Vol.2013-DBS-158 No.10
2013/11/26

GPU スレッド生成手法切り替えによる SSSP 探索の高速化

鈴木順† 菅真樹† 林佑樹† 吉川隆士†

グラフの 1 ノードから他の全てのノードへの最短距離と経路を求める SSSP(Single-Source Shortest Path)探索は, その処理の並列性の高さから GPU を用いて計算が高速化できることが知られている．SSSP 探索の計算は，グラフの始点ノードから全ノードへの距離情報が収束するまで計算を繰り返すことにより実現するが，情報が更新されるノード数は反復処理の各回で大きく異なる．従来手法では更新ノード数が少ない回でもグラフの全ノード分の GPU スレッドを生成するが，特に大規模なグラフでは更新処理を行わない無駄なスレッドが多数発生し，スレッド生成のオーバヘッドが増大する課題があった．本稿では，更新ノード数が少ない回は更新が行われるノードを計算する GPU スレッドだけを生成し，更新ノード数が多い回は更新ノードをカウントするオーバヘッドを避けるためにカウントを行わずにグラフの全ノード分のスレッドを生成する手法を提案する．これにより，143 万ノードの関東地図に対する SSSP 計算において従来手法より処理性能を 18%向上した．

Accelerating SSSP Algorithm on GPU with Adaptive Thread Creation

JUN SUZUKI† MASAKI KAN†
YUKI HAYASHI† TAKASHI YOSHIKAWA†

Single-Source Shortest Path (SSSP) algorithm that calculates minimum costs of paths from a single node of a graph to all the other nodes is known to be accelerated using GPUs, because of its applicability to parallel calculation. SSSP algorithm is processed by iterating updating costs of all nodes from a source node until they converge. Here, the numbers of nodes that are updated differ depending on each phase of iteration. Conventional methods create GPU threads for all nodes even in phases in which the number of updated nodes is small. This increases the overhead of creating threads especially in SSSP calculation of large graphs. In this paper, we propose a method to enhance the performance of SSSP algorithm by creating GPU threads only for the nodes of which costs are updated when the number of updated nodes is small, while creating threads for all nodes when the number of updated nodes is large to avoid the overhead of counting large number of updated nodes. The proposed method increased the performance of SSSP algorithm by 18% when a map of the Kanto Region of Japan is processed.

1. はじめに

グラフの 1 ノードから他の全てのノードへの最短距離と経路を求める SSSP(Single-Source Shortest Path)探索は, グラフにおける基本処理の 1 つとして，ナビゲーション，バイオ情報，ソーシャルネットワーク等の様々な分野に応用されている．また SSSP 探索の計算は，グラフの始点ノードから全ノードへの距離情報の更新を並列で行うため，並列処理によって高速化が可能であり，GPU(Graphics Processing Unit)の適用が有効であることが知られている．

GPU を用いた最初の SSSP 計算は，Harish らによって提案された[1]．この手法では，SSSP 計算の繰り返し処理の各回(以下では Phase と呼ぶ)において，計算対象となるグラフのノード数分の GPU スレッドを生成する. 生成されたスレッドは，担当する各ノードの距離情報が更新されたか確認する．そして更新されていた場合は担当ノードに隣接するノードの次の更新値を計算する．また更新がなければ何も行わない．より具体的には，反復計算の前に始点ノードの距離を 0 とし，その他のノードの距離を無限大として初期化する．そしてそれに続く反復計算において，始点ノードから近い順に各ノードの距離と経路が計算されていく. つまり無限大に初期化した各ノードの距離が始点ノードから近い順に実際の正しい距離に更新される．このようなグラフの全ノード分のスレッドを生成する実装方法は，その後に発表された他の研究でも採用された[1,2,3].

ここでこのように各 Phase において全ノード数分のスレッドを生成する手法では, SSSP 計算全体において生成される GPU スレッドの総数がグラフ規模に対し非線形に増加する．なぜなら，反復処理の各 Phase で生成されるスレッド数がグラフ規模(ノード数)に比例して増加するだけでなく，全てのノードの距離情報が収束するまで行われる反復計算の回数もグラフ規模に依存して増加するためである．

ところが，これらの SSSP 計算で生成される GPU スレッドの中で，意味がある計算を行うスレッドは一部に限定される．このため，意味がある計算を行わない無駄なスレッドを生成するためのコストが SSSP 計算の性能オーバヘッドとなる．今，これを説明するために各ノードが接続する他のノードの平均数(平均次数)が小さいグラフに対して SSSP 計算を適用する場合を考える. このようなグラフの一例は，あるノードと接続する他のノードが地理的に隣接す

† NEC グリーンプラットフォーム研究所
NEC Corporation

るノードに限定される地図データである．このようなグラフの SSSP 計算では，グラフの平均次数が低いため，反復処理の各 Phase において距離情報が伝搬するノード数が限定される．その結果 1 つの計算 Phase 内で情報が更新されるノード数が少なくなる．特に SSSP 計算の各 Phase において距離情報が更新されるノード数がグラフのノード数に比例して増加しない場合，グラフ規模が拡大するほど無駄なスレッドを生成するオーバヘッドが全体の計算時間に占める割合が大きくなる．

