	$7\ell + k + 4\lceil \log \ell \rceil$ $+ \lceil \log k \rceil + 66$	$\lceil \log \lceil \log \ell \rceil \rceil + 44$

$\text{Trunc}(\lceil v \rceil, \ell + 1, \ell) = [0]$  のとき  $v$  は  $\ell$  ビットであると分かる。したがって、 $\text{Trunc}$  プロトコルを用いることで  $\text{BitDec}$  プロトコルの実行を回避して  $v$  の MSB を求めることができる。

このとき、通信量が  $\mathcal{O}(\ell \log \ell)$  である  $\text{BitDec}$  プロトコルを通信量が  $\mathcal{O}(\ell)$  である  $\text{Trunc}$  プロトコルで置き換えることができ、通信量を削減することができる。

#### 4.1.3 処理コスト

従来プロトコルと提案プロトコルについて、処理コストを比較する。ケース 1 における処理コストを比較したものを表 2 に示す。また、乱数共有を事前計算した際の処理コストを比較したものを表 3 に示す。

表 2 より、ケース 1 において従来プロトコルと比べて通信量は  $\mathcal{O}(\ell \log \ell + k)$  から  $\mathcal{O}(\ell + k)$  となり、ラウンド数は  $\mathcal{O}(\log \ell)$  から  $\mathcal{O}(\log \log \ell)$  となっていることが分かる。表 3 より、ケース 1 において事前計算を実行した場合、従来プロトコルと比べて通信量は  $\mathcal{O}(\ell \log \ell + k)$  から  $\mathcal{O}(\ell + k)$  となり、ラウンド数は  $\mathcal{O}(\log \ell)$  から  $\mathcal{O}(\log \log \ell)$  となっていることが分かる。

ここで、IEEE 754 で標準化されている単精度浮動小数点表示および倍精度浮動小数点表示を参考に、単精度の場合は  $\ell = 32$ ,  $k = 8$  とし、倍精度の場合は  $\ell = 64$ ,  $k = 11$  とし通信量の具体値を算出する。また、乗算回数である通信量に法の大きさを掛け合わせたものを通信量 (ビット) と定義する。通信量 (乗算) と通信量 (ビット) について乱数共有を事前計算する場合としない場合で比較し、表 4 および表 5 に示す。法に関しては、従来プロトコルでの法を  $q$ 、提案プロトコルでの法を  $q'$  とすると  $|q'| = 2\ell + \kappa + 1$ ,

表 4 ケース 1 浮動小数点加算プロトコルにおける処理コストの具体値の比較

**Table 4** Case 1: Examples of complexity comparison of floating-point addition protocol in single and double precision.

		通信量 (乗算)	通信量 (ビット)	ラウンド
単精度	従来	789	82,845	35
	提案	885	65,490	47
倍精度	従来	1,504	254,176	36
	提案	1,622	171,932	47

表 5 ケース 1 浮動小数点加算プロトコルにおける処理コストの具体値の比較 (事前計算あり)

**Table 5** Case 1: Examples of complexity comparison of floating-point addition protocol in single and double precision after precomputation.

		通信量 (乗算)	通信量 (ビット)	ラウンド
単精度	従来	478	50,190	35
	提案	321	23,754	47
倍精度	従来	908	153,452	36
	提案	553	58,618	47

$|q'| = \ell + \kappa + 2$  とする。セキュリティパラメータに関しては、既存の MPC 実行エンジンや既存の論文内での実装で用いられている値を参考に決定する。文献 [5] において、Aliasgari らは実装評価時に統計的安全を確保するセキュリティパラメータとして  $\kappa = 48$  を採用している。また、秘密計算実行エンジンである VIFF [16] においてはデフォルトのセキュリティパラメータの値として  $\kappa = 30$  としており、同様に SPDZ-2 [3], SCALE-MAMBA [2], MP-SPDZ [1] の実行エンジンにおいてはデフォルト値に  $\kappa = 40$  としている。以上をふまえて、セキュリティパラメータ  $\kappa$  は実装を想定したときの妥当な値として  $\kappa = 40$  と設定する。

単精度においては、表 4 より提案プロトコルでは通信量 (ビット) を約 20.9%削減でき、表 5 より事前計算を実行した場合は通信量 (ビット) を約 52.7%削減できることが分かる。倍精度においては、表 4 より提案プロトコルでは通信量 (ビット) を約 32.4%削減でき、表 5 より事前計算を実行した場合は通信量 (ビット) を約 61.8%削減できることが分かる。仮数部のビット長である  $\ell$  が大きいときかつ事前計算を実行したときに、コストの削減率が大きくなることが分かる。

ラウンド数については、従来プロトコルから提案プロトコルにおいてオーダで見ると  $\mathcal{O}(\log \ell)$  から  $\mathcal{O}(\log \log \ell)$  へと削減できているが、定数項が大きくなったため増加することが分かる。

#### 4.1.4 安全性

提案プロトコル 3 では、共通してデータの秘匿に秘密分散法を用いており、値の計算には MPC を用いている。プロトコル中で途中の値を公開することはなく、既存プロト

コルの組合せのみからなる。したがって、プロトコル全体の安全性は線形秘密分散とそれに基づく MPC の安全性に帰着する。

ただし、プロトコル内部で用いている LT, EQ, Trunc プロトコルでは実行途中の値に乱数を加えて公開している。シェア  $[a]$  と乱数  $[r]$  に対して  $b = a + r$  を公開するとき、 $a \in [0, 2^\ell)$  に対し乱数  $r$  は  $r \in [0, 2^{\ell+\kappa})$  の範囲をとる。法を  $q$  を  $q > 2^{\ell+\kappa+1}$  としたとき、 $b$  と  $r$  の統計的距離  $\Delta(r, b)$  は、

$$\Delta(r, b) < 2^{-\kappa} \quad (5)$$

となる。これは、統計的距離  $\Delta(r, b)$  がセキュリティパラメータ  $\kappa$  について無視できることになるので、統計的安全であるといえる。このときプロトコル全体は統計的安全となる。LT, EQ, Trunc プロトコルが統計的安全であるので、提案プロトコル全体の安全性も統計的安全となる。これは、従来手法である Aliasgari らの浮動小数点加算プロトコルと同じ水準の安全性である。

#### 4.1.5 誤差の評価

本項では、誤差について言及する。ただし、ここでの誤差とは従来プロトコルと提案プロトコルの実行における出力結果の差分のことである。仮数部の加算を実行する場合と減算を実行する場合で、誤差の生じ方が異なるので、それぞれについて言及する。

