Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 8, Number 2, pages 143–147, July 2019

Accelerating GPU implementation of All Pairs Shortest Path Algorithm

Daisuke Takafuji, Koji Nakano and Yasuaki Ito
Graduate School of Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima City, 739-8527, Japan
{daisuke,nakano,yasuaki}@cs.hiroshima-u.ac.jp

***Abstract*—The Floyd-Warshall algorithm is one of fundamental algorithms, and is useful for many application. Blocked Floyd-Warshall algorithm is adapted to parallel processing more than the Floyd-Warshall algorithm. Its GPU implementations are proposed, and consists of three CUDA kernel. We propose a new implementation with single CUDA kernel. Our implementation achieved a speedup factor of $1.02-1.15$ on NVIDIA Tesla V100. We also propose GPU implementations for bulk computation, which are extended versions of single or multiple kernel implementation.**

I. はじめに

Floyd-Warshall algorithm はグラフの全点対間の最短パスを求める多項式時間アルゴリズムである．グラフが n 個の頂点を持つとき，その計算量は $O(n^3)$ である．このアルゴリズムは基本アルゴリズムの一つであり，幅広い応用がある．たとえば [1] では，グラフの最短パス長の平均値 (ASPL) や直径に着目してグラフ構造が研究されているが，ASPL や直径を計算する際にこのアルゴリズムが有効であるが，計算量が非常に大きい．[2] ではこのアルゴリズムを改良した Blocked Floyd-Warshall algorithm が提案されている．これは並列計算に適したものであり，GPU 実装が提案されており，その性能が評価されている [3]–[5].

既存の Blocked Floyd-Warshall アルゴリズムの GPU 実装 [3]–[5] は 3 つの CUDA kernel で構成されており，効率的なメモリアクセスを実現し高速化している．CUDA kernel 呼び出しはオーバヘッドを引き起こし，多くの CUDA kernel 呼び出しはパフォーマンスの低下を招く．我々は 1 つの CUDA kernel による GPU 実装を提案し，NVIDIA Tesla V100 において既存実装に比べて 1.02–1.15 倍の高速化を実現した．既存の GPU 実装を**複数 *kernel* 実装**，**シングル *kernel* 実装**と呼ぶ．また，複数インスタンスが与えられた場合へ拡張した GPU 実装も提案し，その性能を評価した．

GPU (Graphics Processing Unit) は，画像処理計算の高速化を目的に設計された専用回路である [6]．最近の GPU は汎用計算のために設計され従来の CPU 計算にも応用でき，多くのアプリケーション開発者に注目されている [6]．NVIDIA は，自社が提供する GPU を動作させるための並列計算機アーキテクチャを提供しており，これを *CUDA* (Compute Unified Device Architecture) [7] という．CUDA は，NVIDIA 社 GPU での仮想命令セットと並列計算のメモリを提供する．

II. Floyd-Warshall アルゴリズム

点集合 V，辺集合 E からなる有向グラフ $G=(V,E)$ と距離 $D : V \times V \to R$ （非負実数）が与えられたとき，Floyd-Warshall algorithm は全点対間の最短パスを求めるアルゴリズムである．以下にアルゴリズムを記す．点数を n とすると，この計算量は $O(n^3)$ である．任意の対 $u, v \in V$ に対し，u から v への距離 $D(u,v)$ を $D[u][v]$ と記す．アルゴリズムが終了後，$D[u][v]$ $(u,v \in V)$ は u から v への最短パス長を表しており，もし $D[u][v] \neq \infty$ ならば u から v へのパスは存在しない．

Algorithm 1 Floyd-Warshall アルゴリズム

```
1: for k ← 0 to n do
2:     for i ← 0 to n do
3:         for j ← 0 to n do
4:             if D[i][j] > D[i][k] + D[k][j] then
5:                 D[i][j] ← D[i][k] + D[k][j]
6:             end if
7:         end for
8:     end for
9: end for
```

