社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－ARC－177 (25)
2008－HPC－114 (25)
2008／3／6

CUDAによる全点対最短経路問題の高速化

奥 山 倫 弘† 伊 野 文 彦†† 萩 原 兼 一††

本論文では全点対最短経路（APSP：All-Pairs Shortest Path）問題をGPU（Graphics Processing Unit）を用いて高速化した結果を述べる．提案手法は，GPUで動作するプログラムをGPU向けの開発環境CUDA（Compute Unified Device Architecture）を用いて記述する．アルゴリズムには単一始点最短経路を繰り返し求める手法（SSSP反復法）を用いる．問題全体での逐次処理を減らしてより高い速度向上を得るために，1つのSSSP問題を1つのタスクとし，それらのタスクを並列処理する．さらに，共有メモリを用いてタスク間でデータを共有し，グローバルメモリの参照を削減する．結果，既存手法よりも3.5〜18倍高速であった．また，SSSP反復法の性能がグラフの特性に依存して変動することを示す．

Accelerating All-Pairs Shortest Path Problem Using CUDA

Tomohiro Okuyama,† Fumihiko Ino†† and Kenichi Hagihara††

This paper presents acceleration results of the all-pairs shortest path (APSP) problem on a graphics processing unit (GPU). The proposed method is implemented using compute unified device architecture (CUDA), which offers the development environment for GPUs. The method is based on an iterative algorithm that repeatedly computes single-source shortest paths (SSSPs). In order to obtain a higher speedup, we decrease the sequential part of the algorithm by processing SSSP problems in parallel. Furthermore, we reduce the access to global memory by using shared memory, which allows tasks to share data between them. As a result, our method is 3.5-18 times faster than an existing method. We also show that the iterative method varies the performance depending on the characteristic of the graph.

1. はじめに

近年，GPU[1]（Graphics Processing Unit）の高い計算性能を用いて汎用計算を高速化する研究が盛んである．GPUはグラフィクス処理を高速化するチップであるため，そのプログラミングモデルには様々な制約がある．

nVIDIA社のCUDA[2]（Compute Unified Device Architecture）はGPUを汎用計算に用いるための開発環境である．この環境はGPUを並列計算機として扱う．したがって，グラフィクス主体のプログラミングモデルよりも柔軟にプログラムを記述できる．

そこで，文献3)はCUDAを用いて全点対最短経路（APSP：All-Pairs Shortest Path）問題を含むグラフ処理を高速化している．APSP問題とは，グラフ上の2頂点uおよびv間の最短経路を，すべての組$\langle u, v\rangle$に対し求める問題である．基本的なグラフ処理であり，様々な応用分野で用いられる．

文献3)は，APSPを求めるアルゴリズムとして，単一始点最短経路（SSSP：Single Source Shortest Path）問題を繰り返し解く手法（SSSP反復法）とFW（Floyd Warshall）法を用いている．前者は後者よりも高速であるが，この手法には以下の2点に関して改善の余地がある．(1) 用いる並列性．(2) CUDAのメモリ階層を考慮したデータ参照．

(1)について，既存手法[3]では1つのSSSP問題を並列処理しているが，問題を繰り返し解く部分は逐次実行している．これらの問題は並列計算でき，この部分に改善の余地がある．(2)については，ビデオメモリ上のデータを直接読み書きしていて，GPU内に存在するより高速な共有メモリを用いていない．

本稿では，CUDAを用いてAPSPを求める処理を高速化する．SSSP反復法を用いる既存手法[3]を基に，複数のSSSP問題をタスク並列計算するSSSP並列法

† 大阪大学基礎工学部情報科学科
Department of Information and Computer Sciences, School of Engineering Science, Osaka University
†† 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

を提案する．複数のタスクを並列計算することで，全体に対する並列化部分の割合を増加させて高速化を図る．また，タスク並列計算により，タスク間で共通するデータへの参照が生じることに着目し，これらを共有メモリに配置して遅いメモリの参照を削減する．

以降は，2 節で CUDA のアーキテクチャについて述べ，3 節で SSSP 反復法のアルゴリズムと既存手法を示す．4 節で我々の実装とその CUDA の特性を考慮した調整について説明する．その後，5 節で実験結果を示し，6 節にまとめを述べる．