ここで，これらの無駄なスレッドを生成するためのオーバヘッドを削減するために，意味のある計算を行うスレッドだけを生成する手法が考えられるが，その手法を単純に SSSP 計算に適用することはできない．確かに近年，情報が更新されるノードを計算するスレッドだけを生成し，計算時間を削減する手法が幅優先探索(BFS: Breadth First Search)で提案された[5, 6]．ここで BFS 計算ではグラフの始点ノードと接続する全てのノードの全探索を行う．この計算では一度情報が更新されたノードが再び評価されることはない．一方，SSSP 計算にはノードの距離情報の再更新が頻繁に発生するという違いがある．これは，各ノードにおける始点ノードからの最短距離は，ホップ数ではなく始点からそのノードまでの経路を構成するエッジ距離の合計値で決まるためである．このため SSSP 計算では BFS 計算と比較して反復処理において情報が更新されるノード数が増加する．その理由により SSSP 計算で無駄なスレッドの削減を行う場合，更新ノード数が多い Phase において，更新ノードを GPU 内でカウントするためのオーバヘッドが増加し，計算時間が従来手法より逆に遅くなる．

そこで本稿では，SSSP 計算の反復処理の各 Phase において距離情報が更新されるノード数にばらつきがあることに着目する．そして，更新ノード数が少ない Phase では更新が行われるノードを計算する GPU スレッドだけを生成することで計算時間を削減し，更新ノード数が多い Phase では更新ノードをカウントするオーバヘッドを避けるため，グラフの全ノード分のスレッドを生成する手法を提案する．ここで，これら 2 つのスレッド生成法の切り替えの指標には更新ノード数を用いる必要があるが，提案手法では更新ノードが多い Phase では更新ノード数のカウントを行わない．そのため，提案手法ではホストで測定した各 Phase の GPU の処理時間を測定することで，スレッド生成手法の切り替えの指標として用いる．これにより，更新ノードが多い Phase では従来手法と同じ計算性能を実現する．以上の方法により，提案手法では全体として従来手法より SSSP 計算を高速化する．

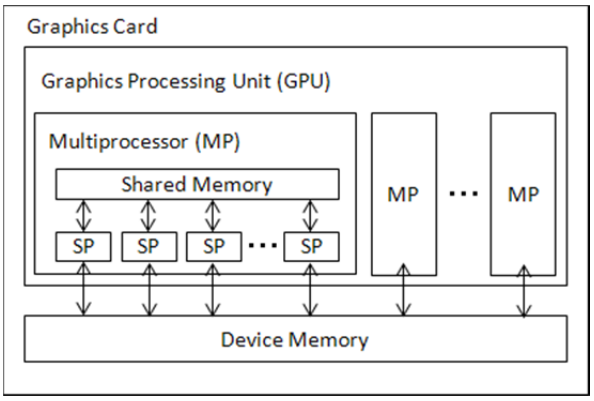


図 1 GPU アーキテクチャ

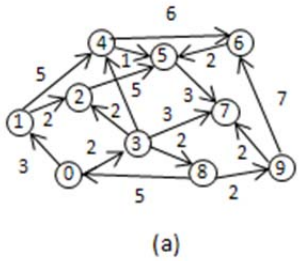
以下本稿では，第 2 節で GPU を用いたプログラミングについて述べる．第 3 節では GPU を用いた SSSP 計算の従来手法について述べる．第 4 節では，地図データに対し SSSP 計算を行った場合の処理の特性について述べる．第 5 節では GPU スレッドの生成手法を切り替える提案手法について述べる．第 6 節では地図データの SSSP 計算に提案手法を適用した実験結果を述べる．最後に第 7 節でまとめる．

2. CUDA による GPU プログラミング

GPU のプログラミングには CUDA(Compute Unified Device Architecture)が利用できる．CUDA はグラフィクスのパイプライン処理をプログラマに意識させることなく，GPU で動作する汎用プログラムを書くことを可能にするフレームワークである[7].

CUDA を用いると GPU 用のプログラムを SPMD(Single Program Multiple Data)型計算機のプログラムとして作成することができる．GPU が実行するプログラムはカーネル関数と呼ばれ，GPU 上に同一のカーネル関数を実行する多数の GPU スレッドが生成される．図 1 に GPU の一般的なアーキテクチャを示す．GPU は複数のストリーミングマルチプロセッサ(SM)を保持し，各 SM は複数のストリーミングプロセッサ(SP)を持つ．GPU で動作するスレッドはブロックと呼ぶグループに分けられる．同一ブロック内のスレッドは同一の SP に割り当てられる．そして各ブロック内のスレッドは Warp と呼ぶ単位のグループに分割され，各 Warp は SP において SIMD(Single Instruction Multiple Data)で実行される．現在のハードウェアでは Warp は 32 スレッドである．

