Vol.2022-HPC-183 No.10
2022/3/17

全点対最短経路問題を解く Seidel のアルゴリズムの Tensor Core を用いた CUDA 実装

竹内 祐哉[†1,a)] 藤本 典幸[†1,b)]

概要：
近年，深層学習が注目を集めている．深層学習の計算の中心は行列積和演算であるが，最近の GPU が備えている Tensor Core は，これを半精度浮動小数点数で高速に実行できる．一方，行列積を用いて重みなし無向グラフを対称とした全点対最短経路問題を解く Seidel のアルゴリズムがある．本研究では，Seidel のアルゴリズムにおける行列積計算に Tensor Core を使うことで全点対最短経路問題が高速に解けることを示す．そして，この計算に半精度浮動小数点数を使うことによるオーバーフローの影響を議論する．NVIDIA A100 PCle 上で Tensor Core を使った GPU 実装では AMD Ryzen 5800X 上でマルチスレッド版 OpenBLAS と OpenMP を使った CPU 実装に比べて，頂点数 14336，次数 4096 の時，約 71.1 倍高速であった．

1. はじめに

近年，深層学習が注目を集めている．深層学習の計算の中心は行列積和演算であるが，最近の GPU が備えている Tensor Core[1] は，これを半精度浮動小数点数で高速に実行できる．半精度浮動小数点数で行列積計算を行う場合，オーバーフローを考慮する必要がある．

一方，全点対最短経路問題を解くアルゴリズムにワーシャル・フロイド法 [2] やダイクストラ法 [2] などがある．これらのアルゴリズムは頂点数 n に対して最悪 $O(n^3)$ 時間かかることが知られている．重み付きグラフの全点対最短経路問題を解くアルゴリズムはこれまで，様々なアルゴリズムが提案されてきたが，任意の $\epsilon > 0$ に対して $O(n^{3-\epsilon})$ 時間でこれを解くアルゴリズムは発見されていない [3]．一方，重みなしグラフの全点対最短経路問題は各頂点から幅優先探索を行うことにより，辺の数を m として $O(mn)$ 時間で解ける [2]．もしグラフが密で，$m = \Omega(n^2)$ ならば，幅優先探索では $O(n^3)$ 時間かかることになる．Seidel のアルゴリズム [4] は，行列積を $O(n^\omega)$ で計算できるとすると，重みなしグラフの全点対最短経路問題を m に依存せず $O(n^\omega \log n)$ 時間で解ける．ここで，$2 < \omega \leq 3$ を満たす行列乗算アルゴリズムがいくつか知られている．

本研究では，Seidel のアルゴリズムにおける行列積計算に Tensor Core を使うことで全点対最短経路問題が高速に解けることを示す．そして，この計算に半精度浮動小数点数を使うことによるオーバーフローの影響を議論する．

†1 現在，大阪府立大学
Presently with Osaka Prefecture Uniersity
a) sab01076@osakafu-u.ac.jp
b) fujimoto@cs.osakafu-u.ac.jp

2. 準備

2.1 GPU

GPU は 2D や 3D などの画像処理を高速に行うために開発された専用のプロセッサである．最近では，GPU の高い処理性能を利用して汎用計算も行う GPGPU（General Purpose computing on GPUs）にも用いられている．GPU 上には複数の SM（ストリーミングマルチプロセッサ）と呼ばれるプロセッサの集合がある．これは演算装置である CUDA コア，データの共有を行うシェアードメモリ，レジスタで構成されている．GPU 上での計算は SM ごとに並列で行われ，また各 SM の中で CUDA コアごとに並列で行われる．SM 内の CUDA コアはシェアードメモリを用いてデータをやり取りすることができる．また，SM 間のデータのやり取りはデバイスメモリ上のグローバルメモリで行われる．グローバルメモリへのアクセスはシェアードメモリやレジスタに比べて速度が落ちる．

2.2 Tensor Core

Tensor Core[1] は 1GPU クロックサイクルで，4×4 行列の積和演算を実行する積和演算器である．半精度浮動小数点数を入力とし，半精度または単精度浮動小数点数で出力する．1 つの丸め演算だけを使用するため，より正確な

出力が得られる．本研究で使用した GPU，NVIDIA A100 は Tensor Core の数が 432 個であり，Tensor 演算のピーク時性能は 312TFLOPS に達する．これは単精度浮動小数点演算性能の約 16 倍である．

2.3 CUDA

CUDA は NVIDIA 社が開発している GPGPU のための統合開発環境である．C 言語をベースにしており，汎用的なプログラミングに適している．BLAS インターフェイス経由でベクトル・行列演算を可能にする CUBLAS などが付属する．CUDA では，並列実行の最小単位をスレッドと呼び，複数のスレッドを管理するブロック，さらに複数のブロックを管理するグリッドと階層的に管理されている．グリッドは 1 ～ 3 次元のブロック，ブロックは 1 ～ 3 次元のスレッドで構成されている．