##### 4.1.5.1 加算実行時

従来プロトコルでは  $v_{max}$  を  $\Delta$  だけ左シフトして仮数部の和を計算する。このとき、 $[v_{max}][2^\Delta]$  の下位  $\Delta$  ビットはすべて 0 なので、仮数部の和を計算する際、下位  $\Delta$  ビットではキャリー（桁上がり）が発生しない。仮数部の和は  $\ell + \Delta$  ビットであり、仮数部を  $\ell$  ビットに設定する必要があるため、下位  $\Delta$  ビットは切り捨てられる。したがって、下位  $\Delta$  ビットを切り捨てたことによって、 $\Delta$  ビットよりも上位に影響を及ぼすことはない。

一方で提案プロトコルでは、はじめに  $[v_{min}]$  の下位  $\Delta$  ビットを切り捨てる。 $[v_{max}]$  と  $[v_{min}]/2^\Delta$  の和は、切り捨てた  $[v_{min}]$  の下位  $\Delta$  ビットは上位  $\ell$  ビットの計算に影響しないことが分かる。

したがって、加算実行時には従来プロトコルと提案プロトコルで出力結果の誤差は生じない。

##### 4.1.5.2 減算実行時

従来プロトコルでは  $[v_{max}]$  を  $\Delta$  だけ左シフトして  $[v_{min}]$  との差を計算する。このとき、 $[v_{max}][2^\Delta]$  の下位  $\Delta$  ビットはすべて 0 なので、 $[v_{min}]$  の下位  $\Delta$  ビットが 0 ではないときに桁借りが発生する。下位で桁借りが発生した場合には、その桁よりも上位で初めて 1 が立つ桁まで影響を及ぼす。仮数部の差を計算後、下位  $\Delta$  ビットは切り捨てられる。したがって、下位  $\Delta$  ビットの計算で桁借りが発生している場合、仮数部の差を計算後に  $\Delta$  ビットよりも上位桁に

影響を及ぼす。

一方で提案プロトコルでは、はじめに  $[v_{min}]$  の下位  $\Delta$  ビットを切り捨てる。加算実行時と同様に  $[v_{max}]$  と  $[v_{min}]/2^\Delta$  の差においても、切り捨てた  $[v_{min}]$  の下位  $\Delta$  ビットは  $\Delta$  ビットよりも上位桁の計算に影響しない。

したがって、減算実行時に  $[v_{min}]$  の下位  $\Delta$  ビットがすべて 0 でないときは、差の計算結果の上位  $\ell$  ビットに 1 ビットの誤差が生じる。そこで、提案プロトコルではプロトコル 3 のステップ 10–12 において  $[v_{min}]$  の下位  $\Delta$  ビットが 0 以上かをチェックして、誤差がある場合にはキャンセルできるように設計している。

以上より、従来プロトコルと提案プロトコルにおいて出力結果の誤差は生じない。

#### 4.1.6 提案プロトコル 3 の有効性

提案プロトコルのケース 1 では、通信量を大きく削減することができるが、実行可能な計算が限定される。2 つの入力  $u_1, u_2$  に対し、加算実行時と減算実行時かつ  $u_1, u_2$  の絶対値の差が大きい場合には本論文の提案プロトコルが適用できる。しかし、減算かつ指数部の差  $\Delta$  が  $\Delta \leq 1$  のときには、仮数部の計算結果の桁数を絞り込むことができないため、提案プロトコルでは処理できない。提案プロトコルが有効なのは、実際のデータ解析などの実行を想定したときに、(1) 非負値データしか扱わないケースや (2) 減算の実行回数が他演算の実行回数に比べて少ないケースである。

(1) の例として、非負値行列因子分解 (NMF) があげられる [20], [28]。NMF は行列分解の一手法であるが、入力を要素が非負値の行列とし加算と乗算と除算のみで構成されている。他にも近似計算やクロス集計表を作成する処理などが (1) の例にあたる。

(2) の例としては、除算の近似計算があげられる。除算のアルゴリズムであるゴールドシュミット法において二項定理を用いた方法では、除数を  $D$  とし被除数を  $N$  としたとき、商  $Q = \frac{N}{D}$  を求める。  $D \in (\frac{1}{2}, 1]$  となるように  $\frac{N}{D}$  を 2 の冪乗でスケールリングし、  $D = 1 - x$  となるように  $x$  を求め、  $F_i = 1 + x^{2^i}$  とし  $\frac{N}{D}$  の除数と被除数に  $F_i$  をかけていく。  $\frac{N}{1-x} = \frac{N \cdot (1+x)}{1-x^2} = \frac{N \cdot (1+x)(1+x^2)}{1-x^4} = \frac{N \cdot (1+x)(1+x^2)(1+x^4)}{1-x^8}$  と計算することで分母は 1 に収束し商  $Q$  を求めることができる。この計算においては、  $\frac{N}{D}$  のスケールリングで乗算を複数回実行し、  $F_i$  の計算に乗算と加算を実行し、  $x$  の計算に減算を 1 回実行し、商の計算には乗算を複数回実行する。一連の計算において、乗算と加算を複数回実行する一方で減算の実行回数は 1 回である\*3。したがって、減算以外は小さい法の上で実行可能である。この計算において、減算実行前に法の変換をして減算のみ従来プロトコルを実行し、その他の加算や乗算は小さい法の上で計算可能な提