GPU を用いて SSSP 計算を行う場合，反復処理の各 Phase におけるグラフノードの更新を GPU に生成されるスレッドが並列で行う．ここで次の Phase において情報が更新されるノードを計算するスレッドだけを生成する場合，その数を現在の Phase で並列に動作しているスレッド間で協調してカウントする必要がある．これは，GPU のデバイスメモリ上に定義した領域に対し，スレッド間の排他処理を用いることにより実現する．従って次の Phase で情報が更新



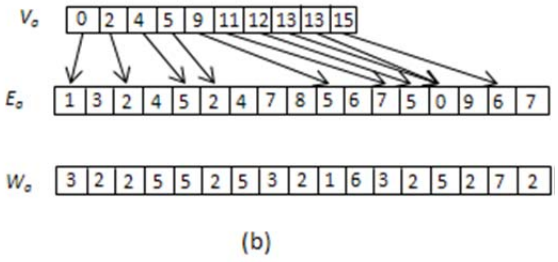


図 2 (1) グラフ. (2) 隣接リスト

されるノード数が多い場合，排他処理の増加により現在の Phase の計算時間が増加する．

3. 従来の SSSP 計算

初めに従来の SSSP 計算で用いるデータ構造を示す．従来手法では計算対象となるグラフ $G(V, E)$を隣接リストを用いて表現し GPU のデバイスメモリに格納する. 例として図 2(a)に示すグラフの隣接リスト表現を図 2(b)に示す．リスト V_a はグラフのノードの情報を示し要素数は$|V|$である．また，リスト E_a はエッジの情報を示し要素数は$|E|$である．リスト V_a の要素 $V_a[v]$はリスト E_a へのインデックスである．リスト E_a の $V_a[v]$から $V_a[v+1]$-1 の要素はノード v と接続するノードの番号である．また，リスト W_a の要素はリスト E_a の要素に対応するエッジの距離を保持する．

次に Harish らが提案した GPU を用いた SSSP 計算の疑似コードをアルゴリズム 1-3 に示す[1]．アルゴリズム 1 は CPU で実行され，2 と 3 は CPU から呼ばれ GPU でカーネル関数として実行される. 従来手法の SSSP 計算では図 2(b)に示した V_a, E_a, W_a に加え，反復計算の個々の Phase で各ノードの距離情報が変更されたかを示すリスト M_a, 各ノードの始点ノードからの距離を示すリスト C_a, 各ノードの距離の更新値を示すリスト U_a を用いる．

アルゴリズム 1 では与えられたグラフ情報 $G(V, E, W)$からリスト V_a, E_a, W_a, M_a, C_a, U_a を作成する．次にリスト C_a と U_a の要素を無限大に初期化する．また，始点ノードの距離は 0 とし，リスト M_a の始点の要素を true とする．5-10 行目は SSSP 計算の反復処理である．全ノードの距離情報が収束するまで計算を繰り返す．収束するまで M_a は empty(全ての要素が false)とならない．反復計算の各 Phase では，全てのグラフノードに対し距離情報の更新を行うスレッドを生成する．生成されるスレッドは，距離情報の計算を行うアルゴリズム 2 に示すカーネル関数 RELAX_KERNEL()を実行するスレッドと, 距離情報の更新を行うアルゴリズム 3 に示すカーネル関数 UPDATE_KERNEL()を実行するスレッドの 2 種類である．

アルゴリズム 2 では，各スレッドが担当するグラフのノードの距離が前回の計算 Phase で更新されたか 2 行目の $M_a[v]$の値から判断する．更新されていた場合，担当するノードと接続するすべてのノードについて接続ノードの距離の更新値を計算する. またその計算結果をリスト U_a の接続ノードが対応する要素に格納する．ここであるスレッドが更新値を計算した接続ノードについて別のスレッドでも更新値を計算している場合がある．これは接続ノードがグラフの複数のノードに接続している可能性があるためである．従って正しい計算結果を得るためにアルゴリズム 2 の 6 行目の処理はアトミックに行われる必要がある．また，SSSP 計算において距離に関する C_a と U_a の 2 つのリストを用いる理由はデータの不整合を防ぐためである. SSSP 計算では，計算を行う各スレッドが反復処理の各 Phase の開始時のリ

アルゴリズム 1. SSSP_ALGORITHM($G(V, E, W)$, s) // s is a source vertex

```
1:  Create V_a, E_a, W_a, M_a, C_a, and U_a
2:  Initialize C_a := ∞ and U_a := ∞ for all v∈V
3:  C_a[s] := 0
4:  M_a[s] := true
5:  while M_a not empty do
6:    for each vertex v∈V in parallel do
7:      invoke RELAX_KERNEL(V_a, E_a, W_a, M_a, C_a, U_a)
8:      invoke UPDATE_KERNEL(V_a, E_a, W_a, M_a, C_a, U_a)
9:    end for
10: end while
```

