情報処理学会論文誌　コンピューティングシステム　Vol.14　No.1　1–13 (Feb. 2021)

小さい定数個の単精度行列への分割を用いた尾崎スキームによる倍精度行列乗算のゲーミング用GPU上での評価

七井 香樹[1]　藤本 典幸[1,a)]

受付日 2020年7月22日, 採録日 2020年11月11日

概要：安価なゲーミング用 GPU による倍精度浮動小数点数の演算は，単精度浮動小数点数の演算よりも多くの時間がかかる．また，高精度な行列積計算法として，行列の各要素を上位ビットと下位ビットに分割し行列積計算を行う尾崎スキームが知られている．さらに椋木らは，尾崎スキームを応用して，最近の GPU が備えている Tensor コアが高速に実行可能な単精度と半精度の混合精度の行列乗算を用いて倍精度相当の行列積を計算する手法を提案している．本論文では尾崎スキームを応用して，安価な GPU でも高速に実行可能な単精度行列乗算を用いて単精度と倍精度の中間の精度の行列積を計算する手法を提案する．提案手法を CUDA と cuBLAS を用いて実装し，NVIDIA GeForce RTX 2080 SUPER を用いて評価したところ，単精度以上倍精度未満の行列積が倍精度行列積計算を実行するよりも高速に得られることが分かった．

キーワード：GPU，CUDA，行列積，最大相対誤差

Evaluation of Double-precision Matrix Multiplication based on the Ozaki Scheme with a Small Constant Number of Split Matrices in Single-precison for Gaming GPUs

Kouki Nanai[1]　Noriyuki Fujimoto[1,a)]

Received: July 22, 2020, Accepted: November 11, 2020

Abstract: On inexpensive GPUs for video games, double precision computation takes much longer time than single precision computation. On the other hand, as a more accurate matrix product computation method, Ozaki et al. proposed the Ozaki scheme to compute a matrix product by partitioning each of given matrices into a matrix of upper bits and a matrix of lower bits. Moreover, Mukunoki et al. applied the Ozaki scheme to compute double-precision matrix product with the mixed-precision (i.e., half and single) matrix multiplication, which can be computed fast using the Tensor cores on recent GPUs. In this paper, we propose a method to compute a matrix product in intermediate-precision between single-precision and double-precision using matrix multiplication in single precision, which can be computed fast also on inexpensive GPUs. We also present our implementation of the proposed method on CUDA and cuBLAS. Experimental results on NVIDIA GeForce RTX 2080 SUPER show that the proposed method yields a matrix product in more accurate precision than single precision and in shorter time than the computing time in double precision.

Keywords: GPU, CUDA, matrix product, maximum relative error

1. はじめに

価格性能比の良さから GPU 計算がここ数年様々な研究分野で活用されている．しかし倍精度演算性能に関してはデータセンターなどを対象とした高性能計算用の高価

[1] 大阪府立大学
Osaka Prefecture University, Sakai, Osaka 599–8531, Japan
[a)] fujimoto@cs.osakafu-u.ac.jp

な GPU だけが高性能である．たとえば Tesla K40 [1] では，単精度演算性能は 5.04 TFLOPS で，倍精度演算性能は 1.68 TFLOPS であり，倍精度演算性能は単精度の 1/3 である．Tesla P100 [1] では，単精度演算性能は 10.6 TFLOPS で，倍精度演算性能は 5.3 TFLOPS であり，Tesla V100 [1], [2] では，単精度演算性能は 15.7 TFLOPS で，倍精度演算性能は 7.8 TFLOPS であり，倍精度演算性能は単精度の 1/2 である．これに対してゲーミング用の安価な GPU は倍精度演算性能を大きく制限されている．たとえば GeForce 20 シリーズなどでは，倍精度演算性能は単精度の 1/32 である [3]．このためゲーミング用の安価な GPU は，コア数とクロック周波数が同じ場合，単精度演算性能は高価な GPU に匹敵するが，倍精度演算性能は高価な GPU の 2/32 または 3/32 しかない[*1]．

一方，尾崎らは数値線形代数計算において多用される行列積計算を高精度に行う手法 [4], [5], [6] を提案している．この手法は BLAS [7] などの高速な行列積ルーチンをサブルーチンとして用いて，倍精度行列積や単精度行列積の演算結果をより高精度化するものであり，その実装は BLAS の gemm ルーチンを用いて容易に高速化できる．NVIDIA 製の GPU には，ベンダーが BLAS を元に GPU に最適化した cuBLAS [8] という行列計算ライブラリが用意されている．さらに椋木らは，尾崎らの手法を応用して，最近の GPU が備えている Tensor コアが高速実行可能な単精度と半精度の混合精度の行列積を用いて倍精度相当の行列積を計算する手法を提案している [9]．尾崎らの手法には行列の分割数というパラメータがあり，分割数を増やすほど高精度になることが期待できるが，椋木らの手法は，確率的誤差限界の観点から分割数をコントロールし，倍精度相当の演算結果を得る．

本論文では，倍精度までの精度は不要だが単精度では精度が足りない問題を対象として，尾崎らの手法を応用して，単精度行列積を用いて倍精度行列積の計算を分割数を小さい定数に固定して行う手法を提案する．さらに NVIDIA 製 GPU 上に cuBLAS と CUDA [10], [11] を用いて提案手法を実装し，分割数と行列積の計算時間および精度の関係の評価を行う．分割数をあらかじめ固定して尾崎スキームを用いる提案手法では，絶対値が非常に大きな，あるいは小さな値が含まれた行列において，通常の単精度行列積よりも精度が低下する恐れがある．このため，入力行列の要素の絶対値の範囲を変えて評価実験を行う．尾崎らの手法は，高精度な倍精度行列積を求める際には倍精度の浮動小数点演算を用いるが，提案手法は倍精度行列積の計算を主に単精度の浮動小数点演算を用いて行う．提案手法を用いれば，倍精度より精度が落ちるが単精度よりは精度が良い行列積の計算を安価なゲーミング用 GPU を用いて高速に行うことが可能となる．

以降の本論文の構成は次のとおりである．まず 2 章で提案手法の理解に必要な基礎知識について述べた後，3 章で尾崎らの手法について解説する．次に 4 章で提案手法を示し，5 章で提案手法の評価を行う．最後に 6 章でまとめと今後の課題について述べる．なお，本論文では CUDA と cuBLAS のプログラミングについては解説しない．これらに不慣れな読者は文献 [8], [10], [11] を参照されたい．

2. 準備

2.1 単精度と倍精度の浮動小数点数の IEEE 754 形式

単精度浮動小数点数は最上位ビットから符号部 1 ビット，指数部 8 ビット，仮数部 23 ビットの合計 32 ビットで構成され，倍精度浮動小数点数は最上位ビットから符号部 1 ビット，指数部 11 ビット，仮数部 52 ビットの合計 64 ビットで構成される．仮数部は正規化されるため，単精度は最大で 24 ビット，倍精度は最大で 53 ビットの有効桁数を持つ．

2.2 アトミック関数

アトミック関数 [12] は，GPU のグローバルメモリやシェアードメモリにある 32 ビットまたは 64 ビットワード長の read-modify-write の操作を他のスレッドからの干渉を防いで実行する関数である．CUDA には 11 種類のアトミック関数が用意されており，そのなかに最大値を求める atomicMax という関数がある．

関数 atomicMax(addr, val) は，ポインタ addr が指すグローバルメモリまたはシェアードメモリから 32 ビットまたは 64 ビット整数 old を読み込み，max(old, val) を *addr に書き込んだ後，old を返す．

3. 尾崎らの高精度な行列積計算法（尾崎スキーム）

高精度な行列積計算法として，行列を各要素の上位ビットの行列と下位ビットの行列に分割してから行列積を計算する尾崎らの手法 [4], [5], [6] が知られている．

上位ビットの行列を $A^{(1)}$，$B^{(1)}$ とし，下位ビットの行列を $A^{(2)}$，$B^{(2)}$ とすると，求める行列積 $C = AB$ は以下のように 3 つの行列積の和で表される．

$$A = A^{(1)} + \underline{A^{(2)}} \tag{1}$$

$$B = B^{(1)} + \underline{B^{(2)}} \tag{2}$$

$$C = A^{(1)}B^{(1)} + A^{(1)}\underline{B^{(2)}} + \underline{A^{(2)}}B \tag{3}$$

このとき，$A^{(1)}B^{(1)}$ の行列積計算は誤差なしで計算できるように行列 A，B を分割する．

$A^{(1)}B^{(1)}$ が誤差なしで計算できるための条件は次のとお

[*1] 単精度演算性能を F TFLOPS とすると，ゲーミング用 GPU の倍精度演算性能は $F/32$ TFLOPS，高性能計算用 GPU の倍精度演算性能は $F/3$ または $F/2$ TFLOPS となる．

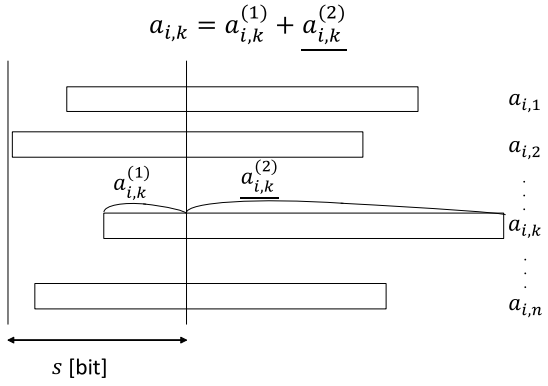

図 1　行列 A の要素の分割

Fig. 1　Partitioning elements of matrix A.

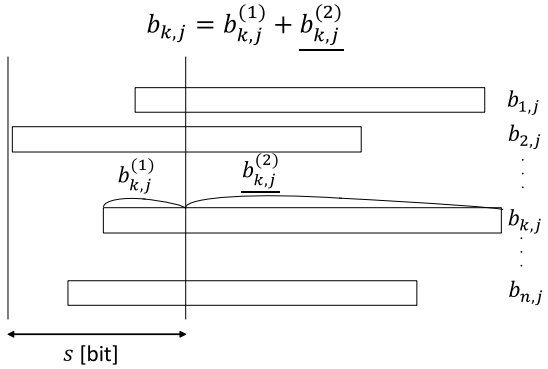

図 2　行列 B の要素の分割

Fig. 2　Partitioning elements of matrix B.