\*3 乗算については、小さい法で実行可能なプロトコルを 5 章で提案する。

プロトコル 4 Proposed FLAdd2

**Input:**  $\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle$

- 1:  $[a] \leftarrow \text{LT}([p_1], [p_2], k)$ ;
- 2:  $[b] \leftarrow \text{EQ}([p_1], [p_2], k)$ ;
- 3:  $[c] \leftarrow \text{LT}([v_1], [v_2], \ell)$ ;
- 4:  $[p_{max}] \leftarrow [a][p_2] + (1 - [a])[p_1]$ ;
- 5:  $[p_{min}] \leftarrow (1 - [a])[p_2] + [a][p_1]$ ;
- 6:  $[v_{max}] \leftarrow (1 - [b])([a][v_2] + (1 - [a])[v_1]) + [b]([c][v_2] + (1 - [c])[v_1])$ ;
- 7:  $[v_{min}] \leftarrow (1 - [b])([a][v_1] + (1 - [a])[v_2]) + [b]([c][v_1] + (1 - [c])[v_2])$ ;
- 8:  $[s_3] \leftarrow \text{XOR}([s_1], [s_2])$ ;
- 9:  $[\Delta] \leftarrow [p_{max}] - [p_{min}]$ ;
- 10:  $[v'_{min}], 2^{[\Delta]-1} \leftarrow \text{Trunc}([v_{min}], \ell, (1 - [b])([\Delta] - 1))$ ;
- 11:  $[e] \leftarrow [v_{min}] - 2^{[\Delta]} \cdot \text{Trunc}([v'_{min}], \ell, 1)$ ;
- 12:  $[f] \leftarrow \text{EQZ}([e], \ell)$ ;
- 13:  $[v_3] \leftarrow (1 - [b])(2[v_{max}] + (1 - 2[s_3])[v'_{min}]) + [b]([v_{max}] + (1 - 2[s_3])[v'_{min}])$ ;
- 14:  $[v_3] \leftarrow [v_3] - [s_3](1 - [f])$ ;
- 15:  $([u_\ell], \dots, [u_0]) \leftarrow \text{BitDec}([v_3], \ell + 1, \ell + 1)$ ;
- 16:  $([h_\ell], \dots, [h_0]) \leftarrow \text{PreOR}([u_\ell], \dots, [u_0])$ ;
- 17:  $[p_0] \leftarrow \ell + 1 - \sum_{i=0}^{\ell+1} [h_i]$ ;
- 18:  $[2^{p_0}] \leftarrow 1 + \sum_{i=0}^{\ell} 2^i(1 - [h_i])$ ;
- 19:  $[v] \leftarrow \text{Trunc}([2^{p_0}][v], \ell + 1, 1)$ ;
- 20:  $[p] \leftarrow [p_{max}] - [p_0] + 1 - [d]$ ;
- 21:  $[v] \leftarrow (1 - [z_1])(1 - [z_2])[v] + [z_1][v_2] + [z_2][v_1]$ ;
- 22:  $[z] \leftarrow \text{EQZ}([v_6], \ell)$ ;
- 23:  $[p] \leftarrow ((1 - [z_1])(1 - [z_2])[p_3] + [z_1][p_2] + [z_2][p_1])(1 - [z])$ ;
- 24:  $[s] \leftarrow (1 - [b])([a][s_2] + (1 - [a])[s_1]) + [b]([c][s_2] + (1 - [c])[s_1])$ ;
- 25:  $[s] \leftarrow (1 - [z_1])(1 - [z_2])[s] + (1 - [z_1])[z_2][s_1] + (1 - [z_2])[z_1][s_2]$ ;
- 26: **return**  $\langle [v], [p], [z], [s] \rangle$

案プロトコルを実行するなどの工夫をすることで、一連の計算全体を従来プロトコルで実行するよりも効率化できることが予想される。

4.2 ケース 2

BitDec プロトコルに文献 [10] の手法を用いたときに通信量を削減できるプロトコルをプロトコル 4 に示す。このプロトコルはケース 1 とは異なり、通信量を削減できかつ実行できる演算に制限がない。

4.2.1 手順

プロトコル 4 で示した、提案プロトコルの実行手順の概略について説明する。

- (1) ステップ 1 からステップ 14 まではケース 1 と同様の処理である。
- (2) ステップ 15 からステップ 20 では BitDec プロトコルでビット列のシェアに分解し、仮数部の MSB の位置を調べて MSB を  $\ell$  ビットに設定する。このとき、従来手法であるプロトコル 1 と同様に、BitDec プロトコルを実行して MSB を  $\ell$  ビット目に正規化する。
- (3) 最後に、ステップ 21 からステップ 25 では、仮数部の桁数に合わせて指数部の大きさを決定し、演算結果の符号ビット、ゼロフラグを決定する。

表 6 ケース 2 浮動小数点加算プロトコルにおける処理コスト比較  
Table 6 Case 2: Complexity comparison of floating-point addition protocol.

	通信量	ラウンド
従来	$20\ell + 9k + 15\lceil \log \ell \rceil + 4\lceil \log k \rceil + 37$	$\lceil \log \ell \rceil + 32$
提案	$22\ell + 5k + 13\lceil \log \ell \rceil + 4\lceil \log k \rceil + 64$	42

表 7 ケース 2 浮動小数点加算プロトコルにおける処理コスト比較 (事前計算あり)

Table 7 Case 2: Complexity comparison of floating-point addition protocol after precomputation.

	通信量	ラウンド
従来	$8\ell + 2k + 6\lceil \log \ell \rceil + \lceil \log k \rceil + 74$	$\lceil \log \ell \rceil + 32$
提案	$9\ell + k + 5\lceil \log \ell \rceil + \lceil \log k \rceil + 59$	42

4.2.2 処理コスト削減の基本アイデア

プロトコル 4 で適用している処理コスト削減のポイントは、プロトコル 3 で採用している手法と同様で、仮数部の和もしくは差を計算する際に右シフトするという点である。このとき、提案プロトコルを実行する際の法の大きさを  $|q'| > \ell + \kappa + 1$  と設定することができる。また、MPC の通信量は法の大きさに比例するため、法の大きさを考慮した通信量である“通信量 (ビット)”の指標において、提案プロトコルは通信量を削減できる。

4.2.3 処理コスト

従来プロトコルと提案プロトコルについて、処理コストを比較する。ケース 2 における処理コストを比較したものを表 6 に示す。また、乱数共有を事前計算した際の処理コストを比較したものを表 7 に示す。

表 6 より、ケース 2 において従来プロトコルに比べてラウンド数が  $\mathcal{O}(\log \ell)$  から  $\mathcal{O}(1)$  となっていることが分かる。また、表 7 より、ケース 2 において事前計算を実行した場合、従来プロトコルに比べてラウンド数が  $\mathcal{O}(\log \ell)$  から  $\mathcal{O}(1)$  となっていることが分かる。

ここで、ケース 1 同様 IEEE 754 で標準化されている単精度浮動小数点表示および倍精度浮動小数点表示を参考に、単精度の場合は  $\ell = 32$ ,  $k = 8$  とし、倍精度の場合は  $\ell = 64$ ,  $k = 11$  として通信量の具体値を算出する。通信量には、乗算回数である通信量 (乗算) に法の大きさを掛け合わせた通信量 (ビット) を用いて比較する。法に関しては、従来プロトコルでの法を  $q$ 、提案プロトコルでの法を  $q'$  とすると  $|q| = 2\ell + \kappa + 1$ ,  $|q'| = \ell + \kappa + 2$  とする。ただし、4.1.3 項と同様にセキュリティパラメータ  $\kappa$  は実装を想定したときの妥当な値として  $\kappa = 40$  と設定する [21]。

表 4 から表 9 までで通信量 (ビット) の指標において、従来手法より提案手法が効率が良いことが分かる。単精度においては、提案プロトコルによって通信量 (ビット) を約 25.4%削減でき、倍精度においては通信量 (ビット) を約 33.2%削減できることが分かる。また、事前計算を実行し

表 8 ケース 2 浮動小数点加算プロトコルにおける処理コストの具体値の比較

Table 8 Case 2: Examples of complexity comparison of floating-point addition protocol in single and double precision.

		通信量 (乗算)	通信量 (ビット)	ラウンド
単精度	従来	836	87,780	37
	提案	885	65,490	42
倍精度	従来	1,522	257,218	38
	提案	1,621	171,826	42

表 9 ケース 2 浮動小数点加算プロトコルにおける処理コストの具体値の比較 (事前計算あり)

Table 9 Case 2: Examples of complexity comparison of floating-point addition protocol in single and double precision after precomputation.