Blocked Floyd-Warshall アルゴリズム [2] は Floyd-Warshall アルゴリズムの拡張版である．有向グラフ $G=(V,E)$ と距離 $D : V \times V \to R$ 非負整数 W を入力として，Floyd-Warshall アルゴリズムと似たように動作する．サイズが $W \times W$ の領域（タイルと呼ぶ）に距離 D を分割する．タイルは $n/W \times n/W$ 個である．I 行 J 列のタイルを $T_{I,J}$ と表す．Blocked Floyd-Warshall アルゴリズムを Algorithm 2 に記す．

$Z = z_1$ と仮定する．$I = J = z_1$ のときタイル $T_{I,J}$ を *pivot tile* とよぶ．$I \neq z_1$（$J \neq z_1$）なるタイル T_{I,z_1} （$T_{z_1,J}$）を *column pivot tile*（*row pivot tile*）とよぶ．$I \neq z_1$ かつ $J \neq z_1$ なるタイル $T_{I,J}$ を *non-pivot tile* とよぶ．$T_{0,0}$ は $Z = 0$ のとき pivot tile であるが，$Z = 1$ のとき non-pivot tile である．$n = 12$, $W = 3$ の場合の Blocked Floyd-Warshall アルゴリズムのおおまかな実行の様子を Fig. 1 に記す．

Column pivot tile T_{I,z_1} の各 distance は pivot tile T_{z_1,z_1} を使って Function F($T_{I,z_1}, T_{I,z_1}, T_{z_1,z_1}$) を実行することにより計算される．同様に，Row pivot tile $T_{z_1,J}$ の各 distance は pivot tile T_{z_1,z_1} を使って Function F($T_{z_1,J}, T_{z_1,J}, T_{z_1,z_1}$) を実行することにより計算される．また non-pivot tile $T_{I,J}$ の各 distance は Column pivot tile T_{I,z_1} と Row pivot tile $T_{z_1,J}$ を使って Function F($T_{I,J}, T_{I,z_1}, T_{z_1,J}$) を実行することにより計算される．

Blocked Floyd-Warshall アルゴリズムでは，pivot tile は他のタイルに依存することなく計算することができる．column pivot tile $T_{z_1,J}$ や row pivot tile T_{I,z_1} は pivot tile T_{z_1,z_1} の計

Algorithm 2 Blocked Floyd-Warshall アルゴリズム

for $Z \leftarrow 0$ to $n/W-1$ **do**
  F($T_{Z,Z}, T_{Z,Z}, T_{Z,Z}$) ▷ Computation of a pivot tile $T_{Z,Z}$
  for $X \leftarrow 0$ to $n/W-1$ and $X \neq Z$ **do**
    F($T_{X,Z}, T_{X,Z}, T_{Z,Z}$) ▷ Computation of a column pivot tile $T_{X,Z}$
  end for
  for $Y \leftarrow 0$ to $n/W-1$ and $Y \neq Z$ **do**
    F($T_{Z,Y}, T_{Z,Y}, T_{Z,Z}$) ▷ Computation of a row pivot tile $T_{Z,Y}$
  end for
  for $X \leftarrow 0$ to $n/W-1$ and $X \neq Z$ **do**
    for $Y \leftarrow 0$ to $n/W-1$ and $Y \neq Z$ **do**
      F($T_{X,Y}, T_{X,Z}, T_{Z,Y}$) ▷ Computation of a non-pivot tile $T_{X,Y}$
    end for
  end for
end for
function F(A, B, C) ▷ /* Let A, B and C be a tile of size $W \times W$.*/
  for $k \leftarrow 0$ to $W-1$ **do**
    for $i \leftarrow 0$ to $W-1$ **do**
      for $j \leftarrow 0$ to $W-1$ **do**
        if $A[i][j] > B[i][k] + C[k][j]$ **then**
          $A[i][j] \leftarrow B[i][k] + C[k][j]$
        end if
      end for
    end for
  end for
end function