りである．行列 A のサイズを $m \times n$，行列 B のサイズを $n \times p$，単位相対丸め（unit roundoff）を u（単精度の場合は $u = 2^{-24}$，倍精度の場合は $u = 2^{-53}$），仮数部の有効桁数を $-\log_2 u$ ビット，上位ビット行列 $A^{(1)}$，$B^{(1)}$ の要素を $a_{i,k}^{(1)}$，$b_{k,j}^{(1)}$ とし，行列積 $C^{(1)} = A^{(1)}B^{(1)}$ の要素を $c_{i,j}^{(1)}$ とすると，

$$c_{i,j}^{(1)} = \sum_{k=1}^{n} a_{i,k}^{(1)} b_{k,j}^{(1)} \tag{4}$$

なので，**図 1**（**図 2**）のように，A の各行（B の各列）の要素をその行（列）の最大値の上位 α ビットの位で分割し，内積計算で**図 3** のように，n 個の浮動小数点数を足し合わせた値が仮数部の桁数である $-\log_2 u$ ビット以下となればよい．n 個の浮動小数点数を足したときの繰り上がりの桁数の最大値は $\lceil \log_2 n \rceil$ であり，また，$A^{(1)}$ と $B^{(1)}$ の各要素の仮数部の（暗黙の 1 を含めて）$\alpha + 1$ ビット目以降がすべて 0 とすると，$a_{i,k}^{(1)} b_{k,j}^{(1)}$ の仮数部の先頭からの非ゼロビット数は最大 2α ビットになるので，以下の式 (5) より，α を式 (6) のように選ぶ．

$$-\log_2 u \geq 2 \times \alpha + \lceil \log_2 n \rceil \tag{5}$$

$$\alpha = \left\lfloor \frac{-\log_2 u - \lceil \log_2 n \rceil}{2} \right\rfloor \tag{6}$$

すなわち，式 (6) を満たすように行列の各要素を上位ビットと残りの下位ビットに分割すればよい．

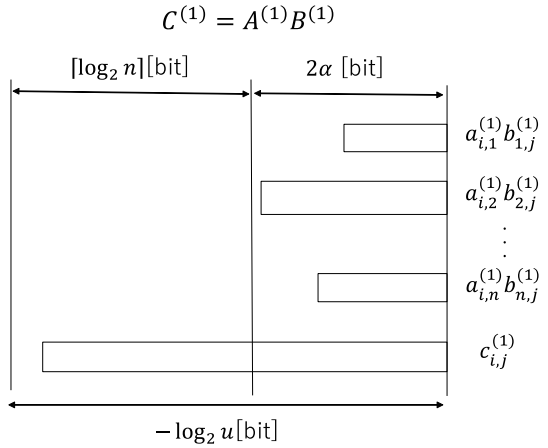

図 3　分割した要素の足し合わせ

Fig. 3　Summation of partitioned elements.

次に，A，B を上位ビットの行列と残りの下位ビットの行列に分割するための手法を示す．行列積を求めるとき，行列 A は各行に注目して計算する．よって，S_A という各行の要素が行列 A の各行の要素に比べて大きな絶対値を持つ行列を用意し，

$$A^{(1)} = \mathrm{fl}\left((A + S_A) - S_A\right) \tag{7}$$

$$\underline{A^{(2)}} = \mathrm{fl}(A - A^{(1)}) \tag{8}$$

という方法で A を分割する．ここで，$\mathrm{fl}(\cdot)$ は括弧内のすべての 2 項演算を浮動小数点演算で評価することを意味する．ここで丸め方式は最も近い値への丸め（round to nearest）（中間値は最も近い偶数に丸める）を仮定する．行列 A と行列 S_A を足すことによって仮数部の範囲内で行列 A の下位ビットが情報落ちする．そして，情報落ちした値から行列 S_A を引くことによって桁落ちした行列 $A^{(1)}$ が求められ，行列 $\underline{A^{(2)}}$ は元の行列 A から桁落ちした行列 $A^{(1)}$ を引くことによって求める．

行列 S_A は次のように求められる．行列 A の各行の絶対値の最大値を求め，その値の上位 α ビットまで取得する．行列 A の i 行目の最大値 μ_{a_i} を求め，

$$\sigma_{a_i} = 2^{(\lceil \log_2 \mu_{a_i} \rceil + \beta)} \tag{9}$$

となる仮数部が $-\log_2 u$ ビットの σ_{a_i} を設定する．**図 4** の灰色の部分を μ_{a_i} の仮数部とすると，$2^{\lceil \log_2 \mu_{a_i} \rceil - 1}$ の位から $2^{\lceil \log_2 \mu_{a_i} \rceil - \alpha}$ の位が上位ビットとなる．σ_{a_i} は仮数部が $-\log_2 u$ ビットであるため，β は，

$$\begin{aligned}
\beta &= -\log_2 u - \alpha \\
&= -\log_2 u - \left\lfloor \frac{-\log_2 u - \lceil \log_2 n \rceil}{2} \right\rfloor \\
&= \left\lceil -\log_2 u - \left(\frac{-\log_2 u - \lceil \log_2 n \rceil}{2} \right) \right\rceil \\
&= \left\lceil \frac{-\log_2 u + \lceil \log_2 n \rceil}{2} \right\rceil
\end{aligned}$$

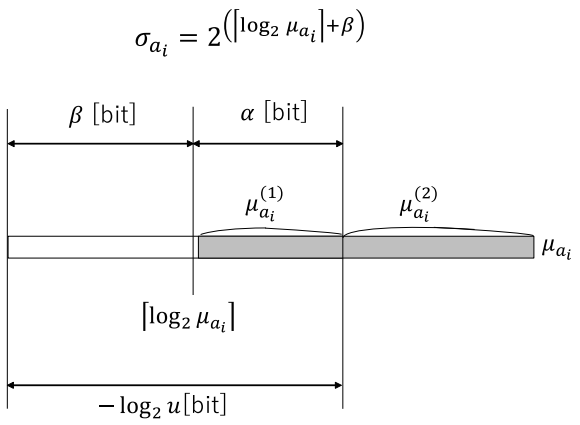

図 4　σ_{a_i} と μ_{a_i} の関係
Fig. 4　The relation between σ_{a_i} and μ_{a_i}.

$$= \left\lceil \frac{-\log_2 u + \log_2 n}{2} \right\rceil \quad (\because -\log_2 u \in \mathbb{N}) \tag{10}$$

と表される．行列 S_A の要素を $s_{a_{i,k}}$ とすると，

$$s_{a_{i,1}} = s_{a_{i,2}} = \cdots = s_{a_{i,n}} = \sigma_{a_i} \tag{11}$$

とし，これを各行で求めたものが行列 S_A である．

また，同様に行列 B を分割する行列 S_B を求められるが，行列積を求めるとき，行列 B は各列に注目して計算するので，各列の要素が行列 B の各列に比べて大きな絶対値を持つ行列 S_B を用意する．行列 B の j 列の最大値 μ_{b_j} を求め，

$$\sigma_{b_j} = 2^{(\lceil \log_2 \mu_{b_j} \rceil + \beta)} \tag{12}$$

となる σ_{b_j} を設定する．図 4 の μ_{a_i} の部分を μ_{b_j} とすれば σ_{b_j} を求められるため，β は同じ値を用いる．行列 S_B の要素を $s_{b_{k,j}}$ とすると，

$$s_{b_{1,j}} = s_{b_{2,j}} = \cdots = s_{b_{n,j}} = \sigma_{b_j} \tag{13}$$

を満たすものが行列 S_B である．

下位ビットに対応する $\underline{A^{(2)}}$，$\underline{B^{(2)}}$ に対してもう 1 度同じアルゴリズムを適用することを考える．このときの上位ビットの行列を $A^{(2)}$，$B^{(2)}$，下位ビットの行列を $\underline{A^{(3)}}$，$\underline{B^{(3)}}$ とすると，

$$\underline{A^{(2)}} = A^{(2)} + \underline{A^{(3)}} \tag{14}$$

$$\underline{B^{(2)}} = B^{(2)} + \underline{B^{(3)}} \tag{15}$$

と表され，元の行列を 3 分割にすることができる．このとき，上位ビット行列の $A^{(2)}$，$B^{(2)}$ は最大 α ビットの要素を持つ行列である．

また，求める行列積 C は，

$$\begin{aligned} C = {} & A^{(1)}B^{(1)} + A^{(1)}B^{(2)} + A^{(2)}B^{(1)} \\ & + A^{(1)}\underline{B^{(3)}} + A^{(2)}\underline{B^{(2)}} + \underline{A^{(3)}}B \end{aligned} \tag{16}$$

と 6 つの行列積の和で表せるので，行列積を計算する回数は多くなるが，$A^{(1)}B^{(1)} + A^{(1)}B^{(2)} + A^{(2)}B^{(1)}$ 部分はそれぞれ最大 α ビットどうしの要素を持つ行列の行列積計算なので誤差なしで計算されるため，より高精度に行列積を計算することが期待される．同様に 4 分割以上への分割も考えられる．

これまでの内容を MATLAB 風の疑似コードで記述したものをアルゴリズム 1 に示す．$\mathbb{F}$ をある固定された精度における正規化数の集合，非正規化数の集合および零の和集合とする ($\mathbb{F} \subsetneq \mathbb{R}$)．アルゴリズム 1 は $A \in \mathbb{F}^{m\times n}, B \in \mathbb{F}^{n\times p}$ を $\mathrm{fl}(A^{(1)}B^{(1)}) = A^{(1)}B^{(1)}$ を満たすように $A = A^{(1)} + \underline{A^{(2)}}$，$B = B^{(1)} + \underline{B^{(2)}}$ と分割する．ただし数値計算中にオーバフロー，アンダフローは起こらないと仮定する．また，A にはすべての要素が 0 の行は含まれず，B にはすべての要素が 0 の列は含まれないと仮定する．