		通信量 (乗算)	通信量 (ビット)	ラウンド
単精度	従来	379	39,795	37
	提案	383	28,342	42
倍精度	従来	648	109,512	38
	提案	680	72,080	42

た場合には単精度において通信量 (ビット) を約 28.8%削減でき、倍精度においては通信量 (ビット) を約 34.2%削減できることが分かる。仮数部のビット長である  $l$  が大きいときかつ事前計算を実行したときに、処理コストの削減率が大きくなることが分かる。

#### 4.2.4 安全性

提案プロトコル 4 では、途中の値を公開することはなく既存プロトコルの組合せのみからなる。したがって、プロトコル全体の安全性は線形秘密分散とそれに基づく MPC の安全性に帰着する。ただし、プロトコル内部で用いている LT, EQ, Trunc プロトコルでは実行途中の値に乱数を加えて公開しているので、ケース 1 と同様にプロトコル全体は統計的安全となる。

#### 4.2.5 誤差の評価

ケース 2 において誤差が生じる箇所は Trunc プロトコルで下位ビットを切り捨てる箇所であり、そこまでの処理はケース 1 と同様である。したがって、4.1.5 項より従来プロトコルと提案プロトコルの間での誤差は発生しない。

#### 4.2.6 提案プロトコル 4 の有効性

提案プロトコルのケース 2 ではケース 1 とは異なり通信量を削減でき、かつ実行可能な演算を限定しない。ただし、ケース 1 ほど処理コスト削減率は高くないという特徴がある。したがって、加算と減算が入り混じった計算で、かつ固定小数点演算ではオーバーフローする可能性のある場合にケース 2 のプロトコルが適しているといえる。

### 4.3 提案プロトコルの位置づけ

ケース 1 とケース 2 のそれぞれで浮動小数点加算の従来

表 10 浮動小数点加算プロトコルにおける従来手法と提案手法の比較

Table 10 Comparison between previous floating-point addition protocol and proposed one.

	従来 C1	従来 C2	提案 C1	提案 C2
通信量 (乗算)	$\Delta$ 908	$\circ$ 648	$\odot$ 564	$\circ$ 680
通信量 (ビット)	$\Delta$ 153,452	$\Delta$ 109,512	$\odot$ 59,784	$\circ$ 72,080
ラウンド	$\circ$ 36	$\circ$ 38	$\Delta$ 47	$\Delta$ 42
適用範囲	$\circ$	$\circ$	$\Delta$	$\circ$

手法と提案手法の処理コストおよび適用可能性を表 10 に示す。ただし、ケース 1 が BitDec プロトコルに文献 [12] を用いることを前提とした場合、ケース 2 が BitDec プロトコルに文献 [10] をそれぞれ用いることを前提とした場合で、ケース 1 の提案手法では BitDec プロトコルの実行を回避したプロトコルとなっている。表中の C1, C2 はケース 1, ケース 2 を表しており、各具体値は倍精度かつ事前計算を実行した際の値である。表 10 より通信量 (ビット) の指標で見たときに、提案手法がどの従来手法よりも効率が良く、特に提案ケース 1 の場合が最も効率が良いことが分かる。ただし、提案ケース 1 の場合は  $\Delta \leq 1$  かつ減算を実行することができない。また、ラウンド数で比較すると提案手法の方が大きい、大規模データの演算を実行する場合にはラウンド数よりも通信量が支配的になることが知られている [29]。

文献 [9] では 12-core Intel® Xeon® プロセッサ、48 GB メモリ搭載のサーバ 3 台を 1 Gbps の通信まで可能なネットワークカードを使用して同一 LAN 内で接続し、SHARE-MIND の実行エンジンで MPC プロトコルを実測している。また、文献 [9] の付録 C では演算における入力並列度を 1 から  $10^8$  まで変化させてプロトコルの実行時間を測定している。グラフ群が表しているように、プロトコルによらず、並列度がある一定 (変曲点) までは並列度に対して実行時間がほぼ横ばいである一方、並列度がある一定より大きくなると並列度に比例して実行時間が長くなる。また、文献 [9] の表 1, 2 から、並列度が 1 ときにはラウンド数の大きいプロトコルほど実行時間が長くなっていることが分かる。ShiftR プロトコルと BitExtr プロトコルを比較すると、ラウンド数は同じだが通信量は BitExtr の方が大きい。このとき、並列度 1 のときの実行時間を比較するとほぼ同じであるが、並列度  $10^7$  のときには BitExtr の方が実行時間が長いことが分かる。すなわち、入力並列度が小さい場合にはプロトコルのラウンド数が実行時間に対して支配的である一方、並列度が大きい場合には通信量が実行時間に対して支配的になることが分かる。また、実行時間の変曲点は 1 ラウンドあたりの通信量によって決まるた

め、プロトコルによって異なる。1 ラウンドあたり通信量が大きいほど変曲点の並列度の値は小さくなる。

文献 [9] では秘密分散法に加法的秘密分散を用いているため、シェアのビット長  $n$  に対して 1 ラウンドあたりに  $15n$  ビットの総通信量を要する。ビット長が 32 ビットのときには整数型の乗算プロトコルに 480 ビットの通信量を要する。このとき、乗算プロトコルを実行する場合には、入力の変曲度が 15,000 以上でネットワークトラフィックの帯域幅の上限を上回ることが分かる。同様に我々の提案プロトコルを SHAREMIND の実行エンジンで MPC を実行すると仮定すると、単精度浮動小数点演算のとき、整数型の乗算プロトコルに  $6 \times |q| = 444$  ビットの総通信量を必要とし、文献 [9] で計測している乗算プロトコルと同程度の入力並列度で、ネットワークトラフィックの帯域幅の上限を上回ることとなる。倍精度浮動小数点演算の場合には変曲点の並列度の値がさらに小さくなり、より小さい入力並列度でネットワークトラフィックの帯域幅の上限を上回るため、通信量削減の効果が大きいといえる。

したがって、文献 [9] での実験と同じ条件下で実行時間を測定する場合には、入力の変曲度がある程度大きい高スケールな計算を実行する場合において、ラウンド数よりも通信量が支配的になる。高スケールな計算の例としては大規模な行列計算やデータベースの並列検索などがある。以上のような条件下では、通信量を削減した提案プロトコルの貢献度が高いといえる。

## 5. 提案する浮動小数点乗算プロトコル

従来の浮動小数点乗算プロトコルでは、はじめに仮数部の積を計算し、Trunc プロトコルで下位  $\ell$  ビットを切り捨てる。 $\ell$  ビットの値と  $\ell$  ビットの値の積を計算するとき、計算結果は最大で  $2\ell$  ビットとなる。このとき、 $2\ell$  ビットの値を入力として Trunc プロトコルを実行すると、法  $q$  のビット長は  $2\ell + \kappa$  以上である必要がある。提案した加算プロトコルは小さい法で実行する手法であるため、それに合わせた小さい法で実行可能な乗算プロトコルについても提案する。 $\ell$  が偶数の場合のプロトコルをプロトコル 5 に示し、奇数の場合のプロトコルをプロトコル 6 に示す。