2.4 半精度浮動小数点数

CUDA で使用できる半精度浮動小数点数のフォーマットは IEEE 754-2008 で規定されており，符号ビットが 1 ビット，指数部が 5 ビット，仮数部が 10 ビットと定められている．指数部がすべて 0 ではない限り，暗黙的に「1」のビットを仮数部に持つので実際には 11 ビットの精度を持つ．半精度浮動小数点数が表現できる最大の数は 65504 であり，これを超える値はオーバーフローが発生する．

2.5 BLAS

BLAS[7] は基本的な線形代数の演算を行うためのライブラリである．ベクトル-ベクトル演算を行う Level 1 BLAS，ベクトル-行列演算を行う Level 2 BLAS，行列-行列演算を行う Level 3 BLAS の 3 つから構成される．CUBLAS は GPU 向きに BLAS を提供したものであり，並列で演算を行えるため高速である．OpenBLAS は BLAS のオープンソース実装であり，様々なプロセッサ向けに最適化を手作業で行っており，GotoBLAS をベースに派生し，今もなお開発が続いている．

2.6 OpenMP

OpenMP[8] は共有メモリ型マシンで並列プログラミングを可能にする API であり，C/C+, FORTRAN から利用できる．ディレクティブを挿入するだけで並列プログラミングを実現できるため，GPGPU などに比べて導入が簡単である．また，OpenMP が使用できない環境ではこのディレクティブは無視されるため，非並列環境への移植が容易であることも利点の 1 つである．

2.7 Seidel のアルゴリズム

Seidel のアルゴリズムは重みなし無向グラフ G の隣接行列（adjacency matrix）A を入力として，G の後続行列（successor matrix）を出力する．ここで G の後続行列 S とは，G の頂点 i から頂点 j への最短経路上で i に続く頂点の番号を s_{ij} とする行列である．Seidel のアルゴリズムの疑似コードを *Algorithm* 1 に示す．*Algorithm* 1 は，まず G の距離行列（distance matrix）D を計算し，次に D と A から G の後続行列を計算する．ここで G の距離行列 D とは，G の頂点 i から頂点 j への最短経路上にある辺の数を d_{ij} とする行列である．G の隣接行列を入力として，G の距離行列を計算するアルゴリズムの疑似コードを *Algorithm* 2 に示す．*Algorithm* 2 は，D と A から G の後続行列を計算する際に，乱択アルゴリズムである *Algorithm* 3 をサブルーチンとして用いる．

Algorithm 1 APSP[4]

Input: the $n \times n$ 0-1 adjacency matrix A of an undirected, unweighted, connected graph G with n vertices
Output: the $n \times n$ shortest path successor matrix S of G

let $D := \mathbf{APD}(A)$
for each $r = 0, 1, 2$ **do**
  let $D^{(r)}$ be the $n \times n$ 0-1 matrix with $d_{ij}^{(r)} = 1$ iff $d_{ij} + 1 \mod 3 = r$
  let $W^{(r)} := \mathbf{BPWM}(A, D^{(r)})$
end for
return $n \times n$ matrix S, where $s_{ij} = w_{ij}^{(\rho)}$, with $\rho = d_{ij} \mod 3$

Algorithm 2 APD[4]

Input: the $n \times n$ 0-1 adjacency matrix A of an undirected, unweighted, connected graph G with n vertices
Output: the $n \times n$ distance matrix D of G

let $Z = A \cdot A$
let B be an $n \times n$ 0-1 matrix, where $b_{ij} = 1$ iff $i \neq j$ **and** ($a_{ij} = 1$ **or** $z_{ij} > 0$)
if $b_{ij} = 1$ for all $i \neq j$ **then**
  return $n \times n$ matrix $D = 2B - A$
end if
let $T = \mathbf{APD}(B)$
let $X = T \cdot A$
return $n \times n$ matrix D, where

$$d_{ij} = \begin{cases} 2t_{ij} & (x_{ij} \geq t_{ij} \times degree(j)) \\ 2t_{ij} - 1 & (x_{ij} < t_{ij} \times degree(j)) \end{cases}$$

2.8 Strassen のアルゴリズム

Strassen のアルゴリズムは行列積演算を高速に計算するアルゴリズムである．$n \times n$ 行列同士の積を計算するためには通常,$O(n^3)$ 時間かかるが，Strassen のアルゴリズムではこの計算を $O(n^{\log_2 7}) \approx O(n^{2.807})$ 時間で行うことができる．