アルゴリズム 1 尾崎らの行列分割アルゴリズム [4], [5], [9]
Input: A, B, u
Output: $A^{(1)}, B^{(1)}, \underline{A^{(2)}}, \underline{B^{(2)}}$
1: $n = \mathrm{size}(A, 2)$;
2: $\mu_A = \max(\mathrm{abs}(A), [\,], 2)$;
　　$\%\mu_A \in \mathbb{F}^{m\times 1}\ \mu_A(i) = \max_{1\le k\le n} |a_{i,k}|$
3: $\mu_B = \max(\mathrm{abs}(B), [\,], 1)$;
　　$\%\mu_B \in \mathbb{F}^{1\times p}\ \mu_B(j) = \max_{1\le k\le n} |b_{k,j}|$
4: $\beta = \left\lceil \frac{-\log_2(u)+\log_2(n)}{2} \right\rceil$;
5: $t_A = 2.\hat{}(\lceil \log_2(\mu_A) \rceil + \beta)$;
6: $t_B = 2.\hat{}(\lceil \log_2(\mu_B) \rceil + \beta)$;
7: $S_A = \mathrm{repmat}(t_A, 1, n)$;
　　$\%S_A = t_A \cdot e^T$ for $e = (1, 1, \cdots, 1)^T \in \mathbb{F}^n$
8: $S_B = \mathrm{repmat}(t_B, n, 1)$;
　　$\%S_B = e \cdot t_B$ for $e = (1, 1, \cdots, 1)^T \in \mathbb{F}^n$
9: $A^{(1)} = (A + S_A) - S_A$;
10: $\underline{A^{(2)}} = A - A^{(1)}$;
11: $B^{(1)} = (B + S_B) - S_B$;
12: $\underline{B^{(2)}} = B - B^{(1)}$;

アルゴリズム 1 は，行列分割過程において各要素に注目して計算をしているので並列処理に適している．

4. 提案手法

尾崎らの手法は高精度計算のため，仮数部のビット数である $-\log_2 u$ の u の値を，倍精度行列積計算を行う場合は $u = 2^{-53}$ を用い，単精度行列積計算を行う場合は $u = 2^{-24}$ を用いる．これに対して，提案手法は倍精度行列積計算であるが，$u = 2^{-24}$ とする．これによって行列積を単精度で行えるので，安価な GPU で高速に計算可能な単精度行列積計算で倍精度行列積の近似計算を実現できる．

$C^{(1)}$ の各要素の仮数部は 24 ビットであるため，図 3 の要素 $c_{i,j}^{(1)}$ は最大 24 ビットであり，これを満たすように α を設定する．行列を 2 分割し行列積を計算するとき，C は式 (3) のように表される．その C の要素を $c_{i,j}$，$A^{(1)}\underline{B^{(2)}}$ の要素を $\underline{c_{i,j}^{(2)}}$，$\underline{A^{(2)}}B$ の要素を $\underline{c_{i,j}^{(3)}}$ とし，倍精度の行列積の結果を $(c_{i,j}^*)$ とする．**図 5** のように，$c_{i,j}^*$ は 53 ビットの

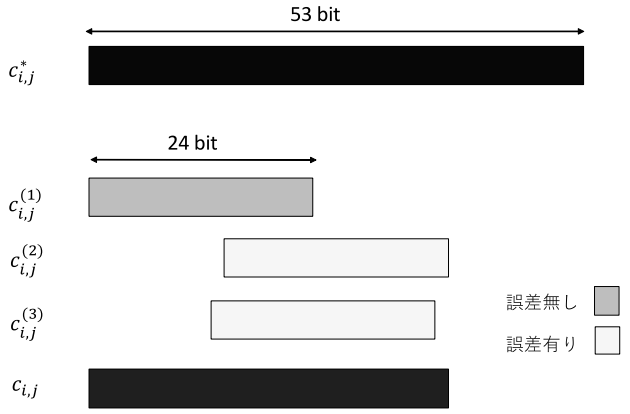

図 5　提案手法による精度の改善

Fig. 5　Improvement of precision by the proposed method.

仮数部を持ち，$c_{i,j}^{(1)}$ は誤差なしで最大 24 ビットの仮数部を持つ．$\underline{c_{i,j}^{(2)}}$，$\underline{c_{i,j}^{(3)}}$ は誤差を含むが，それぞれ $\underline{B^{(2)}}$，$\underline{A^{(2)}}$ のように下位ビット行列の行列積であるので，絶対値は $c_{i,j}^{(1)}$ より小さいと期待できる．よって，図 5 で示すような位置に $\underline{c_{i,j}^{(2)}}$，$\underline{c_{i,j}^{(3)}}$ があるとすると，$c_{i,j}$ は 24 ビットより桁数の多い仮数部を持つことができる．よって，2 分割することによって倍精度の近似解を単精度より高い精度で計算できるのではないかと考えられる．

アルゴリズム 2 提案する倍精度行列積の近似計算アルゴリズム（2 分割の場合）

Input: A, B　　$\%A, B$ は double 型
Output: C　　$\%\ C$ は double 型
1: $n = \mathrm{size}(A, 2)$;
2: $\mu_A = \max(\mathrm{abs}(A), [\,], 2)$;
　　$\%\mu_A \in \mathbb{F}^{m\times 1}\ \mu_A(i) = \max_{1\le k\le n} |a_{i,k}|$
3: $\mu_B = \max(\mathrm{abs}(B), [\,], 1)$;
　　$\%\mu_B \in \mathbb{F}^{1\times p}\ \mu_B(j) = \max_{1\le k\le n} |b_{k,j}|$
4: $\beta = \left\lceil \frac{24+\log_2(n)}{2} \right\rceil$;
5: $t_A = 2.\hat{}(\lceil \log_2(\mu_A) \rceil + \beta)$;
6: $t_B = 2.\hat{}(\lceil \log_2(\mu_B) \rceil + \beta)$;
7: $S_A = \mathrm{single}\,(\mathrm{repmat}(t_A, 1, n))$;
　　$\%S_A = t_A \cdot e^T$ for $e = (1, 1, \cdots, 1)^T \in \mathbb{F}^n$
8: $S_B = \mathrm{single}\,(\mathrm{repmat}(t_B, n, 1))$;
　　$\%S_B = e \cdot t_B$ for $e = (1, 1, \cdots, 1)^T \in \mathbb{F}^n$
9: $A^{(1)} = (\mathrm{single}(A) + S_A) - S_A$;　　$\%\ A^{(1)}$ は float 型
10: $\underline{A^{(2)}} = \mathrm{single}(A - \mathrm{double}(A^{(1)}))$;　　$\%\ A^{(2)}$ は float 型
11: $B^{(1)} = (\mathrm{single}(B) + S_B) - S_B$;　　$\%\ B^{(1)}$ は float 型
12: $\underline{B^{(2)}} = \mathrm{single}(B - \mathrm{double}(B^{(1)}))$;　　$\%\ B^{(2)}$ は float 型
13: $C = A^{(1)} * B^{(1)} + A^{(1)} * \underline{B^{(2)}} + \underline{A^{(2)}} * \mathrm{single}(B)$;

入力行列を 2 分割する場合の提案手法の疑似コードをアルゴリズム 2 に示す．ただし，入力行列に含まれる倍精度浮動小数点数の指数部は単精度浮動小数の指数部で表現できる範囲にあると仮定する．アルゴリズム 2 の 10 行目で単精度に丸める前の右辺の値を $\underline{A^{(2)}}'$ とする．$\underline{A^{(2)}}$ は一般には丸め誤差を含む．このためアルゴリズム 2 では，$A = A^{(1)} + \underline{A^{(2)}}'$ は成り立つが，$A = A^{(1)} + \underline{A^{(2)}}$ は一般には成り立たず，$A \approx A^{(1)} + \underline{A^{(2)}}$ が成り立つのみである．

```
__device__ float atomicMax(float* addr, float val)
// 'val' must be non-negative.
{
    unsigned old = atomicMax((unsigned*)addr,*((unsigned*)&val));

    return *((float*)&old);
}
```

図 6　非負 float 版の atomixMax の提案実装

Fig. 6　Our implementation of atomicMax for non-negative float.

B についても同様であるため，アルゴリズム 2 が求める C はこの丸め誤差による誤差を含む．ただし，$\underline{A^{(2)}}$ および $\underline{B^{(2)}}$ は下位ビットのため，この丸め誤差による影響は比較的小さいと考えられる．3 分割以上に分割する場合への拡張は自明であるので省略する．以下，GPU で並列計算した部分について説明する．

アルゴリズム 2 の 2 行目の行列 A の各行の絶対値の最大値を求めるためのカーネルと，3 行目の行列 B の各列の絶対値の最大値を求めるためのカーネルをそれぞれ別のカーネルで計算する．まず絶対値を求める計算は並列に計算する．そして，行列 A の各行の絶対値の最大値，または行列 B の各列の絶対値の最大値を求めるときに atomicMax 関数を用いて計算する．ここで，float の atomicMax は CUDA では提供されていないので，「任意の非負の float 値 x，y に対して，これらを 2 の補数表現の 32 ビット整数と見なしたときの整数値を x'，y' とすると，$x < y$ のとき，かつ，そのときに限り $x' < y'$ が成り立つ」ことを利用して，float へのポインタを型キャストで unsigned へのポインタと見なして，unsigned 用の atomicMax を呼ぶことにより，非負の float 値用の atomicMax 関数を実装した．実装コードを**図 6** に示す．

$$\mathrm{atomicMax}(\&\mu_A[i], |a_{i,k}|) \tag{17}$$

$$\mathrm{atomicMax}(\&\mu_B[j], |b_{k,j}|) \tag{18}$$

をすべての (i,k)，(k,j) に対して並列に実行することで，行列 A の各行の絶対値の最大値，または行列 B の各列の絶対値の最大値を求める．ブロック間同期をとるため，ここまでを 1 つのカーネルとする．

続いて，アルゴリズム 2 の 5，6 行目を 1 つのカーネルで計算する．この部分は μ_A，μ_B の各要素を並列に計算する．

さらにアルゴリズム 2 の 7〜12 行目を行列 A，B に対してそれぞれ別のカーネルで計算する．7，8 行目は，

$$s_{a_{i,k}} = t_A(i) \quad \text{for} \quad 1 \le i \le m, 1 \le k \le n \tag{19}$$

$$s_{b_{k,j}} = t_B(j) \quad \text{for} \quad 1 \le k \le n, 1 \le j \le p \tag{20}$$