### 5.1 手順

プロトコル 5 を例にして、提案プロトコルの実行手順の概略について説明する。

(1) ステップ 1 からステップ 4 では、入力の浮動小数の仮数部  $[v_1]$ ,  $[v_2]$  を上位ビットと下位ビットに分割する。 $[v_1]$ ,  $[v_2]$  に Trunc プロトコルを実行することで、 $[v_1]$ ,  $[v_2]$  から上位  $\frac{\ell}{2}$  ビットを抽出することができる。上位ビットをそれぞれ  $[v_{1T}]$ ,  $[v_{2T}]$  とし、 $[v_{1T}]$ ,  $[v_{2T}]$  に  $2^{\frac{\ell}{2}}$  をかけ合わせ、 $[v_1]$ ,  $[v_2]$  から引くことで、下位ビットを抽出することができる。下位ビットを  $[v_{1B}]$ ,  $[v_{2B}]$

### プロトコル 5 Proposed FLMul ( $\ell$ が偶数)

---

**Input:**  $\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle$

- 1:  $[v_{1T}] \leftarrow \text{Trunc}([v_1], \ell, \frac{\ell}{2});$
- 2:  $[v_{2T}] \leftarrow \text{Trunc}([v_2], \ell, \frac{\ell}{2});$
- 3:  $[v_{1B}] \leftarrow [v_1] - [v_{1T}]2^{\frac{\ell}{2}};$
- 4:  $[v_{2B}] \leftarrow [v_2] - [v_{2T}]2^{\frac{\ell}{2}};$
- 5:  $[w_1] \leftarrow [v_{1B}][v_{2B}];$
- 6:  $[w_2] \leftarrow [v_{1B}][v_{2T}];$
- 7:  $[w_3] \leftarrow [v_{1T}][v_{2B}];$
- 8:  $[w_4] \leftarrow [v_{1T}][v_{2T}];$
- 9:  $[w_1] \leftarrow \text{Trunc}([w_1], \ell, \frac{\ell}{2});$
- 10:  $[w'] \leftarrow [w_1] + [w_2] + [w_3];$
- 11:  $[w'] \leftarrow \text{Trunc}([w'], \ell + 2, \frac{\ell}{2});$
- 12:  $[w''] \leftarrow [w'] + [w_4];$
- 13:  $[a] \leftarrow \text{LT}([w''], 2^\ell, \ell + 1);$
- 14:  $[v] \leftarrow \text{Trunc}(2[a][w''] + (1 - [a])[w''], \ell + 1, 1);$
- 15:  $[z] \leftarrow \text{OR}([z_1], [z_2]);$
- 16:  $[s] \leftarrow \text{XOR}([s_1], [s_2]);$
- 17:  $[p] \leftarrow ([p_1] + [p_2] + \ell - [a])(1 - [z]);$
- 18: **return**  $\langle [v], [p], [z], [s] \rangle$

---

### プロトコル 6 Proposed FLMul ( $\ell$ が奇数)

---

**Input:**  $\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle$

- 1:  $[v_{1T}] \leftarrow \text{Trunc}([v_1], \ell, \frac{\ell-1}{2});$
- 2:  $[v_{2T}] \leftarrow \text{Trunc}([v_2], \ell, \frac{\ell-1}{2});$
- 3:  $[v_{1B}] \leftarrow [v_1] - [v_{1T}]2^{\frac{\ell-1}{2}};$
- 4:  $[v_{2B}] \leftarrow [v_2] - [v_{2T}]2^{\frac{\ell-1}{2}};$
- 5:  $[w_1] \leftarrow [v_{1B}][v_{2B}];$
- 6:  $[w_2] \leftarrow [v_{1B}][v_{2T}];$
- 7:  $[w_3] \leftarrow [v_{1T}][v_{2B}];$
- 8:  $[w_4] \leftarrow [v_{1T}][v_{2T}];$
- 9:  $[w_1] \leftarrow \text{Trunc}([w_1], \ell - 1, \frac{\ell-1}{2});$
- 10:  $[w'] \leftarrow [w_1] + [w_2] + [w_3];$
- 11:  $[w'] \leftarrow \text{Trunc}([w'], \ell + 2, \frac{\ell-1}{2});$
- 12:  $[w''] \leftarrow [w'] + [w_4];$
- 13:  $[a] \leftarrow \text{LT}([w''], 2^\ell, \ell + 2);$
- 14:  $[v] \leftarrow \text{Trunc}(2[a][w''] + (1 - [a])[w''], \ell + 2, 2);$
- 15:  $[z] \leftarrow \text{OR}([z_1], [z_2]);$
- 16:  $[s] \leftarrow \text{XOR}([s_1], [s_2]);$
- 17:  $[p] \leftarrow ([p_1] + [p_2] + \ell - [a])(1 - [z]);$
- 18: **return**  $\langle [v], [p], [z], [s] \rangle$

---

とする。

- (2) ステップ 5 からステップ 8 では、抽出した上位ビット下位ビット 4 つをそれぞれかけ合わせ、 $[w_1]$ ,  $[w_2]$ ,  $[w_3]$ ,  $[w_4]$  とする。これらを  $\frac{\ell}{2}$  ビットずつ桁をずらし足し合わせることで、 $[v_1][v_2]$  の計算結果を得ることができる。次に、桁をずらして加算を実行する。 $[w_1]$  の下位  $\frac{\ell}{2}$  ビットを切り捨てて、 $[w_2]$ ,  $[w_3]$  と足し合わせる。この加算結果を  $[w']$  とすると、 $[w']$  は最大で  $\ell + 2$  ビットになっているため、下位  $\frac{\ell}{2}$  ビットを再び切り捨てる。
- (3) ステップ 9 からステップ 12 では、 $[w_4]$  と足し合わせ、この加算結果  $[w'']$  が仮数部の乗算結果の上位ビットとなっている。次に、 $[w'']$  を  $\ell$  ビットに調節する。
- (4) ステップ 13 からステップ 14 では、 $[w']$  と  $[w_4]$  の加算

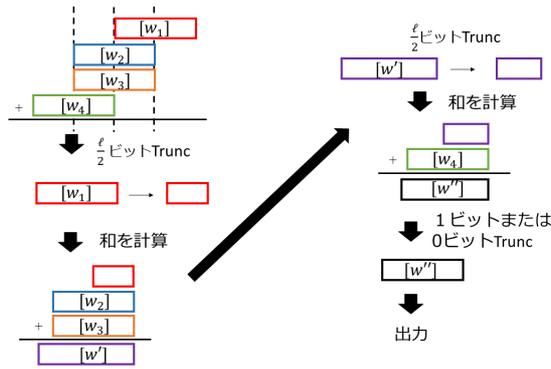


図 1 浮動小数点乗算プロトコルにおける提案手法の実行手順  
 Fig. 1 Procedure of the proposed floating-point multiplication protocol.