アルゴリズム 2. RELAX_KERNEL(V_a, E_a, W_a, M_a, C_a, U_a)

```
1:  v := threadID
2:  if M_a[v] = true then
3:    M_a[v] := false
4:    for i := V_a[v] to V_a[v+1] − 1 do
5:      n := E_a[i]
6:      U_a[n] := min( U_a[n], C_a[v] + W_a[n] )
7:    end for
8:  end if
```

アルゴリズム 3. UPDATE_KERNEL(V_a, E_a, W_a, M_a, C_a, U_a)

```
1:    v ← threadID
2:    if C_a[v] > U_a[v] then
3:        C_a[v] ← U_a[v]
4:        M_a[v] ← true
5:    end if
6:    U_a[v] ← C_a[v]
```

表 1 地図グラフの概要

ノード数	1427876
エッジ数	4584822
ノード平均次数	3.2
エッジ平均距離	83666
エッジ距離標準偏差	184384

スト C_a を参照して計算を行う必要がある．ここであるスレッドが計算した結果を直接 C_a に反映した場合，別のスレッドが参照する値が Phase 開始時の値ではなく同一 Phase を計算する別のスレッドに書き換えられた値となる可能性がある．このデータの不整合を防ぐため，各スレッドの計算結果を一度 U_a に格納する．

次にアルゴリズム 3 では U_a の値を C_a に反映する．具体的には，U_a に格納された各ノードの距離の更新値が，C_a に格納された現在のノードの距離より小さければ更新値を用いて距離を更新し，更新したノードに対応するリスト M_a の要素を true とする．

ここで本節では SSSP 計算においてグラフの始点ノードから各ノードへの最短距離だけを求める計算手法を示したが，最短経路を合わせて求めることも可能である．その場合，各ノードの最短距離を与えた接続ノードを記憶すれば良い．あるノードから始点ノードまでの最短経路は，各ノードが記憶している最短距離を与える接続ノードを始点まで辿ることで求められる．

以上のように従来手法では，SSSP 計算の反復処理の各 Phase においてグラフノード数分のスレッドを GPU に生成する．生成されたスレッドは，計算を担当するノードに対応する M_a の要素が true であれば隣接ノードの距離の更新という意味ある計算を行うが，false であれば何も行わずに終了する．

4. 地図データにおける SSSP 計算

本稿の第 6 節で述べる実験では SSSP 計算の対象として地図データを用いる．本節では用いた地図データに対する SSSP 計算が示す特性について議論する．提案手法による高速化でもこの特性を利用する．

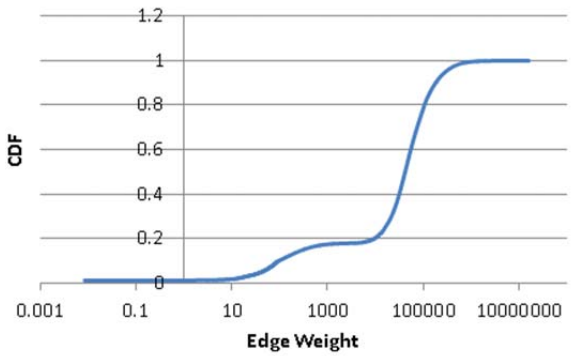


図 3 エッジの距離の累積分布関数

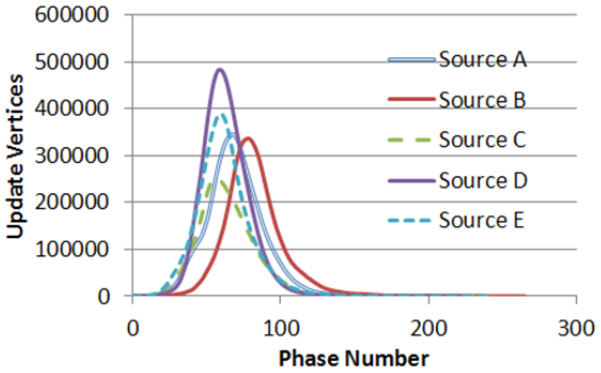


図 4 SSSP 計算の各 Phase の更新ノード数

今回用いたデータはデジタル道路地図データベース[8]及び国土数値情報（鉄道）製品仕様書 第 1.1 版[9]から関東地方の道路と鉄道網を合わせて合成した地図である．都道府県では東京，神奈川，埼玉，千葉，茨城が含まれる．道路は人が歩くために必要となる時間を距離とした．

合成した地図グラフの概要を表 1 に示す．ノード数はおよそ 143 万である．ノードの平均次数は 3.2 でありスパースなグラフと言える．注目すべき特徴はエッジの距離の標準偏差が大きいことである．図 3 にエッジの距離に関する累積分布関数(CDF: Cumulative Distribution Function)を示す．横軸は対数表示である．エッジの平均距離は 83666 だが，各エッジの距離は 10 から 100 万までばらつきがある．このようなグラフを対象とする SSSP 計算では，各経路のホップ数と距離の間の相関関係が小さいため，ホップ数が多い経路による各ノードの距離の再計算が発生し，計算量が増加することが予想される．