と等価である．そして，9〜12 行目は行列の各要素を並列計算する．ここまでが行列分割の計算である．

アルゴリズム 2 の 13 行目の $A^{(1)}B^{(1)}$，$A^{(1)}\underline{B^{(2)}}$，$\underline{A^{(2)}}B$

の 3 つの行列積計算は cuBLAS を用いて計算する．

最後に，1 つのカーネルで 3 つの和計算を double 型で計算することで，単精度の行列積計算を用いて倍精度の行列積の近似解を計算する．

5. 評価実験

5.1 実験方法

4 章で提案した手法を GPU 上で実行して評価した．評価に用いた GPU は NVIDIA GeForce RTX 2080 SUPER，CPU は Intel Core i3-8100，CUDA のバージョンは 11.0 Update1，OS は Windows 10 Pro，コンパイラは Visual Studio 2019 である．用いた GPU の仕様 [13] を**表 1** に示す．

実験条件として，行列の要素 $a_{i,k}$，$b_{k,j}$ は文献 [9] の評価実験と同様に，$(ru - 0.5) \times \exp(\phi \times rn)$ で計算される乱数値とした．ここで ru は $[0, 1)$ の一様乱数，rn は標準正規分布に従う乱数，ϕ は計算される乱数値の絶対値の範囲を制御するパラメータであり，ϕ を大きくするほど範囲は広くなる．また，CPU から GPU および GPU から CPU への行列データのコピー時間は考慮しないものとした．すなわち，計算対象の行列データはすでに VRAM 上にあり，計算結果も VRAM 上に出力するものとして，GPU 上での計算時間を評価した．

4 つの実験を行い，次の 4 つの項目を評価した．

(1) 単精度，2 分割，3 分割，4 分割，5 分割，6 分割，倍精度の行列積計算の実行時間の比較．
(2) 入力行列の要素の値を $\phi = 0.1, 1, 2$ の 3 種類でそれぞれ 10 通りに変化させ，単精度，2 分割，3 分割，4 分割，5 分割，6 分割，倍精度行列積の結果の多倍長行列積に対する最大相対誤差の比較．
(3) (2) の実験の最大相対誤差と，そのときの実行時間の関係．
(4) 提案手法の実行時間内訳．

最大相対誤差は以下のように表されるものとする．ここで $c^*_{i,j}$ は精度 2,048 bit の多倍長精度の行列積の結果の要素，$c_{i,j}$ は単精度，2 分割，3 分割，4 分割，5 分割，6 分割，倍精度の行列積の結果の要素である．

$$\max_{1 \le i,j \le n} \frac{|c^*_{i,j} - c_{i,j}|}{|c^*_{i,j}|} \tag{21}$$

多倍長演算には MPIR [14] のバージョン 3.0.0 を用いた．

表 1　GeForce RTX 2080 SUPER の仕様
Table 1　The specification of GeForce RTX 2080 SUPER.

CUDA コア	3,072 基
ベースクロック	1,650 MHz
VRAM	8 GB

5.2 実験結果

5.2.1 単精度，2 分割，3 分割，4 分割，5 分割，6 分割，倍精度の行列積計算の実行時間の比較

行列のサイズが 128 × 128，256 × 256，512 × 512，1,024 × 1,024，2,048 × 2,048，4,096 × 4,096 の場合に，単精度，2 分割，3 分割，4 分割，5 分割，6 分割，倍精度の行列積計算の実行時間を計測した．測定結果を**図 7** の対数グラフに示す．

図 7 より，行列のサイズが 128 × 128 より大きいときに，2 分割し 3 回行列積計算を行う方が倍精度の行列積計算よりも高速に計算することができ，行列のサイズが 1,024 × 1,024 では 4.66 倍，行列のサイズが 2,048 × 2,048 では 6.15 倍，行列のサイズが 4,096 × 4,096 では 8.44 倍高速に計算を行うことができた．行列のサイズが 512 × 512 より大きいときに，3 分割し 6 回行列積計算を行う方が倍精度の行列積計算よりも高速に計算することができ，行列のサイズが 1,024 × 1,024 では 2.36 倍，行列のサイズが 2,048 × 2,048 では 3.09 倍，行列のサイズが 4,096 × 4,096 では 4.14 倍高速に計算を行うことができた．行列のサイズが 512 × 512 より大きいときに，4 分割し 10 回行列積計算を行う方が倍精度の行列積計算よりも高速に計算することができ，行列のサイズが 1,024 × 1,024 では 1.50 倍，行列のサイズが 2,048 × 2,048 では 1.93 倍，行列のサイズが 4,096 × 4,096 では 2.50 倍高速に計算を行うことができた．行列のサイズが 1,024 × 1,024 より大きいときに，5 分割し 15 回行列積計算を行う方が倍精度の行列積計算よりも高速に計算することができ，行列のサイズが 1,024 × 1,024 では 1.06 倍，行列のサイズが 2,048 × 2,048 では 1.34 倍，行列のサイズが 4,096 × 4,096 では 1.71 倍高速に計算を行うことができた．行列のサイズが 1,024 × 1,024 では，倍精度の行列積計算の方が 6 分割し 21 回行列積計算を行うよりも高速に計算されていた．また，行列のサイズが 2,048 × 2,048 では倍精度の行列積計算と 6 分割し 21 回行列積計算を行

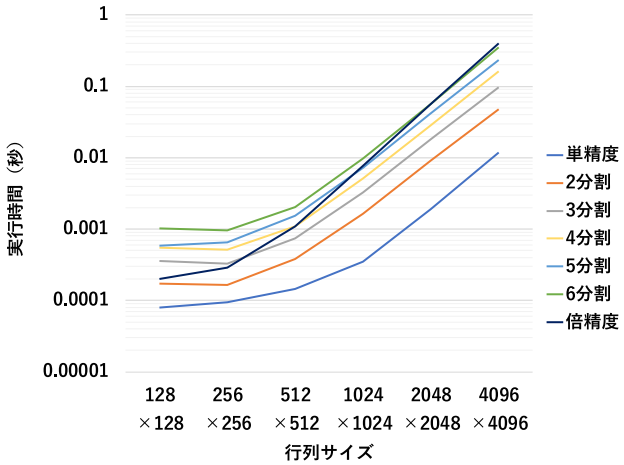

図 7　手法別実行時間
Fig. 7　Computing time for each method.

う計算速度はほぼ同じくらいの計算速度であった．行列のサイズが 4,096 × 4,096 では，6 分割し 21 回行列積計算を行う方が倍精度の行列積計算よりも高速に計算することができ，1.14 倍高速に計算を行うことができた．

よって行列が小さくない場合は，行列を分割し，単精度で複数回行列積計算を行うことは倍精度の行列積計算を 1 度行うよりも高速であった．しかし，5 分割し行列積計算を 15 回行うと，行列のサイズが 1,024 × 1,024 では倍精度と同じくらいの計算時間がかかり，6 分割し行列積計算を 21 回行うと，行列のサイズが 2,048 × 2,048 では倍精度と同じくらいの計算時間がかかった．

5.2.2　単精度，2 分割，3 分割，4 分割，5 分割，6 分割，倍精度行列積の最大相対誤差の比較

行列のサイズが 1,024×1,024, 2,048×2,048, 4,096×4,096 の場合に，$\phi = 0.1, 1, 2$ の 3 種類ぞれぞれで乱数の種の値を 1 から 10 までの 10 通りに変化させ，単精度，2 分割，3 分割，4 分割，5 分割，6 分割，倍精度行列積の結果の最大相対誤差（式 (21)）を比較した．$\phi = 0.1$ のときの実験結果を**図 8**，**図 9**，**図 10**，$\phi = 1$ のときの実験結果を**図 11**，**図 12**，**図 13**，$\phi = 2$ のときの実験結果を**図 14**，**図 15**，**図 16** の対数グラフに示す．

乱数の種によって結果に差があるが，$\phi = 0.1$ のときは種を固定した場合はいずれの場合も，分割数を増やすと最大相対誤差が小さくなった．また，6 分割し行列積計算を行うことによって，倍精度よりも高精度な結果を得られる場合もあった．$\phi = 1$ のときは，$\phi = 0.1$ に比べ分割により最大相対誤差が小さくなる割合が小さかった．また，分割数を増やすと最大相対誤差は小さくなる傾向があるが，行列のサイズ 4,096 × 4,096 の乱数の種が 7 のときに 2 分割の方が 3 分割よりも最大相対誤差が小さかった．$\phi = 2$ では，$\phi = 1$ よりも分割により最大相対誤差が小さくなる割合が小さかった．また，行列のサイズが 1,024 × 1,024 では分割数を増やすと最大相対誤差が小さくなるが，行列のサイズが 2,048 × 2,048，4,096 × 4,096 では分割によって悪化する場合が発生し，単精度よりも悪化する場合もあった．

それぞれの手法の最大相対誤差の平均値を**表 2** に示す．最大相対誤差の平均値を比較すると，行列のサイズと ϕ の値にかかわらず，分割して行列積を計算する方が単精度行列積よりも高精度の近似解を求められる傾向があり，2 分割よりも 3 分割，3 分割よりも 4 分割，4 分割よりも 5 分割，5 分割よりも 6 分割の方が高精度の近似解を求められる傾向があった．

5.2.3　単精度，2 分割，3 分割，4 分割，5 分割，6 分割，倍精度の最大相対誤差と実行時間

行列のサイズが 1,024×1,024, 2,048×2,048, 4,096×4,096 の場合に，$\phi = 0.1, 1, 2$ で乱数の種の値を 1〜10 の 10 通りに変化させ，単精度，2 分割，3 分割，4 分割，5 分割，6 分割，倍精度行列積の最大相対誤差（式 (21)）と実行時間の

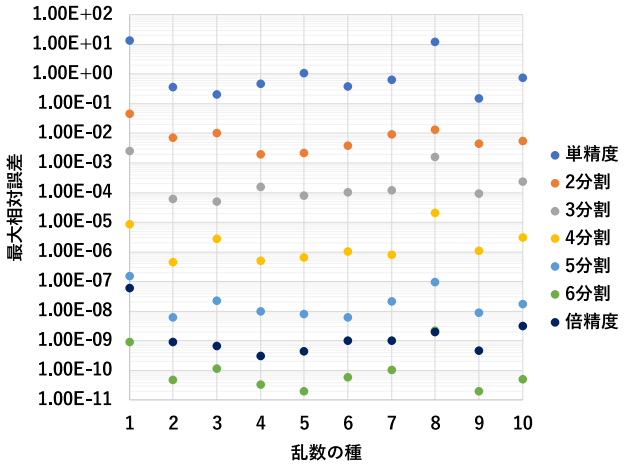

図 8　行列サイズ 1,024 × 1,024，$\phi = 0.1$ の場合の最大相対誤差
Fig. 8　Maximum relative error with matrix size 1,024 × 1,024, $\phi = 0.1$.