結果は  $\ell$  ビットまたは  $\ell + 1$  ビットになっているので、 $\ell + 1$  ビットに揃えてから  $\ell$  ビットへと調節する。

(5) 最後に、ステップ 15 からステップ 17 では、指数部、符号ビット、ゼロフラグの設定をする。

$\ell$  が奇数の場合には上記手順と同様だが、上位ビットと下位ビットを  $\frac{\ell-1}{2}$  ビットと  $\frac{\ell+1}{2}$  ビットに分割する。乗算プロトコルの実行手順の概略図を図 1 に示す。

### 5.2 処理コスト削減の基本アイデア

小さい法で乗算プロトコルを実行するために、多倍長乗算のアルゴリズムにおいて計算量を削減することができる Karatsuba 法に似た手法を取り入れる。基本的なアイデアは仮数部を上位ビットと下位ビットに分割してから積を計算することである。 $[w_1], [w_2], [w_3], [w_4]$  は仮数部ビットを分割してかけ合わせたものなので、計算結果の桁数は最大でも  $\ell$  ビットとなる。最終的な乗算結果を求めるためには、 $[w_1], [w_2], [w_3], [w_4]$  の桁をずらして加算を実行すればよいので、加算時に逐一下位ビットを切り捨てることで計算結果の桁数は最大でも  $\ell + 2$  ビットにしかならない。したがって、 $2\ell$  ビットの値に対して Trunc プロトコルを実行することを回避できるため、小さい法で実行可能である。

### 5.3 処理コスト

従来プロトコルと提案プロトコルであるプロトコル 5 およびプロトコル 6 の処理コストの比較を表 11 に示す。乱数共有を事前計算した際の処理コストについては、表 12 に示す。また、浮動小数点加算の提案プロトコルと同一のビット長および同一の法で実行することを想定し具体値での比較をする。単精度の場合には  $k = 8, \kappa = 40$ 、倍精度の場合には  $k = 11, \kappa = 40$  とし、 $\ell$  が偶数のときには  $\ell = 32, 64$ 、 $\ell$  が奇数のときには  $\ell = 31, 63$  として、通信量の具体値および通信量 (ビット) を求め、表 13 に示す。乱数共有を事前計算した際の処理コストの具体値については、表 14 に示す。ただし、従来プロトコルは法  $|q| = 2\ell + \kappa + 1$

表 11 浮動小数点乗算プロトコルにおける処理コスト比較

Table 11 Complexity comparison of floating-point multiplication protocol.

	通信量	ラウンド
従来	$8\ell + 10$	11
提案 ( $\ell$ : 偶数)	$12\ell + 15$	23
提案 ( $\ell$ : 奇数)	$12\ell + 20$	23

表 12 浮動小数点乗算プロトコルにおける処理コスト比較 (事前計算あり)

Table 12 Complexity comparison floating-point multiplication protocol after precomputation.

	通信量	ラウンド
従来	$2\ell + 11$	12
提案 ( $\ell$ : 偶数)	$3\ell + 20$	18
提案 ( $\ell$ : 奇数)	$3\ell + 22$	18

表 13 浮動小数点乗算プロトコルにおける処理コストの具体値の比較

Table 13 Examples of complexity comparison of floating-point multiplication protocol in single and double precision.

	通信量 (乗算)	通信量 (ビット)
$\ell = 32$ 従来	266	27,930
提案	399	29,127
$\ell = 31$ 従来	258	27,090
提案	392	28,616
$\ell = 64$ 従来	522	88,218
提案	783	82,998
$\ell = 63$ 従来	514	85,838
提案	776	81,480

表 14 浮動小数点乗算プロトコルにおける処理コストの具体値の比較 (事前計算あり)

Table 14 Examples of complexity comparison of floating-point multiplication protocol in single and double precision after precomputation.

	通信量 (乗算)	通信量 (ビット)
$\ell = 32$ 従来	75	7,875
提案	116	8,468
$\ell = 31$ 従来	73	7,519
提案	115	8,280
$\ell = 64$ 従来	139	23,491
提案	212	22,472
$\ell = 63$ 従来	137	22,879
提案	211	22,155

上で実行し、提案プロトコルは法  $|q'| = \ell + \kappa + 1$  上で実行するものとする。

処理コストを具体値で算出したとき、単精度では  $\ell$  の偶奇によらず、また事前計算の有無によらず通信量 (ビット) が 5~10%程度大きくなる。倍精度の場合、 $\ell$  が偶数のときに通信量が約 5.92%減少し、 $\ell$  が奇数のときには約

5.08%減少することが分かる．また，乱数共有を事前計算した場合， $\ell$ が偶数のときに通信量が約4.35%減少し， $\ell$ が奇数のときには約3.16%減少することが分かる．

従来プロトコルと比較して提案プロトコルは，ほぼ同程度の通信量で実行可能であるといえる．ただし，Truncプロトコルを複数回実行することによってラウンド数が増加するというトレードオフが存在する．

#### 5.4 安全性

浮動小数点乗算プロトコルにおいても，浮動小数点加算プロトコルと同様にデータの秘匿に秘密分散法を用いており，値の計算にはMPCを用いている．プロトコル中で値の公開を行っていないので，プロトコル全体の安全性は線形秘密分散とそれに基づくMPCの安全性に帰着する．

ただし，プロトコル内部でLT, Truncプロトコルを実行しているので，提案プロトコル全体は統計的安全となる．提案プロトコルはセキュリティパラメータ $\kappa$ により統計的安全が確保されるため，従来プロトコルと同じ水準の安全性で実行することができる．

#### 5.5 誤差の評価

従来プロトコルと提案プロトコルの実行における誤差について言及する．従来プロトコルと提案プロトコルとの間で誤差が発生するのは，基本的に提案プロトコルにおいて計算に影響する箇所を切り捨てる時である．

従来プロトコルでは， $\ell$ ビットの仮数部をかけ合わせて，一度 $2\ell$ ビットにしてから下位の $\ell$ ビットを切り捨てる．一方で，提案プロトコルでは最初に $[w_1]$ の下位 $\frac{\ell}{2}$ ビットまたは $\frac{\ell-1}{2}$ ビットを切り捨てる． $[w_1]$ の下位ビットを切り捨てないで $[w_2]$ ,  $[w_3]$ と加算をしても，下位ビットではキャリーが発生しない．したがって， $[w_2]$ ,  $[w_3]$ との加算前に $[w_1]$ の下位ビットを切り捨てても上位ビットには影響を与えない．残りの $[w_4]$ との加算においては，加算後に下位ビットを切り捨てるので切り捨てる箇所が上位ビットに影響を与えない．