図 4 はランダムに抽出したグラフ上の始点ノード A～E の 5 つについて，グラフ上の他の全ての点への最短距離を求める SSSP 計算を行った場合に，反復計算の各 Phase で距離情報が更新されるノード数を示している．横軸の Phase には通し番号を付与した．始点ノードによって計算の反復回数は異なるが，どのノードも概ね 230 回で計算が収束する．図 4 を参照すると，5 つのノード全てにおいて，一部の Phase で距離情報が更新されるノード数が多く，他の

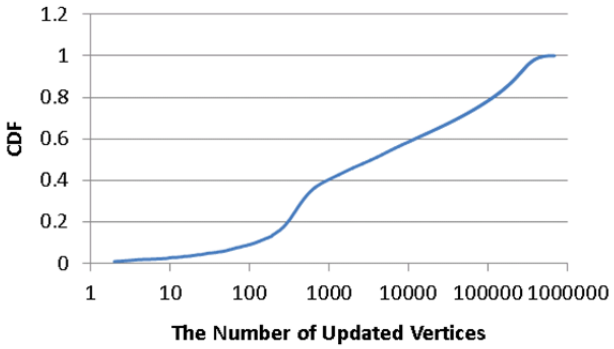


図 5 更新ノード数の累積分布関数

Phase では少ないことがわかる. 特に SSSP 計算の後半では, 距離情報が更新されるノードが 10000 以下とピーク値と比較して 1/10 以下である.

また図 5 に，ランダムに抽出した 1000 ノードの始点に対し, SSSP 計算を行った場合の各 Phase における更新ノード数の CDF を示す. 横軸は対数表示である. 更新ノード数が多い Phase では，50 万を超えるノード情報が更新される一方，過半数(59%)では情報が更新されるノードは 10000 以下である.

このように本稿で用いる地図グラフを例とするグラフデータは, 平均次数が低くエッジの距離のばらつきが大きい. また，このようなグラフに対する SSSP 計算では，反復計算の各 Phase で情報が更新されるノード数に大きな偏りがある. 更新されるノード数は，一部の Phase で多いが，過半数の Phase では更新ノード数のピーク値と比較して 1/10 以下と小さい. 提案手法ではこの特性を考慮して SSSP 計算の高速化を実現する.

5. 提案手法

SSSP 計算では, 反復計算の各 Phase において全ノード分に対応するスレッドを生成する従来手法に対し，距離情報の更新が行われるノードだけにスレッドを生成しスレッド生成のオーバヘッドを削減する手法は従来手法より計算が遅くなる. これは第 4 節で述べたように，地図データを例とした SSSP 計算では反復処理の各 Phase 間で更新ノード数に偏りがあり，更新ノードが多い Phase で更新ノードを数えるための GPU での排他処理のオーバヘッドが増大するためである.

そこで提案手法では，反復処理の各 Phase において更新ノード数が少ない場合は更新ノード数分のスレッドを生成し，計算時間を削減する. 一方更新ノード数が多い場合は更新ノードをカウントするオーバヘッドを避けるため，全ノード分のスレッドを生成する. 更新ノード数が少ない Phase では GPU の排他処理を用いても更新ノードをカウントするオーバヘッドが小さく，全ノード分のスレッドを生成する場合より計算時間を短縮できる. 一方，更新ノードが多い Phase では更新ノードのカウントを行わないため, 2 つのスレッド生成手法の切り替えのために代替となる指標が必要となる. そこで提案手法では，ホストで測定した SSSP 計算の各 Phase の GPU での処理時間をスレッド切り替えの指標として用いる. これにより，更新ノード数が多い Phase では従来手法と同じ計算性能を実現する. そして全体として，SSSP 計算を高速化する.

提案手法の GPU を制御するホストプログラムをアルゴリズム 4 に示す. C_a, U_a, C_a, M_a の初期化はアルゴリズム 1

アルゴリズム 4. SSSP_PROPOSAL($G(V, E, W)$, s)

```
1:  Create V_a, E_a, W_a, M_a, C_a, U_a, NW_a, and NR_a
2:  Initialize C_a := ∞ and U_a := ∞ for all v∈V
3:  C_a[s] := 0
4:  M_a[s] := true and add s to NR_a
5:  sch := true
6:  while M_a not empty do
7:    if sch = true then
8:      start := current time
9:      for each vertex v in NR_a in parallel do
10:       invoke SELECT_RELAX_KERNEL(V_a, E_a, W_a, M_a,
            NW_a, NR_a, C_a, U_a)
11:     end for
12:     NR_a := NW_a and NW_a := empty
13:     for each vertex v in NR_a in parallel do
14:       invoke SELECT_UPDATE_KERNEL(V_a, E_a, W_a, M_a,
            NR_a, C_a, U_a)
15:     end for
16:     end := current time
17:     if end − start > THR then
18:       sch := false
19:     end if
20:   else
21:     start := current time
22:     for each vertex v∈V in parallel do
23:       invoke RELAX_KERNEL(V_a, E_a, W_a, M_a, C_a, U_a)
24:       invoke UPDATE_KERNEL(V_a, E_a, W_a, M_a, C_a, U_a)
25:     end for
26:     end := current time
27:     if end − start < THR then
28:       sch := true
29:       invoke SWITCH_KERNEL(M_a, NR_a)
30:     end if
31:   end if
32: end while
```