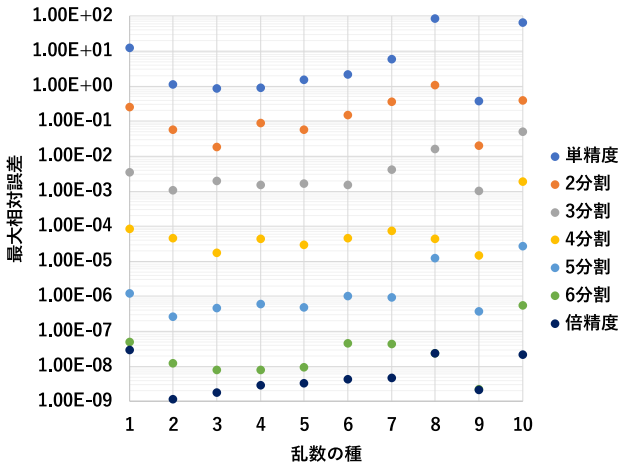

図 9　行列サイズ 2,048 × 2,048，$\phi = 0.1$ の場合の最大相対誤差
Fig. 9　Maximum relative error with matrix size 2,048 × 2,048, $\phi = 0.1$.

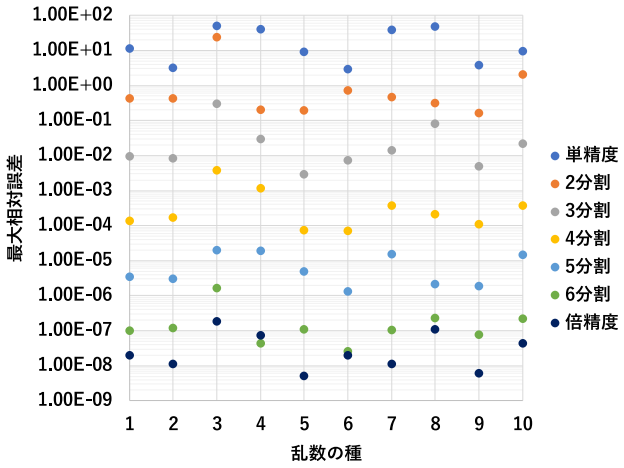

図 10　行列サイズ 4,096 × 4,096，$\phi = 0.1$ の場合の最大相対誤差
Fig. 10　Maximum relative error with matrix size 4,096×4,096, $\phi = 0.1$.

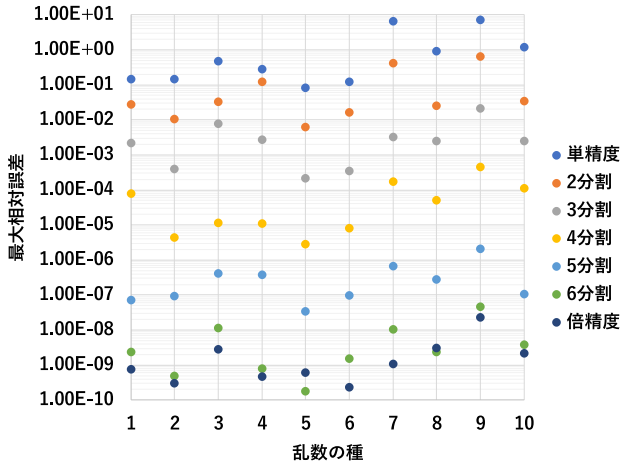

図 11　行列サイズ 1,024 × 1,024，$\phi = 1$ の場合の最大相対誤差

Fig. 11　Maximum relative error with matrix size 1,024×1,024, $\phi = 1$.

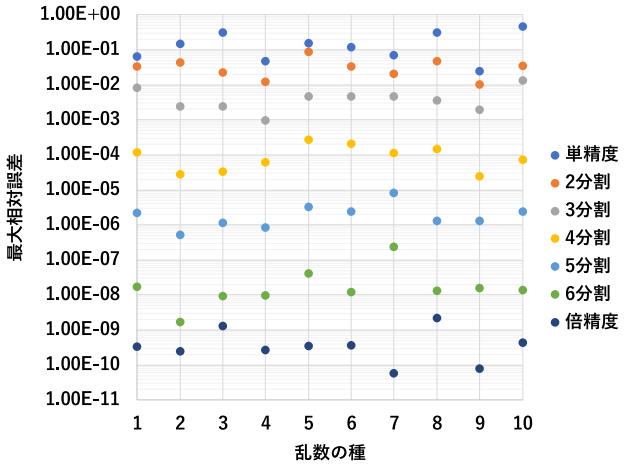

図 14　行列サイズ 1,024 × 1,024，$\phi = 2$ の場合の最大相対誤差

Fig. 14　Maximum relative error with matrix size 1,024×1,024, $\phi = 2$.

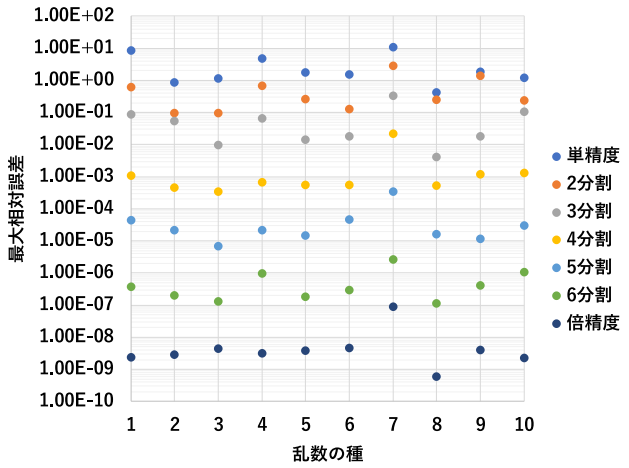

図 12　行列サイズ 2,048 × 2,048，$\phi = 1$ の場合の最大相対誤差

Fig. 12　Maximum relative error with matrix size 2,048×2,048, $\phi = 1$.

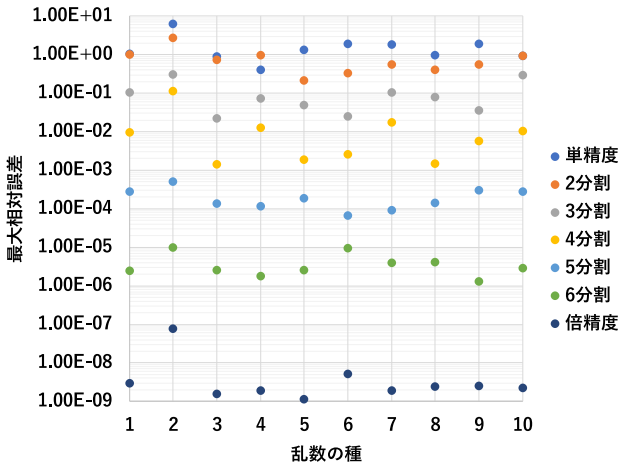

図 15　行列サイズ 2,048 × 2,048，$\phi = 2$ の場合の最大相対誤差

Fig. 15　Maximum relative error with matrix size 2,048×2,048, $\phi = 2$.

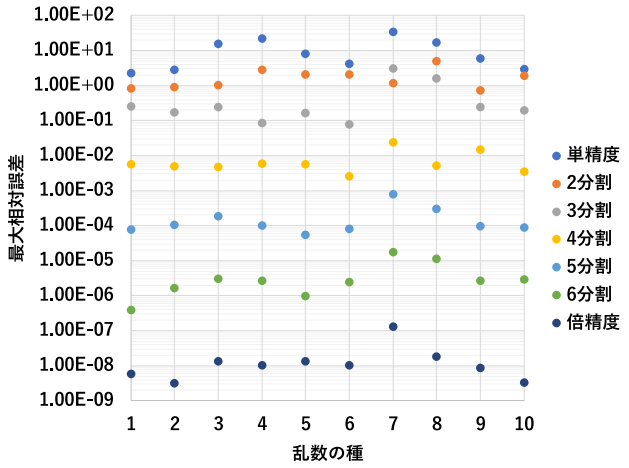

図 13　行列サイズ 4,096 × 4,096，$\phi = 1$ の場合の最大相対誤差

Fig. 13　Maximum relative error with matrix size 4,096×4,096, $\phi = 1$.

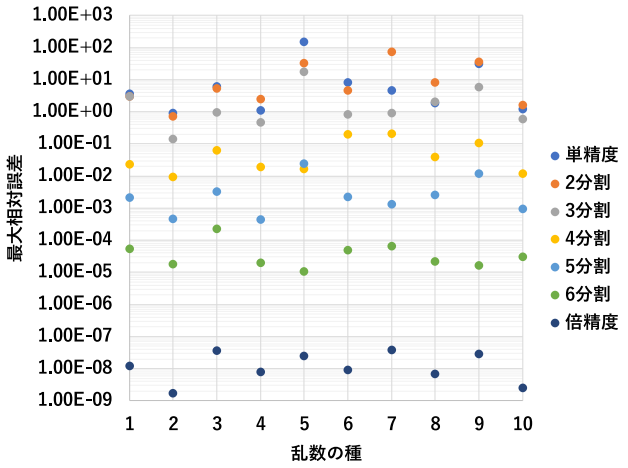

図 16　行列サイズ 4,096 × 4,096，$\phi = 2$ の場合の最大相対誤差

Fig. 16　Maximum relative error with matrix size 4,096×4,096, $\phi = 2$.