簡単のため n を偶数として，行列を次のように $\frac{n}{2} \times \frac{n}{2}$ の部分行列に分割する．

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

そして，分割した部分行列に対して次の行列を計算する．

$$P_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$
$$P_2 = (A_{21} + A_{22}) \cdot B_{11}$$
$$P_3 = A_{11} \cdot (B_{12} - B_{22})$$
$$P_4 = A_{22} \cdot (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12}) \cdot B_{22}$$
$$P_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

このとき，

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 + P_3 - P_2 + P_6$$

が成り立つ．通常では行列積を求めるときに部分行列の乗算が 8 回必要であるのに対して，Strassen のアルゴリズムでは 7 回の乗算と 18 回の加算で行列積を求めることができる．したがって，行列の乗算を再帰的に Strassen のアルゴリズムを使うことにより計算量を減らすことができる．

Algorithm 3 BPWM[4]

Input: $n \times n$ 0-1 matrices A, B
Output: the $n \times n$ witness matrix W for A and B

```
let W = −A · B
for each d = 2^l, where l = 0, ..., ⌈log_2 n⌉ − 1 repeat ⌈3.42 log_2 n⌉ times do
    choose d independent random numbers {k_1, k_2, ..., k_d}, drawn uniformly from {1, ..., n}
    let X be an n × d matrix with columns k_i a_{*k_i} and Y a d × n matrix with rows b_{k_i *} (1 ≤ i ≤ d)
    let C = X · Y
    for each (i, j) s.t. w_ij < 0 and c_ij is a witness for (i, j) do
        w_ij := c_ij
    end for
end for
for each (i, j) s.t. w_ij < 0 do
    w_ij := some witness k for (i, j), found by trying each k
end for
return W
```

2.9 大塚らの Strassn アルゴリズムの最適化

大塚ら [11][12] は，GPU に適した Strassen アルゴリズムの最適化を提案している．部分行列への分割を 1 回だけ行い，分割した部分行列の 7 回の乗算では分割を行わない Strassen アルゴリズムを 1-level Strassen アルゴリズムと呼ぶ．分割した部分行列の乗算に再帰的に深さ $d(\geq 2)$ まで分割を行う Strassen アルゴリズムを d-level Strassen アルゴリズムと呼ぶ．大塚らは，偶数 n に対して $n \times n$ 正方行列を対象とし，行列の乗算に CUBLAS-10.1 を使って，倍精度計算のみを対象とした 2-levelStrassen アルゴリズムの最適化を行っている．彼らは通常の Strassen アルゴリズムでは計算結果を格納するために必要な temporary 行列を削除することにより高速化した 1-level Strassen アルゴリズムを lower level の行列乗算に用い，これを upper level から呼び出すことにより高速な 2-level Strassen アルゴリズムを実現した．この 1-level Strassen アルゴリズムの疑似コードを *Algorithm* 4 に示す．*Algorithm* 4 では，行列 A, B の一部を計算結果を格納するために使用している．そのため，*Algorithm* 4 を再帰的に呼び出して Strassen アルゴリズムを構成することはできない．

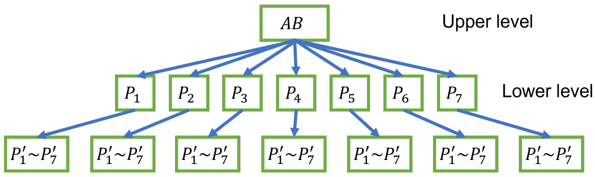

図 1　大塚らの 2-level Strassen アルゴリズムの模式図

大塚らの 2-level Strassen アルゴリズムの模式図を**図 1** に示す．A, B は $n \times n$ 行列であり，$P_1 \sim P_7$ は行列積 $A \cdot B$ に Strassen アルゴリズムを適用したときの 7 回の行列積を表す．

Algorithm 4 temporary 行列が 0 個の 1-level Strassen アルゴリズム 1L_Strassen[11][12]

Input: $n \times n$ 0-1 matrices A, B
Output: the $n \times n$ matrix C

```
C22 = A11 − A21
C11 = B22 − B12
A21 = A21 + A22
C12 = B12 − B11
C21 = C22 · C11
C22 = A21 · C12
A21 = A21 − A11
B12 = B22 − C12
C12 = A12 − A21
C11 = A21 · B12
A21 = C12 · B22
C12 = C22 + A21
B22 = A11 · B11
C11 = B22 + C11
C12 = C11 + C12
C11 = C11 + C21
B11 = B12 − B21
C21 = A22 · B11
A11 = A12 · B21
C21 = C11 − C21
C22 = C11 + C22
C11 = B22 + A11
```