以上より，従来プロトコルと提案プロトコルとの間で誤差は発生しない．

### 6. 提案プロトコルの有効性

本章では特定の計算に適用した際の処理コストを算出し，従来手法と提案手法とで比べることで，提案のプロトコルの有効性について言及する． $n$ 次元ベクトルの内積を例として，浮動小数点演算での処理コストを従来手法と提案手法とで比較する．このとき，実行する演算は加算が $n-1$ 回で乗算が $n$ 回となる．倍精度の演算を想定して $\ell = 64$ ,  $k = 11$ ,  $\kappa = 40$ ,  $|q| = 169$ ,  $|q'| = 106$ とし，事前計算を実行するものとする．ケース1およびケース2での処理コストをそれぞれ表15, 表16に示す．

表15  $n$ 次元ベクトルの内積計算における処理コスト比較 (ケース1)

Table 15 Case 1: Complexity comparison of inner product of  $n$ -dimensional vectors after precomputation.

	通信量 (乗算)	通信量 (ビット)	ラウンド
従来	$1,047n - 908$	$176,943n - 153,452$	$48 \times \lceil \log n \rceil$
提案	$765n - 553$	$81,090n - 58,618$	$55 \times \lceil \log n \rceil$

表16  $n$ 次元ベクトルの内積計算における処理コスト比較 (ケース2)

Table 16 Case 2: Complexity comparison of inner product of  $n$ -dimensional vectors after precomputation.

	通信量 (乗算)	通信量 (ビット)	ラウンド
従来	$787n - 648$	$13,3003n - 109,512$	$50 \times \lceil \log n \rceil$
提案	$892n - 680$	$94,552n - 72,080$	$60 \times \lceil \log n \rceil$

具体的な値で算出すると， $n = 10$ のとき，ケース1においては従来手法の通信量(ビット)が1,615,978，提案手法の通信量(ビット)が752,282となり，提案手法のプロトコルを用いることで約53.4%の処理コスト削減となる．ケース2においては従来手法の通信量(ビット)が1,220,518，提案手法の通信量(ビット)が873,440となり，提案手法のプロトコルを用いることで約28.4%の処理コスト削減となる．どちらのケースにおいても，提案手法の加算と乗算のプロトコルを用いることが処理コストの削減になることが確認できる．

### 7. 結論

浮動小数点演算について，仮数部を右シフトすることで一時的なビット長の増加を抑制し，小さい法の上で実行可能な手法を提案した．これにより，法の大きさを考慮した通信量の指標において，処理コストを削減できる．

浮動小数点加算については，プロトコル内部で用いるBitDecプロトコルの種類によって2通りの効率化手法を提案した．BitDecプロトコルとして文献[12]で述べられている手法を採用する場合，効率化手法としてBitDecプロトコルの実行を回避する手法を提案した．これにより，通信量のオーダーを $\mathcal{O}(\ell \log \ell)$ から $\mathcal{O}(\ell)$ へと削減し，ラウンドのオーダーを $\mathcal{O}(\log \ell)$ から $\mathcal{O}(\log \log \ell)$ へと削減できる．特に， $\ell = 64$ ,  $k = 11$ ,  $\kappa = 40$ の具体的な値での実行を想定した場合，通信量を約32.4%削減でき，乱数共有を事前計算した際には約61.8%削減できる．このとき実行可能な演算が限定されるが，特定のアプリケーションに対して提案プロトコルを用いることを前提とし，適用する計算を工夫することで提案プロトコルが有用であることを示した．一方，BitDecプロトコルとして文献[10]で述べられている手法を採用することを前提とする場合，BitDecプロトコルの実行を回避する必要はないが，小さい法で実行可能なプロトコルを提案した．この場合，通信量のオーダーに変化はないがラウンドのオーダーを $\mathcal{O}(\log \ell)$ から $\mathcal{O}(1)$ に削減することができる． $\ell = 64$ ,  $k = 11$ ,  $\kappa = 40$ の具体的な値で

の実行を想定した場合、通信量を約 32.9%削減でき、乱数共有を事前計算した際には約 34.2%削減でき、かつ前述のケースとは異なり実行可能な演算に制限がない。

浮動小数点乗算プロトコルについては、浮動小数点加算の提案プロトコルで設定する小さい法の上で実行可能な方式を提案した。通信量は従来手法とほぼ同じであるが、浮動小数の加算と乗算を組み合わせた計算を実行する際には、有効であることを示した。

今後の展望については、処理コストの正確な算出の側面から実装して実測値での処理コストを算出することがあげられる。また、応用の側面からは加算と乗算の提案プロトコルを組み合わせ、FLLT プロトコルや FLEQ プロトコルといった上位プロトコルを構成することがあげられる。これらの上位プロトコルはすでに提案されているが、本論文で提案したプロトコルを用いて構成しなおすことで、処理コストの削減が期待できる。