表 2　それぞれの手法の最大相対誤差の平均値
Table 2　Mean of maximum relative error for each method.

ϕ	行列サイズ	単精度	2 分割	3 分割	4 分割	5 分割	6 分割	倍精度
0.1	1,024 × 1,024	2.95e+0	1.05e-2	4.93e-4	3.97e-6	3.50e-8	3.57e-10	6.92e-9
	2,048 × 2,048	1.74e+1	2.45e-1	8.26e-3	2.29e-4	4.55e-6	7.54e-8	9.40e-9
	4,096 × 4,096	2.18e+1	2.92e+0	4.81e-2	6.48e-4	8.64e-6	2.71e-7	4.88e-8
1	1,024 × 1,024	1.69e+0	1.33e-1	4.34e-3	9.09e-5	4.18e-7	7.87e-9	3.42e-9
	2,048 × 2,048	3.26e+0	6.63e-1	7.01e-2	2.85e-3	5.53e-5	6.37e-7	1.16e-8
	4,096 × 4,096	1.14e+1	1.83e+0	6.02e-1	7.69e-3	1.87e-4	4.54e-6	2.16e-8
2	1,024 × 1,024	1.72e-1	3.45e-2	4.68e-3	1.08e-4	2.38e-6	3.65e-8	5.69e-10
	2,048 × 2,048	1.75e+0	8.40e-1	1.09e-1	1.77e-2	2.12e-4	4.10e-6	9.80e-9
	4,096 × 4,096	2.12e+1	1.68e+1	3.24e+0	6.86e-2	4.99e-3	5.05e-5	1.69e-8

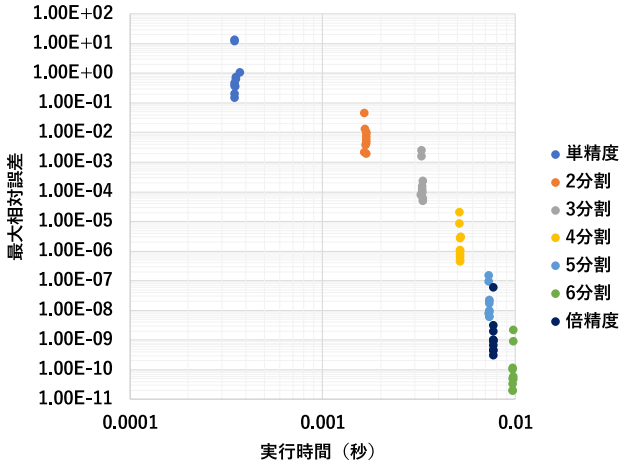

図 17　最大相対誤差と実行時間の関係（行列サイズ 1,024 × 1,024, $\phi = 0.1$）
Fig. 17　Relation between maximum relative error and computing time (matrix size 1,024 × 1,024, $\phi = 0.1$).

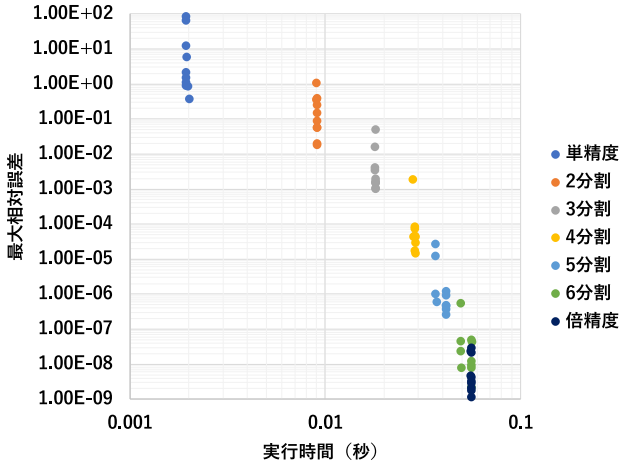

図 18　最大相対誤差と実行時間の関係（行列サイズ 2,048 × 2,048, $\phi = 0.1$）
Fig. 18　Relation between maximum relative error and computing time (matrix size 2,048 × 2,048, $\phi = 0.1$).

関係を調べた．$\phi = 0.1$ のときの実験結果を**図 17**，**図 18**，**図 19**，$\phi = 1$ のときの実験結果を**図 20**，**図 21**，**図 22**，$\phi = 2$ のときの実験結果を**図 23**，**図 24**，**図 25** に示す．

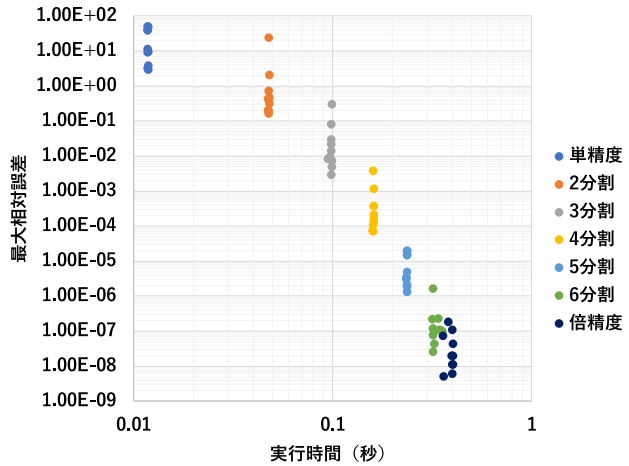

図 19　最大相対誤差と実行時間の関係（行列サイズ 4,096 × 4,096, $\phi = 0.1$）
Fig. 19　Relation between maximum relative error and computing time (matrix size 4,096 × 4,096, $\phi = 0.1$).

10 通りの平均の実行時間と最大相対誤差の改善比率を**表 3** に示す．

行列のサイズが 1,024×1,024, 2,048×2,048, 4,096×4,096 のときの比較では，$\phi = 0.1$ のとき，2 分割行列積計算は，単精度行列積計算より約 4 倍～5 倍遅くなるが，最大相対誤差の平均値は約 7e+0 倍～3e+2 倍良くなった．3 分割行列積計算は，単精度行列積計算より約 8 倍～10 倍遅くなるが，最大相対誤差の平均値は約 5e+2 倍～6e+3 倍良くなった．4 分割行列積計算は，単精度行列積計算より約 13 倍～15 倍遅くなるが，最大相対誤差の平均値は約 3e+4 倍～7e+5 倍良くなった．5 分割行列積計算は，単精度行列積計算より約 20 倍～21 倍遅くなるが，最大相対誤差の平均値は約 3e+6 倍～8e+7 倍良くなった．6 分割行列積計算は，単精度行列積計算より約 27 倍～28 倍遅くなるが，最大相対誤差の平均値は約 8e+7 倍～8e+9 倍良くなった．また，倍精度行列積計算は，単精度行列積計算より約 22 倍～33 倍遅くなるが，最大相対誤差の平均値は約 4e+8 倍～2e+9 倍良くなっている．このことより，10 進数で考えると，2 分割で有効数字 0～2 桁，3 分割で有効数字 2～3 桁，4 分

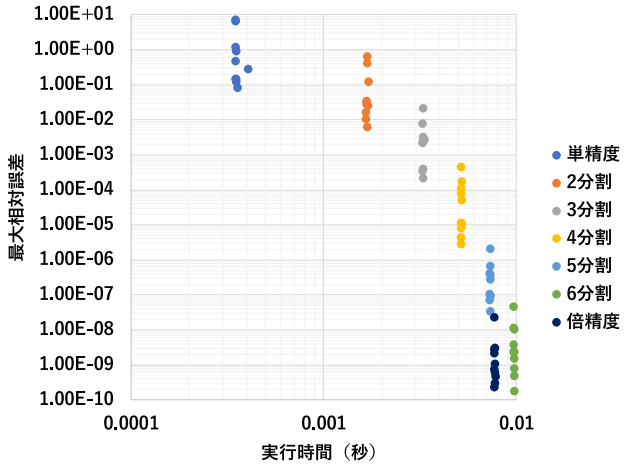

図 20　最大相対誤差と実行時間の関係（行列サイズ 1,024 × 1,024, $\phi = 1$）

Fig. 20　Relation between maximum relative error and computing time (matrix size 1,024 × 1,024, $\phi = 1$).

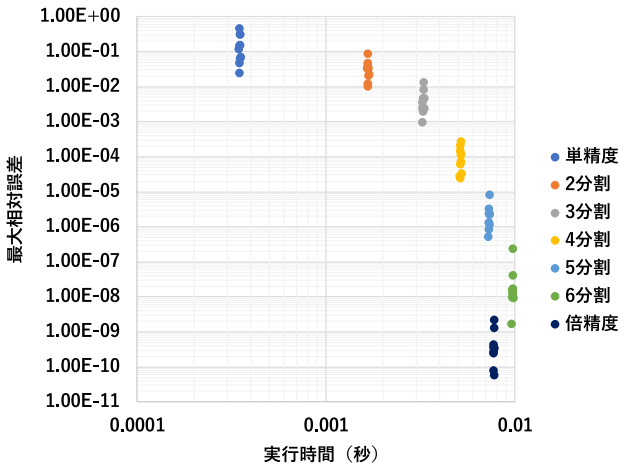

図 23　最大相対誤差と実行時間の関係（行列サイズ 1,024 × 1,024, $\phi = 2$）

Fig. 23　Relation between maximum relative error and computing time (matrix size 1,024 × 1,024, $\phi = 2$).

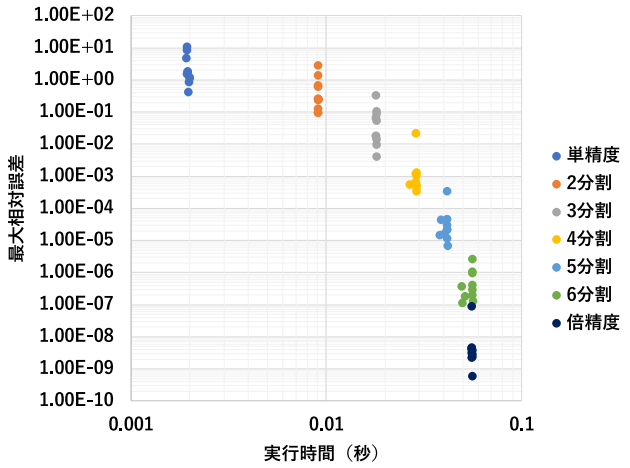

図 21　最大相対誤差と実行時間の関係（行列サイズ 2,048 × 2,048, $\phi = 1$）

Fig. 21　Relation between maximum relative error and computing time (matrix size 2,048 × 2,048, $\phi = 1$).