2. CUDA 開発環境

CUDA のアーキテクチャでは，GPU は複数のマルチプロセッサ（MP）を持つ．各 MP は複数のプロセッサを内包する．CUDA では，これらのプロセッサ上で多数のスレッドを並列実行し，高速化を図る．複数のスレッドをまとめたものをブロックと呼ぶ．MP への仕事の割り当てはブロック単位で行われる．ブロックはワープと呼ばれる単位に分割される．MP 内のプロセッサは 1 つのワープを SIMD 演算により実行する．1 ブロック内のスレッド数をブロックサイズと呼び，ワープ内のスレッド数をワープサイズと呼ぶ．

スレッドが参照できる記憶領域にはレジスタ，共有メモリ，グローバルメモリ等がある．レジスタと共有メモリはほぼ遅延無しで参照できる．一方，グローバルメモリはビデオメモリ上に確保されるため，参照時に 400〜600 クロックサイクルの遅延を生じる．グローバルメモリは全スレッドが共有し，CPU とのデータの受け渡しに用いる．共有メモリのデータは同じブロックのスレッド間で共有される．

GPU の演算性能を使い切るには，グローバルメモリへのアクセス時間の隠蔽，グローバルメモリからの転送量の抑制および転送時に帯域幅を使い切ることが重要となる．アクセスの隠蔽には，各 MP にブロックを複数割り当てる．転送量を抑えるには，共有メモリを用いてグローバルメモリへの読み書きをブロック単位で削減する．帯域幅を使い切るには，複数のスレッドがグローバルメモリ上の連続領域を coalesced 参照[2]する必要がある．

スレッドの動作はカーネルに記述する．GPU は，CPU 側のコードからカーネルを起動することでカーネルを実行する．

3. SSSP 反復法

非負の重み付き有向グラフ $G = (V, E, W)$ に対し，APSP を求める．この手法では，最短経路のコストのみを求め，最短経路の経由頂点列は求めない．ここで，V はグラフ上の頂点の集合，E は辺の集合，W は辺の重みの集合である．$|V|$ を頂点数，$|E|$ を辺の数とする．SSSP 反復法は，始点 $s \in V$ を変えながら，$|V|$ 個の SSSP 問題を逐次解く．

```
1: u := threadID
2: if Ma[u] = true then
3:     Ma[u] := false
4:     for i := Va[u] to Va[u + 1] − 1 do
5:         v := Ea[i]
6:         Ua[v] := min(Ua[v], Ca[u] + Wa[i])
7:     end for
8: end if
```

図 1 SSSP 反復法のカーネル A

3.1 SSSP を求めるアルゴリズム

文献 3) の SSSP を求めるアルゴリズムを説明する．c_v を頂点 v のコスト，N_v を v の隣接頂点集合，$w_{u,v}$ を辺 (u, v) の重みとする．まず，$c_v = \infty$ $(v \neq s,\ v \in V)$，$c_s = 0$ とする．s の各隣接頂点 $n \in N_s$ のコスト c_n を $\min(c_n, c_s + w_{s,n})$ で更新する．続いて，コストが更新された頂点 n について，それぞれの隣接頂点のコストを更新する．以降同様の更新をすべての頂点のコストの上書きがなくなるまで繰り返す．

なお，CUDA を用いて実装する場合，グローバルメモリアクセスに対する排他制御が必要である．CUDA の Compute Capability 1.1 以降の GPU がこの機能を持つ．排他制御が使えない場合，経路のうちの一部に誤りが生じる．

3.2 CUDA を用いた実装

CUDA を用いて APSP を求める既存手法[3] の実装を説明する．この実装では，SSSP を求める際の隣接頂点の更新を GPU で並列処理する．そして，CPU で SSSP を $|V|$ 回解く繰り返しを行う．以降は，1 つの SSSP を求める処理について述べる．

SSSP を求めるアルゴリズムの頂点ごとの処理を並列化し，CUDA の 1 つのスレッドが 1 つの頂点を担当する．隣接頂点のコスト更新を行うカーネル A（**図 1**）と更新を補助するカーネル B の 2 つのカーネルを 1 組として順に起動する．1 つの SSSP を求めるために，CPU はこのカーネルの組を繰り返し起動する．