3. 提案手法とその実装

Rezaul らの研究 [10] では，NVIDIA の Tensor Core や Google の Tensor processing unit といった機械学習に特化

したプロセッサを使って行列積演算を高速化する，Strassen-like アルゴリズムを提案している．このアルゴリズムでは通常の Strassen アルゴリズムと同様に，再帰的に行列積を計算していくが，サブ問題がこれらのプロセッサが行える行列積の最小単位（Tensor Core なら 16×16 行列）に達したところで再帰を終了する．さらに，Rezaul らはこの Strassen-like アルゴリズムを密な重みなし無向グラフの全点対最短経路問題を解く Seidel のアルゴリズム [4] 内の行列積演算に使う手法を提案している．

上記の Strassen-like アルゴリズムを Tensor Core を用いて実装しようと試みたところ，Tensor Core の行列積の最小単位を再帰のベースケースに設定するとオーバーヘッドが増加し，予想に反して実行時間が遅くなってしまった．そこで本研究では，大塚らの 2-level Strassen アルゴリズムを半精度浮動小数点数で Tensor Core を用いて実装し，これを Seidel のアルゴリズムの行列積に使用する手法を提案する．大塚らの論文では upper level では temporary 行列を 7 個用いていたが，本研究では temporary 行列が 4 個の演算スケジュールを考案した．このアルゴリズムを *Algorithm* 5 に示す．

Algorithm 5 の lower level の計算には *Algorithm* 4 を使っている．lower level の行列積に Tensor Core を使えるように変更したものを使って Seidel のアルゴリズムを実装する．提案手法のアルゴリズムを *Algorithm* 6, *Algorithm* 7, *Algorithm* 8 に示す．

Algorithm 5 temporary 行列が 4 個の 2-level Strassen アルゴリズム 2L_Strassen

Input: $n \times n$ 0-1 matrices A, B
Output: the $n \times n$ matrix C

$C_{22} = A_{11} - A_{21}$
$C_{11} = B_{22} - B_{12}$
$T_1 = A_{21} + A_{22}$
$C_{12} = B_{12} - B_{11}$
$T_2 = T_1 - A_{11}$
$T_3 = C_{11} + B_{11}$
$T_4 = A_{12} - T_2$
$C_{21} =$ **1L_ Strassen**(C_{22}, C_{11})
$C_{21} =$ **1L_Strassen**(T_1, C_{12})
$T_1 = T_3 - B_{21}$
$C_{11} =$ **1L_Strassen**(T_2, T_3)
$T_2 =$ **1L_Strassen**(T_4, B_{22})
$C_{12} = C_{22} + T_2$
$A_{21} =$ **1L_Strassen**(A_{11}, B_{11})
$C_{11} = C_{11} + A_{21}$
$C_{12} = C_{12} + C_{11}$
$C_{11} = C_{11} + C_{21}$
$T_3 =$ **1L_Strassen**(A_{22}, T_1)
$A_{11} =$ **1L_Strassen**(A_{12}, B_{21})
$C_{21} = C_{11} - T_3$
$C_{22} = C_{11} + C_{22}$
$C_{11} = A_{21} + A_{11}$

Algorithm 6 APSP の GPU 実装

Input: the $n \times n$ 0-1 adjacency matrix A of an undirected, unweighted, connected graph G with n vertices
Output: the $n \times n$ shortest path successor matrix S of G

let $D :=$ **APD_Strassen**(A)
for each $r = 0, 1, 2$ **do**
  let $D^{(r)}$ be the $n \times n$ 0-1 matrix
    with $d_{ij}^{(r)} = 1$ iff $d_{ij} + 1 \mod 3 = r$
  let $W^{(r)} :=$ **BPWM_parallel**$(A, D^{(r)})$
end for
return $n \times n$ matrix S,
  where $s_{ij} = w_{ij}^{(\rho)}$, with $\rho = d_{ij} \mod 3$

Algorithm 7 APD_Strassen

Input: the $n \times n$ 0-1 adjacency matrix A of an undirected, unweighted, connected graph G with n vertices
Output: the $n \times n$ distance matrix D of G

let $A_1, A_2, A_3 = A$
let $Z =$ **2L_Strassen**(A_1, A_2)
let B be an $n \times n$ 0-1 matrix, where
  $b_{ij} = 1$ iff $i \neq j$ **and** $(a_{ij} = 1$ **or** $z_{ij} > 0)$
if $b_{ij} = 1$ for all $i \neq j$ **then**
  return $n \times n$ matrix $D = 2B - A$
end if
let $T =$ **APD_Strassen**(B)
let $T_1 = T$
let $X =$ **2L_Strassen**(T_1, A_3)
return $n \times n$ matrix D, where

$$d_{ij} = \begin{cases} 2t_{ij} & (x_{ij} \geq t_{ij} \times degree(j)) \\ 2t_{ij} - 1 & (x_{ij} < t_{ij} \times degree(j)) \end{cases}$$