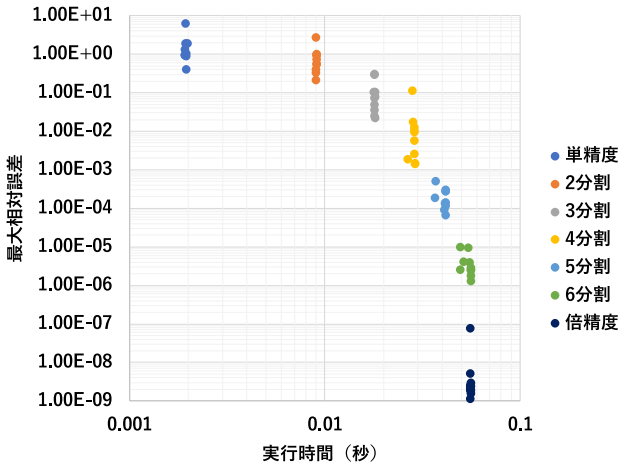

図 24　最大相対誤差と実行時間の関係（行列サイズ 2,048 × 2,048, $\phi = 2$）

Fig. 24　Relation between maximum relative error and computing time (matrix size 2,048 × 2,048, $\phi = 2$).

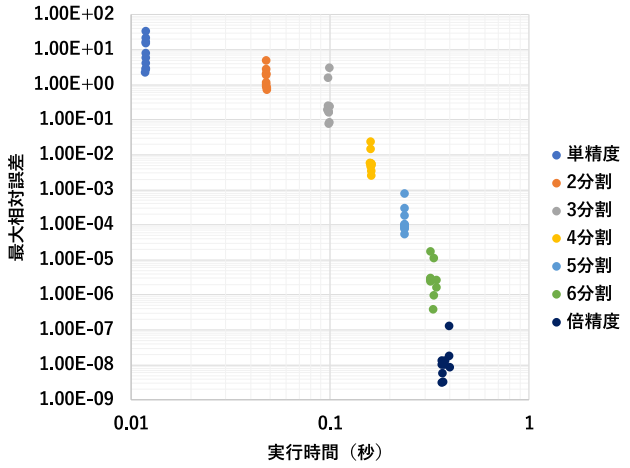

図 22　最大相対誤差と実行時間の関係（行列サイズ 4,096 × 4,096, $\phi = 1$）

Fig. 22　Relation between maximum relative error and computing time (matrix size 4,096 × 4,096, $\phi = 1$).

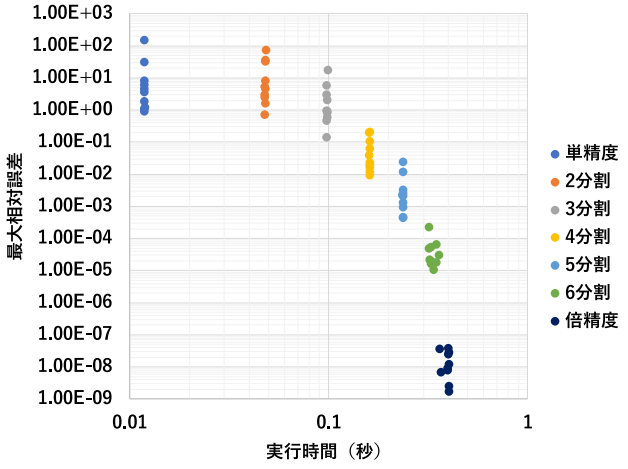

図 25　最大相対誤差と実行時間の関係（行列サイズ 4,096 × 4,096, $\phi = 2$）

Fig. 25　Relation between maximum relative error and computing time (matrix size 4,096 × 4,096, $\phi = 2$).

表 3　実行時間と最大相対誤差の平均値の比率
Table 3　Ratio of mean of computing time and maximum relative error.

ϕ	行列サイズ		単精度 / 2 分割	単精度 / 3 分割	単精度 / 4 分割	単精度 /5 分割	単精度 /6 分割	単精度 /倍精度
0.1	1,024 × 1,024	実行時間	4.74 倍	9.29 倍	14.6 倍	20.7 倍	27.5 倍	21.8 倍
		最大相対誤差	2.81e+2 倍	5.97e+3 倍	7.43e+5 倍	8.41e+7 倍	8.25e+9 倍	4.26e+8 倍
	2,048 × 2,048	実行時間	4.69 倍	9.31 倍	14.8 倍	21.2 倍	27.4 倍	28.6 倍
		最大相対誤差	7.09e+1 倍	2.10e+3 倍	7.60e+4 倍	3.82e+6 倍	2.31e+8 倍	1.85e+9 倍
	4,096 × 4,096	実行時間	4.05 倍	8.31 倍	13.6 倍	20.0 倍	27.8 倍	33.0 倍
		最大相対誤差	7.45e+0 倍	4.52e+2 倍	3.36e+4 倍	2.52e+6 倍	8.03e+7 倍	4.46e+8 倍
1	1,024 × 1,024	実行時間	4.71 倍	9.21 倍	14.5 倍	20.5 倍	27.3 倍	21.7 倍
		最大相対誤差	1.27e+1 倍	3.89e+2 倍	1.86e+4 倍	4.04e+6 倍	2.15e+8 倍	4.94e+8 倍
	2,048 × 2,048	実行時間	4.64 倍	9.22 倍	14.7 倍	20.9 倍	27.7 倍	28.5 倍
		最大相対誤差	4.91e+0 倍	4.65e+1 倍	1.14e+3 倍	5.89e+4 倍	5.11e+6 倍	2.82e+8 倍
	4,096 × 4,096	実行時間	4.05 倍	8.32 倍	13.6 倍	20.0 倍	27.7 倍	31.8 倍
		最大相対誤差	6.24e+0 倍	1.89e+1 倍	1.48e+3 倍	6.10e+4 倍	2.51e+6 倍	5.27e+8 倍
2	1,024 × 1,024	実行時間	4.77 倍	9.37 倍	14.8 倍	20.9 倍	27.8 倍	22.1 倍
		最大相対誤差	4.98e+0 倍	3.66e+1 倍	1.59e+3 倍	7.22e+4 倍	4.70e+6 倍	3.02e+8 倍
	2,048 × 2,048	実行時間	4.66 倍	9.27 倍	14.7 倍	20.9 倍	27.8 倍	28.7 倍
		最大相対誤差	2.09e+0 倍	1.61e+1 倍	9.89e+1 倍	8.27e+3 倍	4.28e+5 倍	1.79e+8 倍
	4,096 × 4,096	実行時間	4.05 倍	8.26 倍	13.5 倍	19.9 倍	28.1 倍	33.1 倍
		最大相対誤差	1.26e+0 倍	6.55e+0 倍	3.10e+2 倍	4.26e+3 倍	4.20e+5 倍	1.25e+9 倍

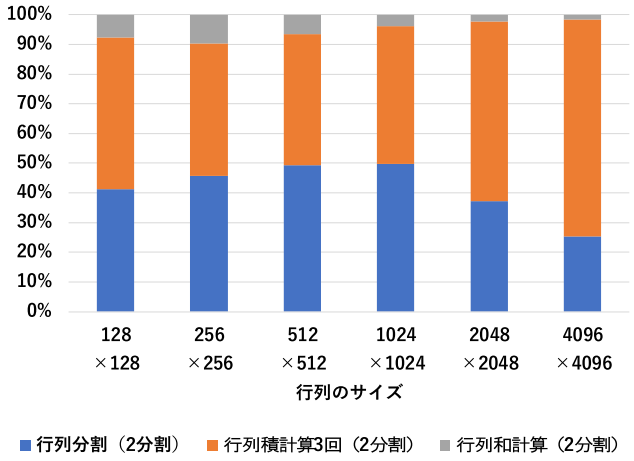

図 26　提案手法の実行時間内訳（2 分割の場合）
Fig. 26　A breakdown of computing time of the proposed method (in case of two partition).

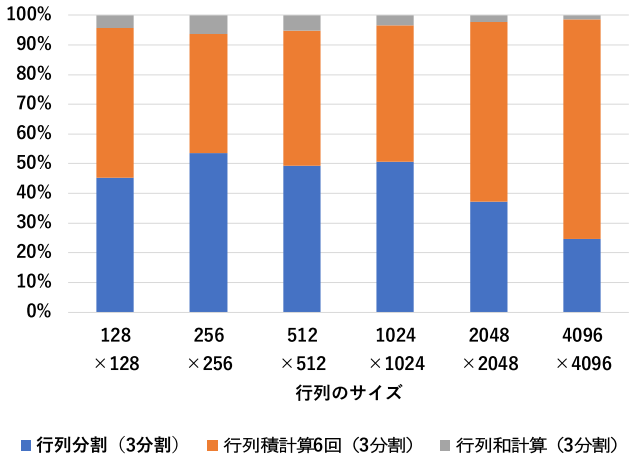

図 27　提案手法の実行時間内訳（3 分割の場合）
Fig. 27　A breakdown of computing time of the proposed method (in case of three partition).