グラフの表現には隣接リストを配列に格納したものを用いる（**図 2**）．そのために，要素数 $|V|$ の配列 Va と要素数 $|E|$ の配列 Ea を用いる．Ea には頂点ごとにすべての隣接頂点の番号を格納する．この頂点ごとの隣接リスト Ea への格納位置を Va に格納する．さらに，要素数 $|E|$ の配列 Wa を用いて，Ea の各要素に対応する辺の重みを格納する．これらの配列と，後

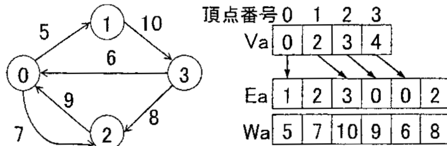

図 2 隣接リストによるグラフの表現

述する配列 Ma, Ca および Ua はすべてビデオメモリ上のグローバルメモリに作成する.

始点から各頂点への最短経路のコストを格納するために, それぞれ要素数 $|V|$ の配列 Ca および Ua を用いる. また, 頂点のコスト更新の有無を示すフラグを要素数 $|V|$ の配列 Ma に格納する.

カーネル A (図 1) では, スレッド u は最初に $Ma[u]$ を参照して担当頂点 u のコスト更新の有無を調べる. 更新されていた場合, 頂点 u のすべての隣接頂点のコストを更新し, Ua を上書きする. 1 回の更新完了後, 全スレッドの同期をとるためにカーネル実行を終える. その後, カーネル B を起動する. カーネル B では, Ua の値を Ca に反映し, コストが更新されていれば Ma を設定する. カーネル B 実行後, Ma のいずれかの要素が true であればカーネル実行を繰り返し, そうでなければ終了する.

4. 提案手法

提案手法は, SSSP 反復法と同様のグラフに対し, 最短経路のコストのみを求める. SSSP 反復法と同様に始点を変えながら $|V|$ 個の SSSP 問題を解く. SSSP を解くアルゴリズムも同じものを用いる. SSSP 反復法と異なるのは, 問題を解く繰り返しを逐次行わず, 1 個の SSSP 問題を 1 つのタスクとし, P 個のタスクを並列に解くことを $|V|/P$ 回繰り返す点である. 全体に対して並列実行する部分の占める割合が増えるため, 高速化を見込める. タスク並列計算を $|V|/P$ 回行う繰り返しは逐次行う. 以降, P 個のタスクの並列計算について説明する.

4.1 タスク並列計算

1 回のカーネル実行で起動するスレッド数を増やし, タスク並列計算を行う. SSSP 反復法と同じく, CPU ではカーネルを繰り返し起動して P 個の SSSP を解く.

GPU では, $P|V|$ 個のスレッドを用いて, P 個のタスクを並列計算する. 各スレッドはいずれかのタスクに属し, 1 つの頂点を担当する. SSSP 反復法の場合と同じく, 2 つのカーネル A' (**図 3**) および B' を用いる. カーネル A および B と異なるのは主に以下の 3 点である. (1) 各スレッドは担当頂点番号に加え, 自身が属するタスク番号を持つ. (2) Va, Ea および Wa の一部をタスク間で共有する. (3) Ma, Ca および Ua はタスクごとに別々の要素を用いる.

```
 1: u := threadID div P // vertex id
 2: t := threadID mod P // task id
 3: shared ms[u] := false
 4: if Ma[u, t] = true then
 5:     ms[u] := true
 6: end if
 7: if ms[u] = true then
 8:     shared begin[u] := Va[u], end[u] := Va[u + 1]
 9:     neighbors := end[u] − begin[u]
10:     if t < neighbors then
11:         shared es[u, t] := Ea[begin[u] + t]
12:         shared ws[u, t] := Wa[begin[u] + t]
13:     end if
14:     if Ma[u, t] = true then
15:         Ma[u, t] := false
16:         for i := 0 to neighbors − 1 do begin
17:             v := es[u, i]
18:             Ua[v, t] := min(Ua[v, t], Ca[v, t] + ws[u, i])
19:         end for
20:     end if
21: end if
```

図 3 カーネル A'

また, ワープ内での分枝の発生を抑えるため, 同じワープ内のスレッドは同じ頂点を担当する. 担当頂点が同じであれば, 隣接頂点のコストを更新する際のループ回数もスレッド間で等しい. よって, このループではワープ内での分枝が発生しない. ただし, Ma の値によって更新が必要か判定するため, ワープ内での分枝は完全には除去できない.