Algorithm 8 BPWM_parallel

Input: $n \times n$ 0-1 matrices A, B
Output: the $n \times n$ witness matrix W for A and B

let $A_1 = A$
let $B_1 = B$
let $W = -$**2L_Strassen**(A_1, B_1)
parallel for each (i, j) s.t. $w_{ij} < 0$ **do**
  for $k = 1$ to n **do**
    if $a_{ik} = 1$ **and** $b_{kj} = 1$ **then**
      $w_{ij} := k$
      break
    end if
  end for
end parallel for
return W

Argorithm 3 は乱択アルゴリズムである．2 行目からの for ループで乱数を用いて大部分の witness を見つけ，10 行目からの for ループで見つけきれなかった witness を補完する構造になっている．2 行目からの for ループは $O(\log_2^2 n)$ 回繰り返しており，5 行目で $O(n^\omega)$ 時間の行列積を実行できるとすれば，*Argorithm* 3 の期待実行時間は $\omega = 2$ なら $O(n^2 \log^2 n)$ 時間，$\omega > 2$ なら $O(n^\omega \log n)$ 時間となる [4]．しかし *Argorithm* 3 を素直に実装したところ，乱数生成に時間がかかり，BPWM の実行時間が APD の実行時間に比べて非常に低速な実装となってしまった．そこで 2 行目からの for ループを削除し，10 行目からの for ループを並列に実装するほうが witness を高速に見つけることができるのではと考えた．このアルゴリズムを BPWM_parallel と名づけ，*Argorithm* 8 に示す．このアルゴリズムは *Argorithm* 3 とは異なり，乱択アルゴリズムとなっていない．7 行目からの for ループにおいて，密なグラフでは k が小さいときに winess が見つかるので，*Argorithm* 8 の計算量は $O(n^2)$ に近づくと予想される．

4. 評価実験

4.1 実験環境

本実験に使用した GPU は NVIDIA A100 PCle (6912 コア) であり，CPU は AMD Ryzen 5800X (8 物理コア，16 論理コア) である．また，OS に Windows 10 Pro，コンパイラに Visual Studio 2019 Community を使用した．

4.2 半精度浮動小数点数の使用による影響

Algorithm 7 の 2, 9 行目と *Algorithm* 8 の 3 行目の行列積を半精度浮動小数点数で計算する場合を考える．A はグラフの隣接行列なので，その要素は 0 か 1 である．T はある時点での距離行列の計算結果が格納されているので，その要素は 0 から n の整数である．このように Seidel のアルゴリズムではどちらも非負整数を要素とする行列の行列積を対象としている．仮に A の 1 行目が対角成分以外すべて 1 であるとすると，A は隣接行列なので 1 列目も対角成分以外はすべて 1 である．$n = 22528$ のとき，A^2 の 1 行 1 列目は 22527 と計算される．これは半精度浮動小数点数が表現できる最大値の 65504 以下であるので，仮数部で丸め誤差が発生し精度が低下する可能性はあるが，オーバーフローは発生しない．今回実験した中で最多の頂点数 22528 の場合，2 行目の行列積では丸め誤差が確認された．しかし，Z で丸め誤差が発生して精度が低下したとしてもオーバーフローが起きなければ 2 行目で $z_{ij} > 0$ が判断でき，アルゴリズム的に問題ない．したがって，1 行目の精度低下の影響は最終的な D の計算に影響しない．一方，9 行目の行列積では次数を変えて実験したがオーバーフローは確認できなかった．グラフが密になれば，その直径は小さくなるので *Algorithm* 7 はほとんど再帰がかからない．そのため 7 行目の T の要素全体が小さな値で構成され，オーバーフローは起こりづらくなると考えられる．また，グラフが疎になれば再帰はかかるが A が疎になるのでオーバーフローは起こりづらくなると考えられる．*Algorithm* 8 の 3 行目では先ほどと同様に精度低下が起きても，$w_{ij} < 0$ が判断できれば問題ない．したがって，3 行目の精度低下の影響は最終的な S の計算に影響しない．なお，オーバーフローの発生の有無は単精度浮動小数点数で実装したものと半精度浮動小数点数で実装したものの計算値を比較することによって確認した．