アルゴリズム 5. SELECT_RELAX_KERNEL(V_a, E_a, W_a, M_a, NW_a, NR_a, C_a, U_a)

```
1:    x ≔ threadID
2:    v ≔ NR_a[x]
3:    if M_a[v] = true then
4:      M_a[v] ≔ false
5:      for i ≔ V_a[v] to V_a[v+1] − 1 do
6:        n ≔ E_a[i]
7:        U_a[n] ≔ min( U_a[n], C_a[v] + W_a[n] )
8:        if U_a[n] = C_a[v] + W_a[n] then
9:          if n is not scheduled then
10:           add n to NW_a
11:         end if
12:       end if
13:     end for
14:   end if
```

に示した従来手法と同じである．NR_a は反復処理の前の計算 Phase で距離情報が更新されたノードのノード番号を格納するリストである． *sch* はスレッドの生成手法を示すパラメータである．*sch*=true であれば更新されるノードだけにスレッドを生成し，false であれば全ノード分のスレッドを生成する．5 行目以降が SSSP 計算の反復処理である．各 Phase では GPU 命令の前後で時刻を測定し GPU の処理時間を算出する．この処理時間を 17 行目と 27 行目においてでしきい値と比較し，スレッド生成手法の切り替えを行う．更新されるノード分のスレッドのみを生成する 9 行目からの処理では，NR_a に更新ノード情報が格納されているため，このノード数分のスレッドを GPU に生成する．また，全ノード分のスレッドを生成する 22 行目以降の処理は，アルゴリズム 1 の従来手法と同じである．ただし，全ノード分のスレッドを生成する処理では，オーバヘッド削減のため更新ノードをカウントする処理を行わない．そのため，27 行目の比較において全ノード分のスレッドを生成した Phase から更新ノード分のスレッドを生成する Phase に移行する場合， 29 行目で移行前に M_a から NR_a を生成する処理を行う．

アルゴリズム 5 に示す SELECT_RELAX_KERNEL()はアルゴリズム 4 の 9 行目において更新ノード分のスレッドを生成する際に呼ばれる GPU のカーネル関数である．カーネル関数を実行する各スレッドは，2 行目において NR_a に格納されている担当ノードの番号をリードする．そして 6 行目以降で担当するノードとエッジで接続する全ての隣接ノードに対し，隣接ノードの距離の更新値を求める．次に求めた更新値が前の Phase までの隣接ノードの距離の値より小さい場合，隣接ノードを NW_a に登録する．ただし，隣接ノードがすでに NW_a に登録されていた場合，重複して登録はしない．この SELECT_RELAX_KERNEL()の完了後，ホストで実行されるアルゴリズム 4 では NW_a を NR_a にコピーし，NW_a を初期化する．これは NW_a は書き込み用であり，NR_a が読み込み用のためである．

アルゴリズム 6. SELECT_UPDATE_KERNEL(V_a, E_a, W_a, M_a, NR_a, C_a, U_a)

```
1:    x ≔ threadID
2:    v ≔ NR_a[x]
3:    if C_a[v] > U_a[v] then
4:      C_a[v] ≔ U_a[v]
5:      M_a[v] ≔ true
6:    end if
```

続いて，アルゴリズム 6 に示す SELECT_UPDATE_KERNEL()のカーネル関数が実行される．このカーネル関数を実行するスレッドでは，担当するノードの距離の更新値が現在のノードの距離よりも小さければ更新値を用いて距離の更新を行う．また距離を更新したノードに対応する M_a の要素をセットする．

以上により，提案手法では更新ノードが少ない Phase において計算を行わない無駄なスレッドを生成するために必要なオーバヘッドを削減し，同時に更新ノードが多い Phase において更新ノードをカウントするためのオーバヘッドを避けることにより，SSSP 計算の高速化を実現する．

6. 実験結果

本節では第 4 節で議論した関東地区の 143 万ノードの地図グラフに対し，提案手法を適用した結果を報告する．実験に用いた GPU のスペックを表 2 に示す．

図 6 はある 1 つの始点ノードに対し SSSP 計算を行った結果である．SSSP 計算の反復処理の各 Phase 毎に計算時間を測定した．ここでしきい値におけるスレッド生成法の切り替えを安定して行うため，しきい値に対する計算時間の比較にはその Phase と前回の Phase の計算時間の平均を用いた．また，しきい値はスレッド生成を更新ノード分から全ノード分に切り替える値と，全ノード分から更新ノード分に切り替える値の 2 種類を個別に設定できるようし計算性能に対しチューニングを行った．本測定では前者を 350us，後者を 330us とした．青い点線が提案手法(Proposal)の計算性能である．また比較のため，グラフの全てのノード分のスレッドを生成する Conventional と、全ての Phase において更新ノードだけにスレッドを生成する Selection も測定した．また，Updated Nodes は反復計算の各 Phase で距