4.2 データ構造と共有メモリの使用

データ構造は, SSSP 反復法と同様の配列を用いる. ただし, タスクごとに値が異なりうる配列 Ma, Ca および Ua は要素数を $P|V|$ に変更し, タスクごとに値を保持する. 一方, Va, Ea および Wa は書き込みが行われないためすべてのタスクで共有する.

担当頂点の同じスレッドが参照する Va, Ea および Wa の要素は同じである. そこで, 共有メモリを用いて, 同じ頂点を担当するスレッド間でこれらの要素を共有する. これにより, グローバルメモリアクセスを削減し, 高速化を図る.

また, 図 3 の 11〜12 行目に示すように, 共有メモリへ Ea および Wa の要素を転送するときは, 同一頂点を担当するスレッドで並列に行う. 各頂点の処理で参照する Ea および Wa の領域はそれぞれ連続している. したがって, この参照は coalesced 参照になる. 特に, 頂点の次数が大きい場合に並列読み込みと共有の効果が高いと考えられる.

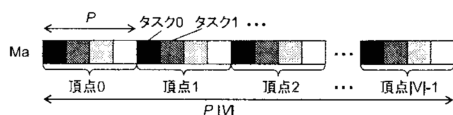

図 4　配列 Ma のレイアウト

Ma, Ca および Ua は，タスクおよび頂点ごとに値が異なる．よって，共有メモリを用いた参照の効率化は容易ではない．しかし，Ma については，スレッドが属するタスクと担当頂点の割り当てを考慮して，**図 4** のように配置することで coalesced 参照にできる．さらに，参照先要素の添字計算に必要な命令を削減するために，Ca および Ua も Ma と同様の配置にする．

4.3　並列処理するタスク数

SSSP 反復法ではビデオメモリ使用量は $4|V|+2|E|$ であったが，提案手法では $(3P+1)|V|+2|E|$ に増加する．したがって，扱えるグラフの大きさは SSSP 反復法よりも小さい．大きなグラフを扱うためには P を小さくすべきである．しかし，P が小さいと SSSP 反復法そのものに近づくため，P の選択は重要である．

P はブロックサイズもしくはワープサイズとすることが適切と考えられる．ブロックサイズの場合，共有メモリによりデータ共有を行える．一方，ワープサイズの場合，ワープが SIMD 型の並列実行単位であり，ワープ単位での分岐コストが小さい．ブロックはワープよりも大きいほうが効率がよく，通常はワープの方が小さい．そこで，P をワープサイズとする．

5. 評価実験

実験環境の CPU は 2.93GHz 駆動の Intel Core2 Extreme X6800 であり，2GB のメインメモリを持つ．GPU は nVIDIA GeForce 8800 GTX であり，16 基の MP を持つ．ビデオメモリの容量は 768MB である．OS は Windows XP Professional，ビデオドライバのバージョンは 169.21 で，CUDA SDK バージョン 1.1 を用いた．

5.1　実行時間の評価

表 1 に SSSP 反復法[3]（既存手法）と提案手法のランダムグラフ[4] に対する実行時間を示す．

グラフは $|E| = 4|V|$ とし，辺の重みは 1〜$|V|$ の間の整数である．比較のため，提案手法で共有メモリを用いない場合，FW 法および CPU 版の実行時間も示す．CPU での SSSP 反復法の実装は，ダイクストラ法を反復するものである．ダイクストラ法は GPU 実装での SSSP を求める手法とは異なるが，CPU で SSSP を求める高速な手法である．

FW 法の GPU 実装は，文献 3) の手法を基に CUDA の特性に合わせて改良したもので，約 10 倍高速である．改良点は，1 ブロックで隣接行列の 1 行を更新することで coalesced でない参照を削減する点である．また，行単位で更新が不要な場合は行の更新を省略し，グローバルメモリ参照を削減する．さらに，共有メモリやテクスチャメモリを使用することで高速化した．