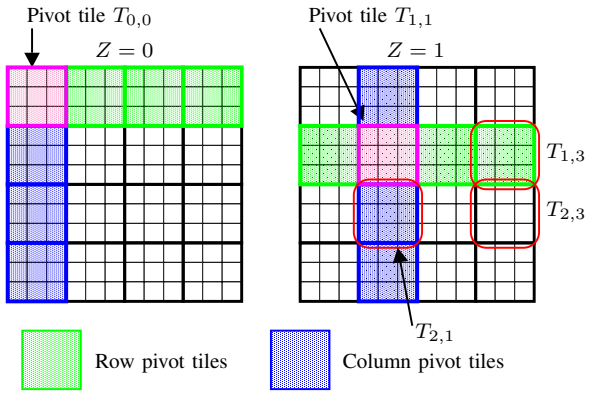

Fig. 1. $n = 12$, $W = 3$ の場合の Blocked Floyd-Warshall アルゴリズムの流れ

算が終了した後に計算を開始できる．同様に non-pivot tile $T_{I,J}$ は column pivot tile $T_{z_1,J}$ と row pivot tile T_{I,z_1} の計算が終了した後に計算を開始できる．

$n = 12$, $W = 3$, $Z = 1$ の Blocked Floyd-Warshall アルゴリズムの例を Fig. 1 に示す．$T_{1,1}$ は pivot tile なので，独立に計算できる．$T_{1,3}$ または $T_{2,1}$ は column pivot tile または row pivot tile なので，$T_{1,1}$ の計算の終了後に計算を開始できる．$T_{2,3}$ は non-pivot tile なので，$T_{1,3}$ も $T_{2,1}$ もともに計算が終了した後に計算を開始できる．つまり，$T_{1,3}$, $T_{2,1}$, $T_{1,1}$ のいずれかのタイルの計算が終わる前には $T_{2,3}$ の計算は開始できない．

任意の 3 つのタイルにおいて A, B, C, Function F(A, B, C) は各 k ($0 \leq k < W$) において $A[i][j] \leftarrow \min\{A[i][j], B[i][k]+C[k][j]\}$ を計算することで距離 $A[i][j]$ を求める．

III. 複数 KERNEL 実装

まず既存実装 [5] を紹介する．既存の GPU 実装は，次の 3 つの kernel を用意し，各 Z ($0 < Z < n/W$) においてこれらを実行するものである．(1) pivot tile を計算する kernel, (2) row pivot tile と column pivot tile を計算する kernel, (3) non-pivot tile を計算する kernel

まず pivot tile $T_{Z,Z}$ を計算する kernel では $T_{Z,Z}$ を shared memory にコピーする．1 thread が 1 つの distance を計算することで，function F() を実行する．各 Z, $0 \leq Z < n/W$, では pivot tile はちょうど 1 つであるため，起動ブロック数は 1 つである．

Fig. 2 に例を示す．各点はタイルを表し，有向辺はタイル計算の依存関係を表している．任意の有向辺において，始点に対応するタイルの計算が終了した後，終点に対応するタイルの計算を開始できる．$Z = 0$ と仮定する．$T_{0,1}$ の計算終了後に $T_{1,1}$ の計算を開始できることを $T_{0,1}$ から $T_{1,1}$ への有向辺で表している．$T_{1,0}$ の計算終了後に $T_{1,1}$ の計算を開始できることを $T_{1,0}$ から $T_{1,1}$ への有向辺で表している．

1 CUDA ブロック辺りのスレッド数を T と表す．1 CUDA ブロックが 1 つのタイルの距離を計算する．1 スレッドが 1 個の距離を計算し，各スレッドは $W \times W/T$ 個の計算を行う．the number of CUDA blocks of kernel A の pivot tile はちょうど 1 つなので，起動 CUDA ブロックは 1 つである．row pivot tile と column pivot tile の数はいずれも $n/W-1$ なので，kernel B または C の起動 CUDA ブロック数は，それぞれ $2(n/W-1)$ または $(n/W-1)^2$ である．