4.3 GPU 実装と CPU 実装の比較

まず最初に Strassen を使わず Tensor Core だけを使った GPU 実装（CUBLAS-11.4）と，OpenBLAS（0.3.18）と OpenMP を使った CPU 実装の実行時間を比較する．CPU 実装では *Algorithm* 7 の行列積に OpenBLAS を，*Algorithm* 8 に OpenMP を使って並列に実装した．なお，入力データは頂点数と次数を指定してランダムなグラフを生成するプログラム [9] を一部改変し，異なるシード値で生成できるようにしたものを使い作成した．頂点数が n，次数が d の 10 個のランダムなグラフを生成し，それぞれついて 5 回実験を行った．実験結果を**表 1**，**表 2** に示す．頂点数 16384，次数 4096 の時，Tensor Core だけを使った GPU 実装が OpenBLAS と OpenMP を使った CPU 実装より，約 71.1 倍高速であった．

4.4 APD と BPWM の実行時間の内訳

次に Strassen を使わず Tensor Core だけを使った GPU 実装と，OpenBLAS と OpenMP を使った CPU 実装の実行時間の内訳を比較する．実験結果は**表 3**，**表 4** のようになった．表 3，表 4 から見てわかるように，次数が増え，グラフが密になると GPU 実装では BPWM の実行時間は

表 1 頂点数 n，次数 2048 のグラフにおける GPU 実装と CPU 実装の平均実行時間 (s) と GPU 実装による speed up(倍).

n	GPU(Tensor Core)	CPU(OpenBLAS & OpenMP)	speed up(倍)
4096	0.1065236	6.23126182	58.5
8192	0.88282632	33.63225706	38.1
10240	1.52908622	60.4782364	39.6
12288	2.47859816	101.7024635	41.0
14336	3.78253686	170.7875935	45.2
16384	5.61231542	286.0589882	51.0

表 2 頂点数 n，次数 4096 のグラフにおける GPU 実装と CPU 実装の平均実行時間 (s) と GPU 実装による speed up(倍).

n	GPU(Tensor Core)	CPU(OpenBLAS & OpenMP)	speed up(倍)
8192	0.89063726	53.78678668	60.4
10240	1.50524972	94.43262222	62.7
12288	2.46025284	155.9162419	63.4
14336	3.69756672	245.8830974	66.5
16384	5.45570802	387.9040621	71.1

表 3 頂点数 n，次数 2048 のグラフにおける GPU 実装と CPU 実装の平均実行時間 (s) の内訳.

n	GPU(Tensor Core)		CPU(OpenBLAS & OpenMP)	
	APD	BPWM	APD	BPWM
4096	0.0027039	0.10287718	1.0914514	5.01799232
8192	0.03053376	0.84610626	5.48548858	27.66128812
10240	0.06255288	1.45615064	9.18989072	50.51734082
12288	0.10191718	2.36255094	14.5395526	86.0593973
14336	0.11643016	3.65293724	21.28332798	147.9607886
16384	0.13398636	5.4631666	30.61324868	253.2057436

表 4 頂点数 n，次数 4096 のグラフにおける GPU 実装と CPU 実装の平均実行時間 (s) の内訳.

n	GPU(Tensor Core)		CPU(OpenBLAS & OpenMP)	
	APD	BPWM	APD	BPWM
8192	0.034532	0.8491555	5.60235008	47.65990532
10240	0.06266272	1.43232762	9.4895087	84.12572268
12288	0.09545854	2.3520406	14.97485902	139.7598095
14336	0.11504286	3.57016298	21.9547185	222.2897872
16384	0.1365232	5.30421072	30.87711842	354.8143632

ほとんど変化しないが，CPU 実装では明らかに増加している．また，頂点数 16384，次数 2048 の時，APD の GPU 実装が CPU 実装の約 228 倍高速であった.

4.5 Tensor Core の効果

この節では Tensor Core の効果を確認するため APD の実行時間について，Strassen を使わず Tensor Core だけを使った APD の GPU 実装（CUBLAS-11.3）と Strassen も Tensor Core も使わない単精度浮動小数点数での APD の GPU 実装（CUBLAS-11.3）を比較する．なお，この実験では異なる頂点数，次数毎に 1 つのランダムに生成されたグラフを使って，それぞれに 10 回実験を行ったものの平均を取っている．実験結果を**表 5**，**表 6** に示す．表 5，表 6 から分かるように Tensor Core を使用することによって，APD 単体で見ると 3.94 ∼ 14.4 倍高速であった.