CPU 版 FW 法の $|V| = 32$K，GPU 版 FW 法の $|V| = 16$K 以上はメモリ容量の制限により計測できない．CPU ではメインメモリ容量の制限により，FW 法で $|V| \geq 22$K を，GPU ではビデオメモリ容量の制限により，FW 法で $|V| \geq 14$K を扱えない．

表 1 より提案手法は既存手法より最大で約 18 倍高速である．また，提案手法で共有メモリを用いない場合に比べ 1.2〜1.5 倍高速である．特に，$|V|$ が少ない場合に既存手法より高速である．既存手法では $|V|$ が少ないとメモリアクセスの隠蔽を行うには不利なためである．例えば，$|V| = 1$K のグラフの場合，既存手法では 1024 個のスレッドが並列実行される．このとき，MP1 基あたりのスレッド数は 64 と少なく，メモリアクセスの隠蔽を行うには不利である．

一方，提案手法では $P = 32$ より，$|V| = 1$K のときでもスレッド数は 32K となる．よって，メモリアクセスの隠蔽のために，各 MP に複数のブロックを割り当て可能である．また，提案手法のカーネル実行回数は既存手法のほぼ $1/P$ であるため，カーネル実行を完了させるための同期待ち時間の合計も既存手法より短いと考えられる．これらの理由により，$|V|$ が少ないときに提案手法がより高速であったと考えられる．

CPU でのダイクストラ法の反復は，$|V| \leq 4$K のグラフで既存手法より高速である．表 1 の実験に用いたグラフは $|V|$ に比べて頂点次数の少ない疎なグラフである．ダイクストラ法は，疎なグラフの処理に適しているため，ダイクストラ法が高速に動作した．

5.2　グラフ形状への実行時間の依存

既存手法および提案手法の実行時間は，グラフの形状に依存する．**図 5** は $|V|$ を固定して，$|E|$ を変化させた場合の実行時間である．

$|E|$ を増やすと既存手法および提案手法の実行時間は増加している．ただし，$|E| = 8$K から 16K に倍増する場合は実行時間が減少している．これは，SSSP あたりのカーネル起動回数が減少したためである．既存手法の場合，平均 27 回から 20 回に減少している．

$|E| \geq 16$K の場合も辺が増えるごとにカーネル起動回数は減少する．しかし，減少の度合いは小さく，$|E|$ を倍増させても 1 回の SSSP あたり 1〜2 回の減

表 1　ランダムグラフ[4] での実行時間（単位：ミリ秒）

実装	手法	頂点数 $\|V\|$					
		1K	2K	4K	8K	16K	32K
GPU	既存手法（SSSP 反復法[3]）	827	1,813	4,316	10,117	27,506	75,300
	提案手法	44	125	407	1,400	5,478	21,437
	提案手法（共有メモリなし）	54	164	583	2,147	8,634	34,408
	FW 法	131	629	3,863	27,804	—	—
CPU	ダイクストラ法反復	145	645	2,851	12,495	53,854	233,628
	FW 法（2 スレッド）	583	5,281	39,615	327,793	2,585,213	—

—：メモリ不足により計測不能

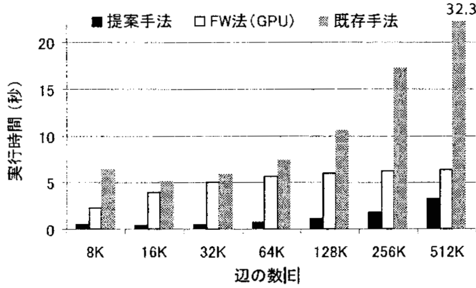

図 5　辺の数 $|E|$ を変化させた場合の実行時間

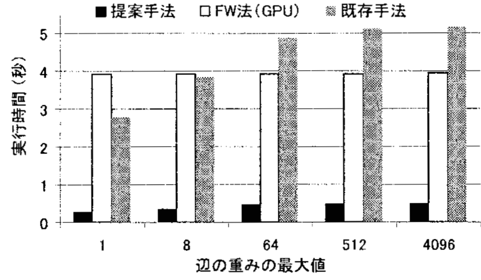

図 6　辺の重みの最大値を変化させた場合の実行時間

少であった．したがって，$|E| \geq 32\mathrm{K}$ では，1 回あたりのカーネル実行時間が全体の実行時間を増加させている．カーネル実行時間が増加する理由は，隣接頂点のコストを更新するときに，カーネル内で頂点の出次数回の繰り返しを行うためである．また，1 回のカーネル実行で更新処理を行うスレッド数が増加していることも考えられる．これらにより，カーネル実行時間が $|E|$ の増加に伴って長くなったと考えられる．