row pivot tile $T_{X,Z}$, column pivot tile $T_{Z,Y}$ を計算する kernel では 1 CUDA block は 1 つの row pivot tile または column pivot tile を計算する．まず $T_{Z,Z}$, $T_{X,Z}$ または $T_{Z,Y}$ を shared memory にコピーする．1 thread が 1 つの distance を計算することで，function F() を実行する．各 Z, $0 \leq Z < n/W$, では row pivot tile および column pivot tile はそれぞれ $n/W-1$ 個であるため，起動ブロック数は $2(n/W-1)$ つである．

non-pivot tile $T_{X,Y}$ を計算する kernel では 1 CUDA block は 1 つの non-pivot tile を計算する．まず $T_{X,Z}, T_{Z,Y}$ を shared memory にコピーし，$T_{X,Y}$ を register にコピーする．1 thread が 1 つの distance を計算することで，function F() を実行する．各 Z, $0 \leq Z < n/W$, では non-pivot tile $(n/W-1)^2$ 個であるため，起動ブロック数は $(n/W-1)^2$ 個である．

Blocked algorithm において pivot tile の各距離 $D[][]$ を計算するためにそのタイルの $D[][]$ のみが必要である．column pivot tile（row pivot tile）の $D[][]$ を計算するためには，pivot tile とその column pivot tile（row pivot tile，それぞれ）の $D[][]$ のみが必要である．non-pivot tile の計算には non-pivot tile 自身の $D[][]$ と column pivot tile と row pivot tile の $D[][]$

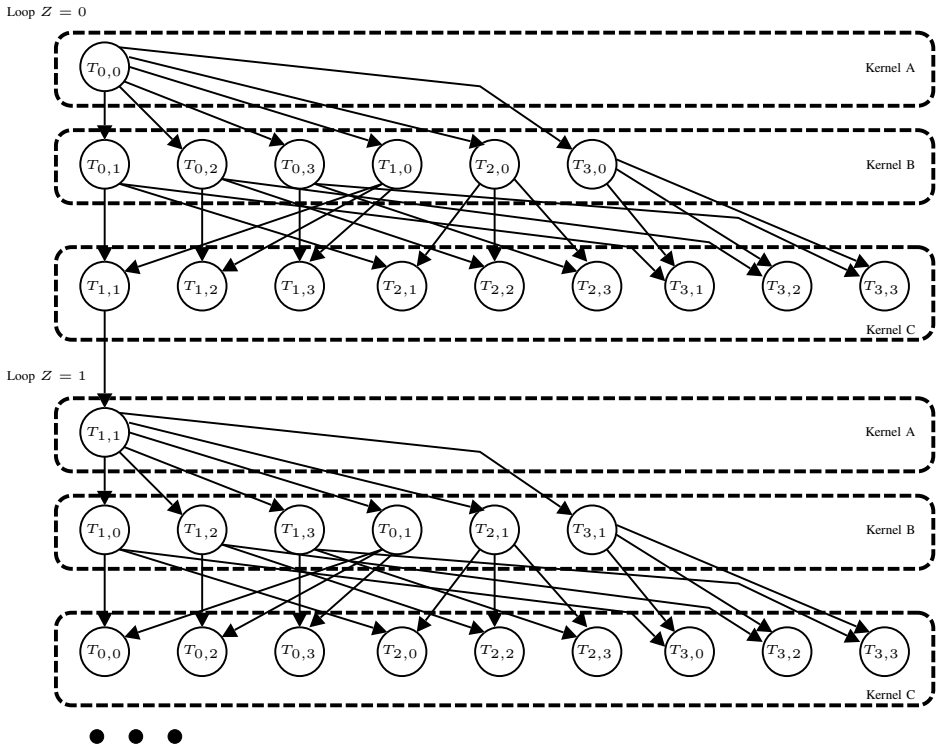

Fig. 2. 複数 kernel 実装における kernel

が必要になる．そこで次のように CUDA ブロックにコピーする．

まず，pivot tile $T_{Z,Z}$ の計算では，$T_{Z,Z}$ の $W \times W$ の距離 $D[][]$ を shared memory にコピーする．$X \neq Z$, $Y \neq Z$, $Z = z_1$ とする．row pivot tile $T_{X,Z}$（column pivot tile $T_{Z,Y}$ それぞれ）の計算では，pivot tile $T_{Z,Z}$ と $T_{X,Z}$（$T_{Z,Y}$）を shared memory にコピーする．non-pivot tile $T_{X,Y}$ の計算では，row pivot tile $T_{X,Z}$ と column pivot tile $T_{Z,Y}$ を shared memory にコピーする．一方，$T_{X,Y}$ を register にコピーする．$T_{X,Z}$ および $T_{Z,Y}$ は CUDA ブロックのどのスレッドからもアクセスされるため，shared memory を使用するが，$T_{X,Y}$ の距離は独立に計算できるため register を使用して高速化を実現する．