4.6 2-level Strassen アルゴリズムの効果

提案する 2-level Strassen アルゴリズムの効果を確認するため，Tensor Core を用いた行列積の実行時間を CUBLAS と比較する．頂点数が n，次数が 4096 の 1 個のランダムなグラフを生成し，その隣接行列 A を入力として $A \cdot A$ を

表 5 頂点数 n，次数 2048 のグラフにおける Tensor Core の有無による APD の平均実行時間 (s) とその speed up(倍).

n	GPU(Tensor Core)	GPU(Without Tensor Core)	speed up(倍)
4096	0.0034615	0.049717	14.4
8192	0.0342479	0.2668134	7.79
10240	0.0632636	0.3451847	5.46
12288	0.1013165	0.4734058	4.67
14336	0.1315237	0.5834482	4.44
16384	0.1735481	0.6830961	3.94
18432	0.2003403	0.8743424	4.36
20480	0.1945336	1.1126282	5.72

表 6 頂点数 n，次数 4096 のグラフにおける Tensor Core の有無による APD の平均実行時間 (s) とその speed up(倍).

n	GPU(Tensor Core)	GPU(Without Tensor Core)	speed up(倍)
8192	0.0347021	0.2644771	7.62
10240	0.0630311	0.3725535	5.91
12288	0.1045994	0.4673956	4.47
14336	0.1307274	0.5881268	4.50
16384	0.1593589	0.7096246	4.45
18432	0.1900153	0.9025731	4.75
20480	0.1664439	1.1244147	6.76
22528	0.2206297	1.3847672	6.28

計算し，それぞれについて 5 回実験を行った．実験結果を**表 7** に示す．頂点数が 18432 の時に CUBLAS に比べて約 1.04 倍高速であった.

4.7 Strassen 利用の有無による GPU 実装の比較

次に提案する 2-level Strassen が Seidel のアルゴリズムの高速化に効果があるのかを確認するため，提案する 2-level Strassen と Tensor Core を使った GPU 実装（CUBLAS-11.4）と Tensor Core だけを使った GPU 実装（CUBLAS-11.4）の実行時間を比較する．実験結果を**表 8**，**表 9** に示す．表 8，表 9 を見てわかるように，Strassen と Tensor Core を用いた実装は単純に Tensor Core のみを用いた実装に比べて，ほとんどの場合で大幅な改善は見られなかった.

4.8 考察

Tensor Core を使った GPU 実装と CPU 実装の比較では，表 1，表 2 から見てわかるように頂点数 n が増加すると，概ね speed up の倍率が上昇している．この結果から，頂点数を増加させてもこの傾向は続くと予想される．さらに，APD と BPWM の実行時間に数十倍の差がでた原因としては，十分に密なグラフを対象としているため，APD の再帰がほとんど発生しないことが考えられる.

APD における Tensor Core の使用による効果は，理論値である単精度浮動小数の演算性能の約 16 倍には達しなかったものの，最大で約 14.4 倍の高速化を実現した．表 5，表 6 を見ると，頂点数が小さい場合に speed up の倍率が高く，$n = 16384$ まで大きくなるにつれ倍率が下がっている．さらにそこから徐々に倍率が上昇していることが分かる．Tensor Core を使用した APD の GPU 実装では，単

表 7 行列の 2 乗を提案する 2-level Strassen と CUBLAS で計算したときの平均実行時間 (s) とその speed up(倍).

n	2-level	CUBLAS	speed up(倍)
8192	0.0406098	0.0246366	0.61
10240	0.0638364	0.0481676	0.75
12288	0.0994732	0.0829214	0.83
14336	0.1371554	0.1124802	0.82
16384	0.1751772	0.1489526	0.85
18432	0.1751602	0.1828742	1.04
20480	0.2027922	0.2090728	1.03
22528	0.2258206	0.232849	1.03

表 8 頂点数 n，次数 2048 のグラフにおける Strassen の有無による平均実行時間 (s) と speed up(倍).

n	GPU (Strassen & Tensor Core)	GPU (Tensor Core)	speed up(倍) TC/Str&TC
4096	0.10925468	0.1065236	0.975
8192	0.91653294	0.88282632	0.963
10240	1.51444482	1.52908622	1.01
12288	2.49263636	2.47859816	0.994
14336	3.80780788	3.78253686	0.993
16384	5.64582028	5.61231542	0.994
18432	7.945251	7.90438486	0.995
20480	10.85443752	10.8181682	0.997

表 9 頂点数 n，次数 4096 のグラフにおける GPU 実装の平均実行時間 (s).

n	GPU (Strassen & Tensor Core)	GPU (Tensor Core)	speed up(倍) TC/Str&TC
8192	0.9196334	0.89063726	0.968
10240	1.51022966	1.50524972	0.997
12288	2.48737616	2.46025284	0.989
14336	3.73549818	3.69756672	0.990
16384	5.48519192	5.45570802	0.995
18432	7.68496692	7.66610824	0.998
20480	10.48763122	10.4399349	0.995
22528	14.00993274	13.95168328	0.996