一方，FW 法では実行時間の変動は少ない．このアルゴリズムは，計算量が $O(|V|^3)$ と $|V|$ にのみ依存するためである．$|E|$ が少ないときに実行時間が短いのは，FW 法が計算に用いる隣接行列の更新を行単位で省く処理が多数発生した結果である．

図 6 に辺の重みの範囲を変化させた場合の実行時間を示す．$|V| = 4\mathrm{K}$，$|E| = 16\mathrm{K}$，最小の重みを 1 に固定し，重みの最大値を変化させた．

既存手法および提案手法では，重みの増加に伴い実行時間も増加しているが，その増加量は $|E|$ を変化させた場合よりも小さい．これは，辺の重みがアルゴリズムでの繰り返し回数に直接的に影響しないためである．この場合の実行時間の増加は，以下の理由によると考えられる．

辺の重みの幅が広がると，重みの大きな辺が頂点のコストを設定した後，経路長は短いが多くの辺を辿る経路が頂点のコストを上書きする可能性が高くなる．つまり，各頂点のコストが更新される回数が増加する．さらに，ある頂点 u のコストが更新されると，始点 s からの最短経路が u を経由しているすべての頂点のコスト更新が発生する．その結果，実行時間が増加する．

最後にグラフの構造を変えた場合の実行時間を**表 2** に示す．R-MAT のグラフ生成の都合により，各グラフは $|V| = 4.6\mathrm{K}$ とした．ランダムグラフ[4] と R-MAT[5][6] では $|E| = 16\mathrm{K}$，辺の重みを 1〜4K までの間とした．輪と完全グラフでは，各辺の重みは 1 とした．ランダムグラフは表 1 で用いたグラフである．R-MAT は頂点次数の分布がべき乗則にしたがい，少数の次数の高い頂点と多数の次数の低い頂点が存在するグラフである．用いたグラフでは頂点の最大次数は 1241 で，平均次数 7 以下の頂点が全頂点の約 85%を占める．輪は，全頂点を輪状につないだグラフである．

R-MAT での既存手法および提案手法の実行時間は，それぞれランダムグラフの実行時間の 1.7 および 1.6 倍である．このグラフでは頂点の次数に偏りがあるため，カーネル A 内のループ回数がスレッドごとに異なる．したがって，MP 内の負荷が偏り，R-MAT の実行時間はランダムグラフよりも長くなる．

輪の場合，提案手法の実行時間は既存手法の約 1/17 である．しかし，ランダムグラフの場合よりも 100 倍以上長い．このグラフでは 1 回のカーネル実行で 1 つの頂点のみ処理が行われる．したがって，1 つの SSSP を求めるために $|V|$ 回カーネルを起動する．その結果，このグラフの場合，既存手法および提案手法の実行時

表 2 形状を変更した場合の実行時間（単位：秒）

グラフ	既存手法[3]	提案手法	FW 法（GPU）
ランダムグラフ[4]	5.9	0.6	5.4
R-MAT[5]	10.3	1.0	3.2
輪	1150.8	67.8	3.8
完全グラフ	1446.3	8.9	10.2

間はランダムグラフの場合よりも長くなる．

既存手法よりも提案手法の実行時間が短い理由は，P 個の SSSP を並列計算するためであると考えられる．既存手法では，1 回のカーネル実行で 1 つの頂点しか処理しないが，提案手法は P 個処理する．P は GPU が持つプロセッサ数よりも少なく，P 個のタスクの並列計算により単純に実行時間が $1/P$ になると考えられる．一方で，1 回のカーネル実行で 1 頂点しか更新されない場合，提案手法のほうがスレッド内での条件分岐等のコストが多くなる．その結果，全体の実行時間は $1/P$ にはならず，提案手法の実行時間が既存手法の約 1/17 であったと考えられる．