IV. 提案実装

本稿で提案する 1 つの kernel で実装したシングル kernel 実装を説明する．まず，Blocked Floyd-Warshall Algorithm では各 Z において，すべてのタイルの中で pivot tile が最も早く計算され，この計算が終了した後 row pivot tile または column pivot tile の計算が開始される．これらの計算が終了した後 non-pivot tile の計算が開始される．つまり，tile の計算順序が決められている．

我々の実装では整数変数 c を使用するが，変数 c は global memory に保存され，0 で初期化されている．CUDA atomic 関数 atomicAdd($\&c, 1$) を使用するが，atomicAdd($\&c, 1$) は排他的に c を 1 増やし，増やす前の c の値を戻り値とするものである．我々は CUDA ブロックにタイルの計算を割り当てることで，グローバルカウンタ技術を使用する．ある CUDA ブロックの最初のスレッドは atomicAdd($\&c, 1$) を実行する．atomicAdd($\&c, 1$) の戻り値を t_i とする．その CUDA ブロックは task ID t_i をもつタイル $T_{I,J}$ の計算を行う．ただし，$T_{I,J}$ の計算が始まる前には $T_{I,J}$ に依存するすべてのタイルの計算が終了していることを確認する．

$T_{I,J}$ の計算が終了した後 atomicAdd($\&c, 1$) を実行し，戻り値が $(n/W)^3$ よりも小さい間同様の操作を繰り返す．またタイルの計算状態を表すフラグの配列を global memory に用意する．$Z = z_1$ におけるあるタイルの計算が終了したならば CUDA ブロックはそのタイルの計算状態を表すフラグに z_1 を書き込む．関数 F() によって計算されるタイルの数は $(n/W)^3$ である．$q = (n/W)^3$ とおく．

CUDA ブロックが atomicAdd($\&c, 1$) を実行し，t_i ($\leq q-1$) を得たとき，計算中の CUDA ブロックは既に 0 から $t_i - 1$ までの task ID をもつタイルが割当てられている．CUDA kernel 呼び出しは正しくタイルの計算が行われている．

CUDA ブロックがどのように kernel 呼び出しをしているか説明する．streaming multiprocessor に同時に送られた CUDA ブロック数を r とする．$q \leq r$ ならばすべての q CUDA ブロックは streaming multiprocessor に同時に割当てられる．我々の kernel はすべての計算を同時に完了することが可能である．もし $q > r$ ならば，グローバルカウンタ c を使用した atomicAdd 関数によって r 個の CUDA ブロックは 0 から $r-1$ のタイルを割り当てる．残りの $q-r$ CUDA ブロックは計算中の r ブロックの計算終了を待つ．計算中の CUDA ブロックの 1 つが計算を完了すると，再び最初のスレッドは atomicAdd($\&c, 1$) を実行し，整数値 r を受け取る．この CUDA ブロックは task ID r をもつタイルの計算を実行する．他の計算中の CUDA ブロックが計算を完了すると，