表 2 GPU のスペック

モデル	Tesla K20c
マルチプロセッサ(MP)	13
MP あたりのコア数	192
メモリ	4800MB

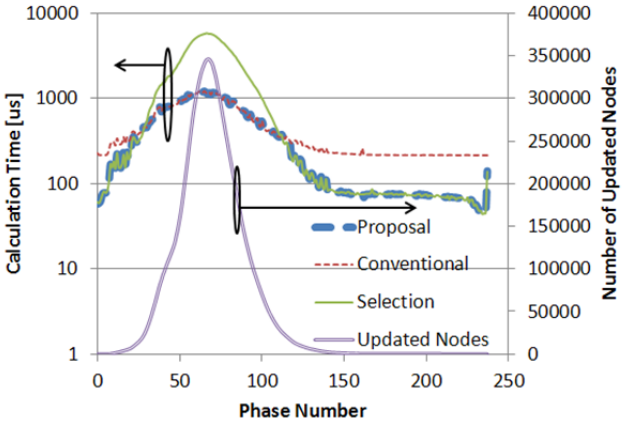


図 6 SSSP 計算の各 Phase の計算時間と更新ノード数

離情報が更新されるノード数を示している．計算時間の縦軸は対数表示である．提案方式の計算時間を参照すると，グラフの更新ノード数が少ない Phase において無駄な GPU スレッドの生成を行わないことで従来手法より計算時間が削減されていることがわかる．一方，グラフの更新ノード数が多い場合，更新ノード数分のスレッドを生成する Selection では，更新ノードをカウントするための GPU での排他処理のオーバヘッドが増大し，従来手法より計算性能が悪化している．これに対し提案手法では，更新ノードが多い Phase では更新ノード数をカウントしない従来手法に切り替えることで，従来手法と同じ性能を実現した．

次に提案手法と従来手法の SSSP 計算の性能比較を図 7 に示す．図 6 における各計算 Phase の計算時間の和に相当する．計算性能の測定はグラフ規模に対する提案手法の効果を見積もるため，31 万ノードと 143 万ノードのグラフについて行った．31 万ノードのグラフは 143 万ノードのグラフから幹線や主だった道路を抽出することで作成した．測定した値はランダムに選択した 1000 ノードを始点とする SSSP 計算の平均時間である．図 7 が示すように，31 万ノードでは従来手法と提案手法に性能差が見られなかった．一方 143 万ノードでは 18%の性能改善が見られた．これは 31 万ノードのグラフでは，143 万ノードのグラフと比較して計算を行わない無駄なスレッドを生成するオーバヘッドが全体の計算時間に占める割合が小さいためだと予想される．従って，無駄なスレッドを削減することにより得られる高速化の効果が小さい．よって，提案手法は大規模なグラフに対する SSSP 計算の高速化により効果的であること

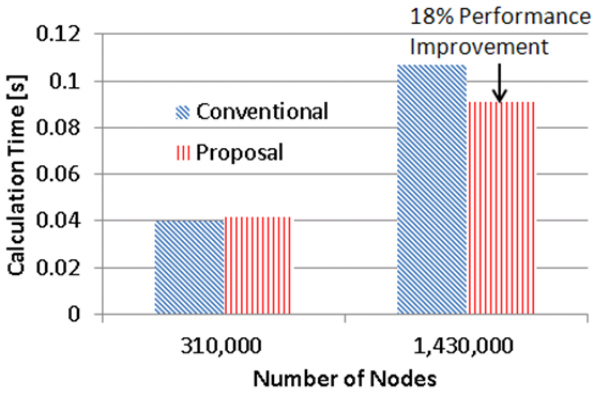


図 7 提案手法と従来手法の SSSP 計算性能の比較

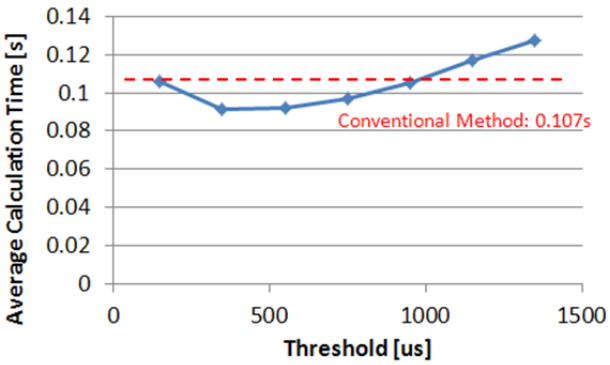


図 8 しきい値を変化させた場合の提案手法の性能

が期待される． また，Selection の計算時間は 31 万ノードで 0.06s，143 万ノードで 0.251s であり，従来手法および提案手法の性能より低かった．