完全グラフでは，既存手法の実行時間がランダムグラフの 240 倍以上である．一方，提案手法では，ランダムグラフの 15 倍程度に抑えられている．

このグラフでは，各頂点の隣接頂点数が $|V|-1$ 個存在する．これにより，カーネル内の隣接頂点のコスト更新を行うループ回数が $|V|-1$ 回となる．このループは各スレッド内で逐次行われるため，カーネル実行時間が非常に長くなる．

このループ内では，1 頂点の処理につき Ea，Wa の $|V|-1$ 個要素をそれぞれ参照する．提案手法では，共有メモリを用いて Ea および Wa を P スレッド間で共有し，共有メモリへの読み出しも P スレッドで並列に行う．これにより，提案手法では既存手法ほど実行時間が延びなかったと考えられる．

FW 法は R-MAT および輪で実行時間がランダムグラフより短い．輪はランダムグラフよりも疎であり，R-MAT は大多数の頂点の次数が低いため，隣接行列の行単位での更新省略が多数回発生する．その結果，実行時間が短くなったと考えられる．完全グラフでは実行時間が増加しているが，これは行単位での更新省略が全く起きないためである．しかし，ランダムグラフと比較した場合の実行時間は最大でも 1.9 倍であり，既存手法および提案手法の場合のような変動はない．

6. まとめ

SSSP 反復法を基に，1 つの SSSP 問題を 1 つのタスクとし，複数のタスクを並列計算することで CUDA を用いた APSP 問題の解法を高速化した．複数のタスクを並列に解くことで，小さなグラフに対しても GPU が効率よく動作するスレッド数を確保できた．また，共有メモリを用いてタスク間でデータを共有することにより，グローバルメモリ参照を削減した．しかし，SSSP 反復法と同様，提案手法でも排他制御機構を持たない GPU では正しい結果を得ることはできない．また，コストだけでなく経路の経由頂点も求める場合は CUDA の排他制御機構だけでは正しい結果を得ることはできないと考えられる．

実験の結果，既存手法である SSSP 反復法よりも 3.5〜18 倍高速であった．辺の数を変更した場合は，辺の数が少ない領域ではカーネル起動回数の減少によって実行時間が減少した．辺の数が多い領域では，カーネル実行時間の増加により，実行時間が増加した．辺の重みの範囲を変化させた場合も実行時間は変動するが，その変動幅は辺の数を変化させる場合よりも小さい．

また，グラフの形状を変更した場合，既存手法および提案手法は共に実行時間が長くなる場合があった．

今後は，排他機構を備えた GPU での実験や，排他制御を必要としない SSSP 問題を解くアルゴリズムを用いることを検討していく予定である．

謝辞 本研究の一部は，科学研究費補助金基盤研究（B）（2）（18300009），若手研究（B）（19700061）および大阪大学グローバル COE プログラム「予測医学基盤」の補助による．

参考文献

1) Owens, J.D. and et al.: A Survey of General-Purpose Computation on Graphics Hardware, *Computer Graphics Forum*, Vol.26, No.1, pp. 80–113 (2007).
2) nVIDIA Corporation: CUDA Programming Guide Version 1.1 (2007). http://developer.nvidia.com/cuda/.
3) Harish, P. and Narayanan, P. J.: Accelerating Large Graph Algorithms on the GPU using CUDA, *Proc. 14th Int'l Conf. High Performance Computing (HiPC'07)*, pp. 197–208 (2007).
4) 9th DIMACS implementation challange - Shortest paths. http://www.dis.uniroma1.it/~challenge9/download.shtml.
5) Bader, D. A. and Madduri, K.: GTgraph, http://www.cc.gatech.edu/~kamesh/GTgraph/.
6) Chakrabarti, D., Zhan, Y. and Faloutsos, C.: R-MAT: A Recursive Model for Graph Mining, *Proc. 4th SIAM Int'l Conf. Data Mining (SDM'04)*, pp. 197–208 (2004). CD-ROM (5 pages).