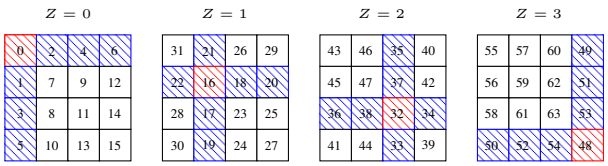

Fig. 3. 4×4 のタイルの task ID

atomicAdd(&c, 1) によって $r+1$ を受け取り，task ID $r+1$ をもつタイルの計算を開始する．atomicAdd(&c, 1) の戻り値が $q-1$ を超えない限り同様の操作を繰り返す．計算中の CUDA ブロックが q を受け取ると終了する．待機中の CUDA ブロックは streaming multiprocessor に割当てられており，最初のスレッドは atomicAdd(&c, 1) を実行する．戻り値は $q-1$ よりも大きいのですぐに停止する．一方，タイルの計算を終了した起動中の CUDA ブロックも atomicAdd(&c, 1) を実行し，$q-1$ よりも大きい値を受け取り，すぐに終了する．このようにすべての計算は終了する．

$n/W \times n/W$ のタイルに task ID を与える．このとき task ID は，各 z, $0 \leq z < n/W$, において pivot tile が最も小さく，column pivot tile または row pivot tile は non-pivot tile よりも小さく，non-pivot tile が最も大きくなるように与えられる．また row pivot tile または column pivot tile は pivot tile の計算終了後に，non-pivot tile は，row pivot tile または column pivot tile の計算終了後に，計算が開始されるため，各 tile に計算開始前と計算終了を表すフラグを用意する．各 tile に対応する CUDA block は tile の計算前に計算終了フラグを確認する．

$X \neq Z$, $Y \neq Z$ とおく．各 Z, $0 \leq Z < n/W$, において各 tile の計算終了フラグを確認することにより，row pivot tile $T_{X,Z}$ または column pivot tile $T_{Z,Y}$ は pivot tile $T_{Z,Z}$ の計算終了後に計算を開始できる．また，non-pivot tile $T_{X,Y}$ は，row pivot tile $T_{X,Z}$ または column pivot tile $T_{Z,Y}$ の計算終了後に，計算を開始できる．

まず，Figure 3 に，$n/W = 4$ のときの各 z, $0 \leq z < n/W = 4$ のタイルの task ID を表す．たとえば Figure 1 では，row pivot tile $T_{1,3}$ も column pivot tile $T_{2,1}$ も pivot tile $T_{1,1}$ の計算終了後に計算が開始でき，non-pivot tile $T_{2,3}$ は $T_{1,3}$ も $T_{2,1}$ も計算が終了した後に計算が開始できる．

複数 kernel 実装と同じく，shared memory または register にコピーされた距離 D を計算する function F() は 1 thread が 1 つの距離 D を計算する．

V. バルク計算

まず複数 kernel 実装によるバルク計算法を提案する．III 節で説明したとおり複数 kernel 実装は 3 つの kernel で構成されているため，次のように各 kernel を拡張することでバルク実行を行うことができる．kernel A はすべてのインスタンスの pivot tile のみの距離を計算する．kernel B はすべてのインスタンスの column pivot tile および row pivot tile の計算を行い，kernel C はすべてのインスタンスの non-pivot tile の計算を行う．これにより与えられたインスタンスに対し同時に計算できる．

次に，シングル kernel 実装によるバルク計算法を提案する．$(n/W)^2$ 個のタイルをもつ m 個のインスタンスのすべてのタイルに task ID を与える．タイルの総数は $m \times (n/W)^2$ である．各インスタンスの点数を n とする．4×4 タイルをもつ 8 個のインスタンスのタイルの task ID を Fig. 4 に示す．ただし $Z = 0, 1$ の場合である．task ID は各インスタンスで連続するものではなく，全インスタンスで連続している．IV 節でシングル kernel 実装について説明したが，どの CUDA ブロックも最初のスレッドは atomicAdd(&c, 1) を実行する．atomicAdd(&c, 1) で受け取った値を t_i とすると，依存関係があるすべてのタイルの計算が終了していれば task ID t_i をもつタイルの計算を開始する．

TABLE I
シングル KERNEL 実装および複数 KERNEL 実装の GPU 計算時間（ミリ秒）．シングル KERNEL 実装ににおける起動 CUDA ブロック数を B とする．ただし，$B \geq 2048$ のとき $B = 1024$ とする．なお，距離 D は FLOAT 型浮動小数点数とする．