精度浮動小数から半精度浮動小数点数に変換する処理や，半精度浮動小数点数のメモリを確保・解放する処理がある．n が小さい時には行列積部分に比べてこの処理にかかる時間の割合が低く，そこから徐々に高くなり，$n = 16384$ をピークにまた割合が下がっている可能性がこのような結果になる原因として考えられる．

一方，提案する 2-level Strassen では表 7 のように，$n = 18432$ から CUBLAS で単純に Tensor Core を呼び出したものより約 1.04 倍高速になったものの，Seildel のアルゴリズムに組み込むと全体の高速化に寄与しなかった．2-level Strassen アルゴリズムでは入力行列を破壊するため，*Algorithm* 7 や *Algorithm* 8 では入力行列を退避させておく必要がある．このとき，メモリの確保や解放，コピーに余分な時間が発生したことが原因として考えられる．

5. まとめと今後の課題

本研究では，Seidel のアルゴリズム内の行列積に Tensor Core を用いることにより高速化を実現した．また，Seidel のアルゴリズムにおいて半精度を使うことによる精度低下とオーバーフローの影響は比較的小さいと予想され，頂点数 $n = 22528$，次数 $4, 5, 6, 7, 8, 16, 32, 64, 128, 256, 512, 1024, 2048$ のランダムグラフでこの予想が正しい可能性が高いことを実験的に確認した．今後の課題として，精度低下とオーバーフローが影響するグラフの条件を明確にしたい．

一方，BPWM 部分が最悪 $O(n^3)$ 時間かかる実装となってしまい，全体の時間計算量が $O(n^3 \log n)$ 時間になってしまった．BPWM 部分において Seidel のオリジナルの乱択アルゴリズムの並列化を考案し実装することでより高速な Seidel のアルゴリズムを実現したい．

本研究で提案する Strassen アルゴリズムの実装は，行列乗算の高速化に成功したものの，Seidel のアルゴリズムに十分に生かすことができなかった．入力行列を破壊しない高速な Strassen アルゴリズムの構築，もしくはメモリの退避を含めても高速な Strassen アルゴリズムを考案したい．

また，小さな整数の重みをもつ無向グラフの全点対最短経路問題を解くアルゴリズムが Galil ら [14] によって示されている．このアルゴリズムも内部で行列積を使用しているので，Tensor Core を使うことによって高速化を実現できる可能性がある．

参考文献

[1] NVIDIA：NVIDA A100 Tensor コア GPU アーキテクチャ，`https://www.nvidia.com/content/dam/en-zz/ja/Solutions/Data-Center/documents/nvidia-ampere-architecture-whitepaper-jp.pdf`
[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C.：Introduction to Algorithms, third edition, MIT Press (2009)
[3] Pettie, S.：All Pairs Shortest Paths in Sparse Graphs, Encyclopedia of Algorithms, Springer (2016)
[4] Seidel, R.：On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs, Journal of Computer and System Sciences, Vol.51, pp.400–403 (1995)
[5] Strassen, V.：Gaussian Elimination Is Not Optimal, Numerische Mathematik, Vol.13, pp.354–356 (1969)
[6] NVIDIA：CUDA Programming Guide, `https://www.nvidia.co.jp/docs/IO/51174/NVIDIA_CUDA_Programming_Guide_1.1_JPN.pdf`
[7] BLAS (Basic Linear Algebra Subprograms), `http://www.netlib.org/blas/`
[8] OpenMP：The OpenMP API Specification for Parallel Programming, `https://www.openmp.org/`
[9] 藤原一毅：create-random.py, `http://research.nii.ac.jp/graphgolf/problem.html`
[10] Rezaul, C., Francesco, S., and Flavio, V.：A Computational Model for Tensor Core Units, ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp.519–521 (2020)
[11] 大塚 達史, 井口 寧：GPU 向き Strassen アルゴリズムの最適化, 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol.2020-HPC-173, No.21, pp.1–8 (2020)
[12] 大塚 達史：GPU 向き Strassen アルゴリズムの最適化, 北陸先端科学技術大学院大学修士学位論文, `http://hdl.handle.net/10119/16385` (2020)
[13] Lai, P., Arafat, H., Elango, V., and Sadayappan, P.：Accelerating Strassen-Winograd's Matrix Multiplication Algorithm on GPUs, International Conference on High Performance Computing (HiPC), pp. 139–148 (2013)
[14] Galil, Z. and Margalit, O.：All Pairs Shortest Paths for Graphs with Small Integer Length Edges. Journal of Computer and System Sciences. Vol.54, No.2, pp.243–254 (1997)