割で有効数字 4〜5 桁，5 分割で有効数字 6〜7 桁，6 分割で有効数字 7〜9 桁分は単精度行列積より高精度な近似解を求められる．計算時間は ϕ の値を変更しても同じ傾向だが，$\phi = 1$ のとき，最大相対誤差の平均値は，2 分割の場合約 6e+0 倍〜1e+1 倍，3 分割の場合約 2e+1 倍〜4e+2 倍，4 分割の場合約 1e+3 倍〜2e+4 倍，5 分割の場合約 6e+4 倍〜4e+6 倍，6 分割の場合約 3e+6 倍〜2e+8 倍良くなった．また，倍精度行列積計算では，最大相対誤差の平均値は約 3e+8 倍〜5e+8 倍良くなっている．$\phi = 2$ のとき，最大相対誤差の平均値は，2 分割の場合約 1e+0 倍〜5e+0 倍，3 分割の場合約 7e+0 倍〜4e+1 倍，4 分割の場合約 3e+2 倍〜2e+3 倍，5 分割の場合約 4e+3 倍〜7e+4 倍，6 分割の場合約 4e+5 倍〜5e+6 倍良くなった．また，倍精度行列積計算では，最大相対誤差の平均値は約 2e+8 倍〜1e+9 倍良くなっている．

5.2.4　提案手法の実行時間内訳

提案手法の実行における行列分割，行列積計算，行列和計算の 3 過程の実行時間の割合を，行列のサイズが 128×128, 256×256, 512×512, 1,024×1,024, 2,048×2,048, 4,096×4,096 の場合に比較した．実験結果を**図 26**，**図 27**，**図 28**，**図 29**，**図 30** に示す．

2 分割，3 分割，4 分割，5 分割，6 分割行列積計算ともに行列のサイズが大きくなるにつれて行列積計算にかかる時間の割合が増加したが，行列のサイズが小さいときは行

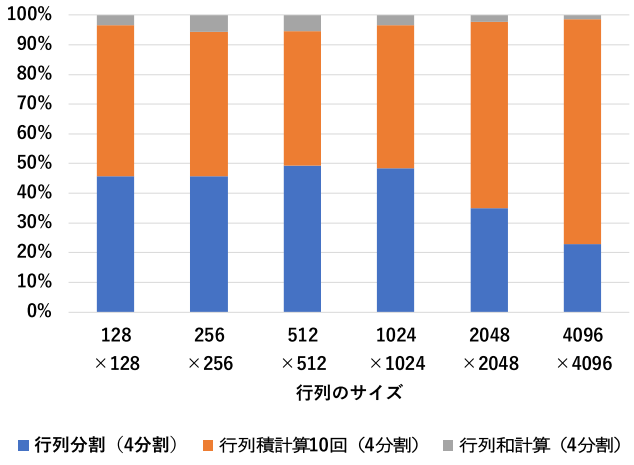

図 28　提案手法の実行時間内訳（4 分割の場合）

Fig. 28　A breakdown of computing time of the proposed method (in case of four partition).

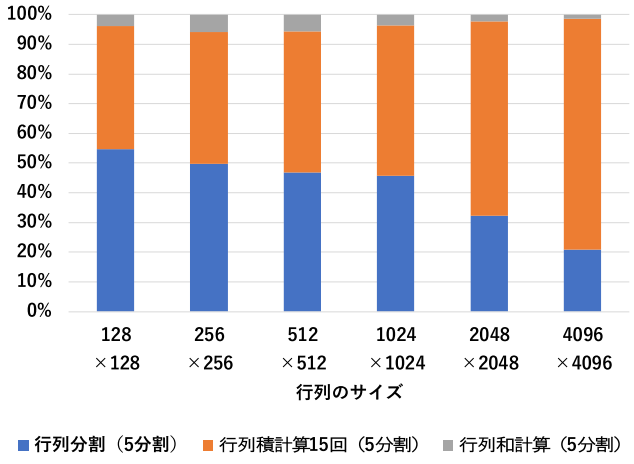

図 29　提案手法の実行時間内訳（5 分割の場合）

Fig. 29　A breakdown of computing time of the proposed method (in case of five partition).

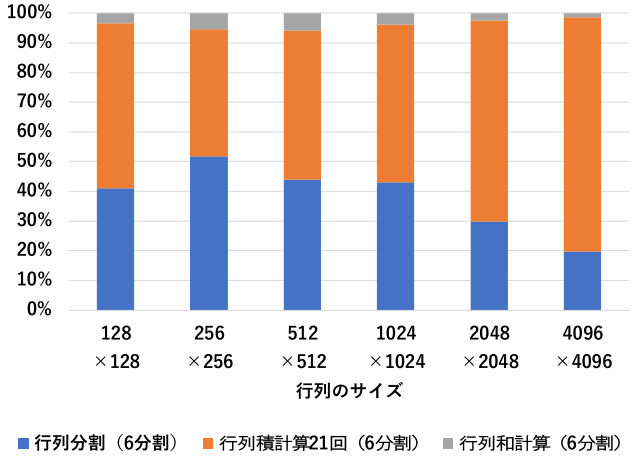

図 30　提案手法の実行時間内訳（6 分割の場合）

Fig. 30　A breakdown of computing time of the proposed method (in case of six partition).

列分割が占める割合が高かった．この行列分割を高速化することができれば，解をより高速に求めることが可能になると考えられる．

6. まとめと今後の課題

安価なゲーミング用 GPU 上で倍精度よりも高速に計算ができ，単精度よりも高精度な計算結果を得られる行列積の計算手法を提案し，実装して評価を行った．

$\phi = 0.1$ で生成されるような乱数では，単精度よりも高精度の計算を倍精度よりも高速に行うことができた．

しかし，$\phi = 1, 2$ で生成されるような値の大きな要素と小さな要素といった極端に差がある要素を持つ入力行列では，2 分割ではあまり改善されず，4 分割，5 分割，6 分割する必要があった．しかし，分割数を増やすと倍精度に比べ高速に計算できるメリットが減ってしまうという問題点がある．また，2 分割計算では 3 回，3 分割計算では 6 回，4 分割計算では 10 回，5 分割計算では 15 回，6 分割計算では 21 回単精度行列積を計算する必要があるため，提案手法にはメモリ使用量が増大してしまう欠点がある．

今後の課題としては，値の大きな要素と小さな要素といった極端に差がある要素を持つ入力行列への対応を考える必要がある．また，入力行列が疎行列の場合の実装を考える必要がある．単精度計算を使って倍精度計算の近似を行う手法としては，2 個の倍精度浮動小数点数を用いて 4 倍精度に近い精度を実現する double-double と呼ばれる手法 [15] に類似する手法として，2 個の単精度浮動小数点数を用いて倍精度に近い精度を実現する手法（double-float）も考えられる．double-float の行列積の GPU 用高性能実装と提案手法の比較も今後の課題である．

謝辞　本研究は JSPS 科研費 17K00171，20K11842 の助成を受けたものです．

参考文献

[1] NVIDIA: Inside Volta: The World's Most Advanced Data Center GPU (2017), available from ⟨https://devblogs.nvidia.com/inside-volta/⟩ (accessed 2020-11-26).
[2] NVIDIA：NVIDIA V100 世界最先端のデータセンター GPU，入手先 ⟨https://www.nvidia.com/ja-jp/data-center/tesla-v100/⟩（参照 2020-11-26）.
[3] NVIDIA: NVIDIA TURING GPU ARCHITECTURE, available from ⟨https://www.industry-era.com/images/pdf/NVIDIA-Turing-Architecture-Whitepaper.pdf⟩ (accessed 2020-11-26).
[4] 大石進一ほか：精度保証付き数値計算の基礎，コロナ社 (2018).
[5] Ozaki, K., Ogita, T., Oishi, S. and Rump, S.M.: Error Free Transformations of Matrix Multiplication by Using Fast Routines of Matrix Multiplication and Its Applications, *Numerical Algorithms*, Vol.59, No.1, pp.95–118 (2012).
[6] Ichimura, S., Ogita, T., Katagiri, T., Nagai, T. and Ozaki, K.: Threaded Accurate Matrix-Matrix Multipli-

cations with Sparse Matrix-Vector Multiplications, *Proc. 2018 IEEE International Parallel and Distributed Processing Symposium Workshops*, pp.1093–1102 (2018).
[7] BLAS (Basic Linear Algebra Subprograms), available from ⟨http://www.netlib.org/blas/⟩ (accessed 2020-11-26).
[8] NVIDIA: cuBLAS, available from ⟨https://docs.nvidia.com/cuda/cublas/index.html⟩ (accessed 2020-11-26).
[9] Mukunoki, D., Ozaki, K., Ogita, T. and Imamura, T.: DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions, *Lecture Notes in Computer Science*, Vol.12151, pp.230–248, Springer (2020).
[10] NVIDIA: CUDA C++ Programming Guide, available from ⟨https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html⟩ (accessed 2020-11-26).
[11] Cheng, J., Grossman, M. and McKercher, T.: Professional CUDA®C Programming, John Wiley & Sons, Inc. (2014).
[12] NVIDIA: Atomic Functions, available from ⟨https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions⟩ (accessed 2020-11-26).
[13] NVIDIA: GEFORCE®RTX 2080 SUPER™, available from ⟨https://www.nvidia.com/ja-jp/geforce/graphics-cards/rtx-2080-super/⟩ (accessed 2020-11-26).
[14] Gladman, B., Hart, W. and Moxham, J., et al.: MPIR: Multiple Precision Integers and Rationals, version 2.7.0, A fork of the GNU MP package, Granlund, T., et al., available from ⟨http://mpir.org⟩ (accessed 2020-11-26) (2015).
[15] Bailey, D.H.: High-Precision Software Directory, available from ⟨https://www.davidhbailey.com/dhbsoftware/⟩ (accessed 2020-11-26).

七井 香樹

1997 年 6 月 1 日生．2020 年 3 月大阪府立大学工学域電気電子系学類情報工学課程卒業．現在，同大学大学院工学研究科電気・情報系専攻知能情報工学分野在学中．GPU における高精度計算に関する研究に従事．

藤本 典幸　（正会員）

1969 年 4 月 14 日生．1997 年 3 月大阪大学大学院基礎工学研究科物理系専攻情報工学分野博士後期課程単位取得退学．同年 4 月大阪大学大学院基礎工学研究科助手．2002 年 4 月大阪大学大学院情報科学研究科助教授，2007 年 4 月同研究科准教授，2008 年 4 月大阪府立大学大学院理学系研究科教授，2016 年 4 月大阪府立大学大学院工学研究科教授となり現在に至る．情報科学，特に並列処理のソフトウェアに関する研究に従事．博士（工学）．2003 年度第 3 回船井情報科学奨励賞ほかを受賞．IEEE，ACM，電子情報通信学会等の会員．