T	点数				
	64	128	256	512	1024
複数 kernel 実装 [5]					
64	0.1197	0.2533	0.504	1.020	2.61
128	0.0648	0.1282	0.257	0.576	1.74
256	*0.0441	0.0864	0.171	0.421	*1.39
512	0.0450	*0.0795	*0.159	*0.401	1.40
1024	0.0540	0.0960	0.191	0.474	1.70
提案シングル kernel 実装					
64	0.1162	0.2321	0.477	0.962	2.68
128	0.0688	0.1332	0.249	0.511	1.70
256	0.0440	0.0812	0.158	*0.388	*1.36
512	*0.0388	*0.0737	*0.146	0.422	1.57
1024	0.0478	0.0917	0.191	0.608	2.29
B	4	16	32	128	256
Speedup	1.14	1.08	1.08	1.03	1.02

T	点数				
	2048	4096	8192	16384	32768
複数 kernel 実装 [5]					
64	12.99	68.2	474	3669	29250
128	8.33	55.0	419	3365	27081
256	*7.33	*51.8	*400	*3149	*25674
512	7.77	54.3	421	3313	27058
1024	9.71	68.0	517	4064	32710
提案シングル kernel 実装					
64	13.33	74.1	551	4501	37458
128	7.60	50.4	396	3523	29297
256	*7.04	*48.3	*378	*3020	*24897
512	8.41	59.4	462	3815	32138
1024	12.86	94.3	739	6091	51572
Speedup	1.04	1.07	1.06	1.04	1.03

VI. 性能評価

Blocked Floyd-Warshall アルゴリズムのシングル kernel 実装の性能を評価するために，複数 kernel 実装 [5] と計算時間の比較を行った．入力は点数 $n = 64, 128, \ldots, 32768$ の 4-regular グラフで，距離 $D[][]$ は float 型の乱数とした．各タイルの大きさ W を $W = 32$ とした．

Table I はインスタンス数が 1 つの場合の NVIDIA Tesla V100 上の計算時間（ミリ秒）を表す．CUDA ブロックあたりのスレッド数を T とし，起動する CUDA ブロック数を B とする．我々が提案するシングル kernel 実装は，複数 kernel 実装に比べて 1.02–1.15 の高速化を実現している．シングル kernel 実装は既存実装に比べて kernel 起動回数が非常に少なく，kernel 呼び出しのオーバーヘッドが軽減されたためと思われる．

$Z = 0$

Instance G_0

0	9	11	13
8	56	58	60
10	57	96	98
12	59	97	120

Instance G_1

1	15	17	19
14	61	63	65
16	62	99	101
18	64	100	121

Instance G_2

2	21	23	25
20	66	68	70
22	67	102	104
24	69	103	122

Instance G_3

3	27	29	31
26	71	73	75
28	72	105	107
30	74	106	123

Instance G_4

4	33	35	37
32	76	78	80
34	77	108	110
36	79	109	124

Instance G_5

5	39	41	43
38	81	83	85
40	82	111	113
42	84	112	125

Instance G_6

6	45	47	49
44	86	88	90
46	87	114	116
48	89	115	126

Instance G_7

7	51	53	55
50	91	93	95
52	92	117	119
54	94	118	127

$Z = 1$

Instance G_0

248	140	187	225
141	128	137	139
188	136	184	186
226	138	185	224

Instance G_1

249	146	192	228
147	129	143	145
193	142	189	191
229	144	190	227

Instance G_2

250	152	197	231
153	130	149	151
198	148	194	196
232	150	195	230

Instance G_3

251	158	202	234
159	131	155	157
203	154	199	201
235	156	200	233

Instance G_4

252	164	207	237
165	132	161	163
208	160	204	206
238	162	205	236

Instance G_5

253	170	212	240
171	133	167	169
213	166	209	211
241	168	210	239

Instance G_6

254	176	217	243
177	134	173	175
218	172	214	216
244	174	215	242

Instance G_7

255	182	222	246
183	135	179	181
223	178	219	221
247	180	220	245

Fig. 4. 4×4 のタイルを持つ 8 個のインスタンスに対する task ID．ただし $Z = 0, 1$ の場合のみ．

また，バルク計算法の性能評価のために，GPU 上に実装し計算時間を比較した．float 型乱数を距離としてもつ $n = 64, 128, \ldots 2048$ なる 4-regular グラフを 1 つのインスタンスとし，$4, 8, 16, \ldots 2048$ 個のインスタンスに対し計算を行った．NVIDIA Tesla V100 における計算時間（ミリ秒）を Table II に記す．