#### 参考文献

- [1] MP-SPDZ, available from (<https://github.com/nlanalytics/MP-SPDZ>).
- [2] SCALE-MAMBA, available from (<https://github.com/KULeuven-COSIC/SCALE-MAMBA>).
- [3] SPDZ-2, available from (<https://github.com/bristolcrypto/SPDZ-2>).
- [4] Aliasgari, M., Blanton, M. and Bayatbabolghani, F.: Secure computation of hidden Markov models and secure floating-point arithmetic in the malicious model, *Int. J. Inf. Sec.*, Vol.16, No.6, pp.577–601 (2017).
- [5] Aliasgari, M., Blanton, M., Zhang, Y. and Steele, A.: Secure Computation on Floating Point Numbers, *20th Annual Network and Distributed System Security Symposium, NDSS 2013*, The Internet Society (2013).
- [6] Araki, T., Barak, A., Furukawa, J., Lichter, T., Lindell, Y., Nof, A., Ohara, K., Watzman, A. and Weinstein, O.: Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier, *2017 IEEE Symposium on Security and Privacy, SP 2017*, pp.843–862, IEEE Computer Society (2017).
- [7] Araki, T., Furukawa, J., Lindell, Y., Nof, A. and Ohara, K.: High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority, *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security*, Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C. and Halevi, S. (Eds.), pp.805–817, ACM (2016).
- [8] Ben-Or, M., Goldwasser, S. and Wigderson, A.: Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract), *Proc. 20th Annual ACM Symposium on Theory of Computing*, Simon, J. (Ed.), pp.1–10, ACM (1988).
- [9] Bogdanov, D., Niitsoo, M., Toft, T. and Willemson, J.: High-performance secure multi-party computation for data mining applications, *Int. J. Inf. Sec.*, Vol.11, No.6, pp.403–418 (2012).
- [10] Catrina, O.: Round-Efficient Protocols for Secure Multiparty Fixed-Point Arithmetic, *2018 International Conference on Communications (COMM)*, pp.431–436 (2018).
- [11] Catrina, O. and de Hoogh, S.: Improved Primitives for Secure Multiparty Integer Computation, *Proc. Security and Cryptography for Networks, 7th International Conference, SCN 2010*, 2010, Garay, J.A., Prisco, R.D. (Eds.), Lecture Notes in Computer Science, Vol.6280, pp.182–199, Springer (2010).
- [12] Catrina, O. and Saxena, A.: Secure Computation with Fixed-Point Numbers, *Financial Cryptography and Data Security, 14th International Conference, FC 2010*, Revised Selected Papers, Sion, R. (Ed.), Lecture Notes in Computer Science, Vol.6052, pp.35–50, Springer (2010).
- [13] Cramer, R., Damgård, I. and Nielsen, J.B.: *Secure Multiparty Computation and Secret Sharing*, Cambridge University Press (2015).
- [14] Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B. and Toft, T.: Unconditionally Secure Constant-Rounds Multiparty Computation for Equality, Comparison, Bits and Exponentiation, *Proc. Theory of Cryptography, 3rd Theory of Cryptography Conference, TCC 2006*, Halevi, S. and Rabin, T. (Eds.), Lecture Notes in Computer Science, Vol.3876, pp.285–304, Springer (2006).
- [15] Franz, M., Deiseroth, B., Hamacher, K., Jha, S., Katzenbeisser, S. and Schröder, H.: Secure computations on non-integer values, *2010 IEEE International Workshop on Information Forensics and Security, WIFS 2010*, pp.1–6, IEEE (2010).
- [16] Geisler, M., Toft, T., Krøigård, M., Jakobsen, T.P., Pagter, J.I., Meldgaard, S., Keller, M., Reistad, T., Damgård, I. and Nielsen, J.D.: VIFF software, available from (<http://hg.viff.dk/viff/>).
- [17] Kamm, L. and Willemson, J.: Secure floating point arithmetic and private satellite collision analysis, *Int. J. Inf. Sec.*, Vol.14, No.6, pp.531–548 (2015).
- [18] Kerik, L., Laud, P. and Randmets, J.: Optimizing MPC for Robust and Scalable Integer and Floating-Point Arithmetic, *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Revised Selected Papers*, Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D.S., Brenner, M. and Rohloff, K. (Eds.), Lecture Notes in Computer Science, Vol.9604, pp.271–287, Springer (2016).
- [19] Krips, T. and Willemson, J.: Hybrid Model of Fixed and Floating Point Numbers in Secure Multiparty Computations, *Proc. Information Security - 17th International Conference, ISC 2014*, Chow, S.S.M., Camenisch, J., Hui, L.C.K. and Yiu, S. (Eds.), Lecture Notes in Computer Science, Vol.8783, pp.179–197, Springer (2014).
- [20] Lee, D.D. and Seung, H.S.: Algorithms for Non-negative Matrix Factorization, *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS 2000)*, Leen, T.K., Dietterich, T.G. and Tresp, V. (Eds.), pp.556–562, MIT Press (2000).
- [21] Lindell, Y.: Secure Computation for Privacy Preserving Data Mining, *Encyclopedia of Data Warehousing and Mining, 2nd Edition (4 Volumes)*, Wang, J. (Ed.), pp.1747–1752, IGI Global (2009).
- [22] Liu, Y., Chiang, Y., Hsu, T., Liau, C. and Wang, D.: Floating Point Arithmetic Protocols for Constructing Secure Data Analysis Application, *17th International Conference in Knowledge Based and Intelligent Information and Engineering Systems, KES 2013*, Watada, J., Jain, L.C., Howlett, R.J., Mukai, N. and Asakura, K. (Eds.), *Procedia Computer Science*, Vol.22, pp.152–161, Elsevier (2013).
- [23] Nishide, T. and Ohta, K.: Multiparty Computation

for Interval, Equality, and Comparison Without Bit-Decomposition Protocol, *Proc. Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography*, Okamoto, T. and Wang, X. (Eds.), Lecture Notes in Computer Science, Vol.4450, pp.343–360, Springer (2007).

- [24] Shamir, A.: How to Share a Secret, *Comm. ACM*, Vol.22, No.11, pp.612–613 (1979).
- [25] 尾形わかは, 黒沢 馨: 秘密分散共有法とその応用, 電子情報通信学会誌, Vol.82, No.12, pp.1228–1236 (1999).
- [26] 岡本栄司, 西出隆志: 暗号と情報セキュリティ, 8, コロナ社 (2016).
- [27] 黒沢 馨, 尾形わかは: 現代暗号の基礎数理, D-8, コロナ社 (2004).
- [28] 澤田 宏: 非負値行列因子分解 NMF の基礎とデータ/信号解析への応用, 電子情報通信学会誌 = The journal of the Institute of Electronics, Information and Communication Engineers, Vol.95, No.9, pp.829–833 (2012).
- [29] 五十嵐大, 濱田浩気, 菊池 亮, 千田浩司: 少パーティの秘密分散ベース秘密計算のための  $O(1)$  ビット通信ビット分解および  $O(|p'|)$  ビット通信 Modulus 変換法, コンピュータセキュリティシンポジウム 2013 論文集, Vol.2013, No.4, pp.785–792 (2013).



吉浦 裕 (正会員)

1981年東京大学理学部情報科学科卒業。日立製作所を経て、2003年より電気通信大学勤務。現在、情報理工学研究所教授。情報セキュリティ、プライバシー保護の研究に従事。博士(理学)。日立製作所社長技術賞(2000年)、情報処理学会論文賞(2005年、2011年)、システム制御情報学会産業技術賞(2005年)、IEEE IHH-MSP best paper award(2006)、日本セキュリティ・マネジメント学会論文賞(2010年、2016年、2017年)、IFIP I3E best paper award(2016)等受賞。電子情報通信学会、日本セキュリティ・マネジメント学会、人工知能学会、システム制御情報学会、IEEE各会員。本会フェロー。



天田 拓磨

2016年電気通信大学情報理工学部総合情報学科卒業。2018年同大学大学院情報理工学研究科情報学専攻博士前期課程修了。現在、日本電気株式会社勤務。



奈良 成泰 (正会員)

2016年電気通信大学情報理工学部総合情報学科卒業。2018年同大学大学院情報理工学研究科情報学専攻博士前期課程修了。現在、日本電気株式会社勤務。



西出 隆志 (正会員)

1997年東京大学理学部情報科学科卒業。同年日立ソフト株式会社入社。セキュリティ、ネットワーク製品の設計開発に従事。2003年南カリフォルニア大学大学院修士課程修了。2008年電気通信大学博士(工学)。2009年九州大学大学院システム情報科学研究科助教を経て、2013年より筑波大学システム情報系准教授。暗号、情報セキュリティの研究に従事。