次に図 8 にスレッド生成法の切り替えしきい値を変化させたときの提案手法の計算性能を示す．ここで従来手法の計算時間は 0.107s であり，提案手法で高速化が得られるためのしきい値の設定マージンは 800us 強となる．これは測定点の中で最良の性能が得られたしきい値が 350us であったことを考えると，その値の 2 倍以上の範囲であり，ある程度大きいと言える．

以上の実験結果より，提案手法では SSSP 計算の反復処理において更新ノードが少ない Phase で無駄なスレッドの生成を削減し従来手法に対して計算を高速化した．また，更新ノードの多い Phase では，更新ノード増加時の更新ノードカウントによる排他処理のオーバヘッドを回避し，従来手法と同じ計算性能を実現した．このため 2 種類のスレッド生成法の切り替えの指標には、GPU の排他処理を必要とする更新ノードのカウント値ではなくホストで測定された GPU の処理時間を用いた．これらの手法の結果，143 万ノードのグラフに対し SSSP 計算全体として従来手法より 18%の性能向上を実現した．

7. まとめ

従来の GPU による SSSP 計算では，計算全体で生成される GPU スレッドの総数がグラフ規模の増加により非線形に増加した．ところが地図グラフなどのスパースなグラフに対する SSSP 計算では，生成されたスレッドの大半がノードの距離情報の更新を行わずに終了する意味のないスレッドであり，その生成コストが計算性能のオーバヘッドとなっていた．しかし，このオーバヘッドを解決するため距離情報が更新されるノードにだけスレッドを生成する手法を単純に適用することはできなかった．これは SSSP 計算の更新ノード数が多い Phase では，GPU 内で排他処理を用いて更新ノード数をカウントするオーバヘッドが大きくなるため, SSSP 計算全体として計算性能が従来手法より逆に遅くなるためである．

提案手法では SSSP 計算の各 Phase で更新されるノード数に偏りがあることに着目した．そして，更新ノードが少ない Phase では更新ノードを計算するスレッドだけを生成することで無駄なスレッドを生成するオーバヘッドを削減し計算の高速化を行った．また更新ノードが多い Phase では，更新ノードをカウントせず全ノード分のスレッドを生成することで更新ノードをカウントするためのオーバヘッドを回避した．これにより，この 2 つのスレッド生成法の切り替えに GPU の排他処理を用いた更新ノードのカウンタ値を用いることができなくなったため，ホストで測定した GPU の各 Phase の処理時間を代替の指標として用いた．これにより，更新ノードが多い Phase では従来手法と同じ性能を実現した．以上の手法により，SSSP 計算全体として 143 万ノードのグラフにおいて従来手法より 18%の高速化を実現した．また，提案手法はより大規模なグラフに対し効果が大きいと考えられる．

今後は関東地区以外の地図に対しても提案手法を適用し，手法の効果の一般性を確認する．また，複数の GPU を用いた手法を検討し，SSSP 計算をさらに高速化する．

謝辞　本研究の一部は、総務省の委託研究「「モノのインターネット」時代の通信規格の開発・実証」プロジェクトの成果である.

参考文献

1) P. Harish and P. J. Narayanan: Accelerating Large Graph Algorithms on the GPU Using CUDA, Proceedings of the 14th international conference on High performance computing (HiPC'07), pp.197-208 (2007).
2) T. Okuyama et al.: A Task Parallel Algorithm for Computing the Costs of All-Pairs Shortest Paths on the CUDA-compatible GPU, Proceeding of the 2008 International Symposium on Parallel and Distributed Processing with Applications, pp.284-291 (2008).
3) S. Kumar et al.: A Modified Parallel Approach to Single Source Shortest Path Problem for Massively Dense Graphs Using CUDA, Proceeding of the International Conference on Computer & Communication Technology (ICCCT)-2011, pp.635-639 (2011).
4) H. O. Arranz et al.: A New GPU-based Approach to the Shortest Path Problem, Proceeding of the International Conference on High Performance Computing and Simulation (HPCS 2013), (2013).
5) L. Luo et al.: An Effective GPU Implementation of Breadth-First Search, Proceeding of the 47th Design Automation Conference (DAC'10), pp52-55 (2010).
6) D. Merrill et al.: Scalable GPU Graph Traversal, Proceeding of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'12), pp117-128 (2012).
7) CUDA C Programming Guide: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
8) デジタル道路地図データベース, http://www.drm.jp/database/structure.html, 一般財団法人日本デジタル道路地図協会.
9) 国土数値情報（鉄道）製品仕様書 第 1.1 版, http://nlftp.mlit.go.jp/ksj/jpgis/product_spec/KS-PS-N02-v1_1.pdf, 国土交通省.