VII. まとめ

本稿では，Blocked Floyd-Warshall アルゴリズムに対し，1 つの kernel による GPU 実装を提案し，NVIDIA Tesla V100 において既存実装に比べて 1.02–1.15 倍高速であることを示した．また，複数インスタンスが与えられた場合へ拡張した GPU 実装も提案し，その性能を評価した．

REFERENCES

[1] K. Nakano, D. Takafuji, S. Fujita, H. Matsutani, I. Fujiwara, and M. Koibuchi, “Randomly optimized grid graph for low-latency interconnection networks,” in *Proc. of International Conference on Parallel Processing (ICPP)*, 2016, pp. 340–349.

[2] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, “A blocked all-pairs shortest-paths algorithm,” *J. Exp. Algorithmics*, vol. 8, Dec. 2003.

[3] G. J. Katz and J. T. Kider, “All-pairs shortest-paths for large graphs on the GPU,” in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2008, pp. 47–55. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413957.1413966

[4] T. Okuyama, F. Ino, and K. Hagihara, “Fast blocked floyd-warshall algorithm on the GPU,” *IPSJ Transactions on Advanced Computing Systems*, vol. 3, no. 2, pp. 57–66, Jun 2010.

[5] B. D. Lund and J. W. Smith, “A multi-stage CUDA kernel for floyd-warshall,” *CoRR*, vol. abs/1001.4108, 2010. [Online]. Available: http://arxiv.org/abs/1001.4108

[6] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[7] NVIDIA Corporation, “NVIDIA CUDA C programming guide version 10.1,” 2019.

TABLE II
$4, 8, 16, \ldots 2048$ 個のインスタンスに対するシングル KERNEL 実装および複数 KERNEL 実装の GPU 計算時間（ミリ秒）．ただし，各インスタンスは FLOAT 型乱数を距離としてもつ $n = 64, 128, \ldots 2048$ なる 4-REGULAR グラフである．

4 instances

	the number of vertices					
	64	128	256	512	1024	2048
複数 kernel	0.0406	0.0837	0.209	0.720	3.90	26.4
シングル kernel	0.0394	0.0777	0.211	0.654	3.54	24.6

8 instances

複数 kernel	0.0470	0.1041	0.313	1.21	7.11	51.3
シングル kernel	0.0437	0.0959	0.227	1.04	6.58	48.7

16 instances

複数 kernel	0.0527	0.136	0.456	2.05	13.5	101.2
シングル kernel	0.0465	0.104	0.360	1.84	12.7	97.4

32 instances

複数 kernel	0.0625	0.191	0.728	3.73	26.4	201
シングル kernel	0.0525	0.124	0.552	3.46	25.2	194

64 instances

複数 kernel	0.0838	0.260	1.11	7.11	52.0	400
シングル kernel	0.0601	0.181	1.00	6.75	50.2	388

128 instances

複数 kernel	0.104	0.391	2.05	13.9	103	799
シングル kernel	0.080	0.303	1.91	13.4	100	777

256 instances

複数 kernel	0.148	0.632	3.92	27.4	206	1597
シングル kernel	0.107	0.567	3.74	26.7	200	1553

512 instances

複数 kernel	0.225	1.15	7.64	54.5	411	3193
シングル kernel	0.198	1.09	7.43	53.2	401	3106

1024 instances

複数 kernel	0.409	2.19	15.1	109	822
シングル kernel	0.349	2.15	14.8	106	802

2048 instances

複数 kernel	0.682	4.24	29.9	217	1643
シングル kernel	0.644	4.27	29.5	213	1603