社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

GPUを用いた分割グラフ型データベースの高速化

森島　　信† 　松谷　宏紀†,††,†††

† 慶應義塾大学大学院 理工学研究科 〒 223-8522 神奈川県横浜市港北区日吉 3-14-1
†† 科学技術新興機構さきがけ
††† 国立情報学研究所
E-mail: †{morisima,matutani}@arc.ics.keio.ac.jp

あらまし　グラフ型データベースは、データをグラフ形式で蓄積し処理するデータベースである。グラフ型データベースはノード間の関係性を表現するのに向いているため、SNS(Social Networking Service) やソーシャルグラフを基にしたリコメンデーションエンジンへの応用が期待されている。ソーシャルグラフのような大規模なグラフの場合、データ量が多いため、複数の計算機にグラフデータベースを分割してグラフを保存する必要がある。分割されたグラフデータベースにおいてグラフの探索をする際、複数の計算機に跨って探索を行う必要があり、非常に計算時間が長くなる。そこで、本論文では、グラフ探索に必要最低限の情報のみをグラフデータベースから抽出し、一台の計算機で扱える規模のキャッシュを作成する。さらに、そのキャッシュされたグラフに対して GPU で計算を行い、グラフ探索を高速化する。データの抽出により、キャッシュのデータサイズは元のデータに対して 1/139 から 1/136 となった。1000 万ノード、次数 200 のグラフに対する Dijkstra 法において、GPU による実装は CPU に対して 2.6 倍の高速化に成功した。

キーワード　構造型ストレージ、グラフ型データベース、GPU

GPU-Based Acceleration on Partitioned Graph Databases

Shin MORISHIMA† and Hiroki MATSUTANI†,††,†††

† Graduate School of Science and Technology, Keio University 3-14-1, Hiyoshi, Yokohama, JAPAN 223-8522
†† PRESTO, Japan Science and Technology Agency
††† National Institute of Informatics
E-mail: †{morisima,matutani}@arc.ics.keio.ac.jp

Abstract　Graph databases use graph structures to store data sets as nodes, edges, and properties. They are used to store and search the relationships between a large number of nodes, such as social networking services and recommendation engines that use customer social graphs. Social graphs are too large to store the graph databases of a single computer. So they must be partitioned to store the graphs. If a graph search is performed in the partitioned graph databases, it is performed across multiple coputers. In this case, the calculation time is very long. In this work, we extract the minimum information to search graph from partitioned graph databases and make a cache using the information. Because the cache size is smaller than original graph databases, the cache can be stored a single computer. Further, we accelerate a graph search in the graph stored the cathe using GPUs. The data size of cathe is from 1/139 to 1/136 with respect to oritinal graph databases.In Dijkstra algorithm for the graph that have 10 million nodes and 200 degree, GPU implementation can speed up 2.6x with respect to CPU implementation.

Key words　Structured storage, graph database, GPUs

1. はじめに

今日、情報通信技術と各種センシング技術の発展にともない、膨大な情報 (ビッグデータ) が生成されている。これらの情報は主にデータベースとして蓄積されるが、現在最もよく使われているデータベースシステムである RDBMS(Relational DataBase Management System) は、汎用性が高く豊富な機能を持つものの、スケーラビリティはそれほど高くない。そのた

め、構造型ストレージと呼ばれる、一つの機能のみに特化しているが、スケーラビリティに優れるデータベースが近年注目されている。

本論文で扱うグラフ型データベースは構造型ストレージの一種であり、データをグラフ形式で蓄積し処理するデータベースである。このデータ構造は、ノード間の関係性を表現するのに向いており、SNS(Social Networking Service) やソーシャルグラフを基にしたリコメンデーションエンジンへの応用が期待されている。

代表的な SNS である Facebook のノード数 (ユーザ数) は数億、次数はおよそ 200 である [4]。このように、SNS やソーシャルグラフのグラフの規模は大きく、このようなグラフの場合、一台の計算機のグラフデータベースにグラフ全体を保存することは不可能であり、複数の計算機にグラフデータベースを分割してグラフを保存する必要がある。グラフデータベースをグラフ分割アルゴリズムを用いて分割することが可能であることは、Alex らによって示されている [6]。

分割されたグラフデータベースにおいてグラフの探索をする際、複数の計算機に跨って探索を行う必要があり、非常に計算時間が長くなる。そこで、本論文では、グラフ探索に必要最低限の情報のみをグラフデータベースから抽出し、一台の計算機で扱える規模のキャッシュを作成する。さらに、そのキャッシュされたグラフに対して CUDA を用いて GPU で計算を行い、グラフ探索を高速化する。対象とするグラフ探索アルゴリズムは Dijkstra 法である。

本論文の構成は以下の通りである。2 章は本論文の関連研究を述べる。3 章は対象とするグラフ型データベースの詳細を述べる。4 章はグラフ型データベースのキャッシュについて述べる。5 章はキャッシュに対する GPU による高速化手法を述べる。6 章はキャッシュのデータサイズと実行時間の評価を述べる。7 章で本論文をまとめる。

2. 関 連 研 究

2.1 GPU によるグラフ処理アルゴリズムの高速化

Ortega-Arranz らが GPU を用いた Dijkstra 法の高速化を行った [1]。Dijkstra 法は、ある始点から他のノードへの最短経路を求めるアルゴリズムである。通常の Dijkstra 法では、まだ探索していないノードのうち始点からの距離が最短である 1 つのノードを選択して探索を行う。Ortega-Arranz らは、条件を満たす複数のノードを同時に選択し、並列して探索を行うことで、高速化している。

ノード x の始点からの距離を $d(x)$、ノード x と隣接する辺の重みの最小値を $E(x)$ とし、まだ探索を行っていないノードの集合を U とする。このとき、始点からの距離が次式 1 の Δ よりも小さいノードを並列に探索する。

$$\Delta = min(d(u) + E(u)) : u \in U \quad (1)$$

すなわち、並列に探索されるノード v が満たす条件は $d(v) \leqq \Delta$ である。

この方法で GPU の並列性を活かしたことで、GPU による実装は CPU に対して 13 倍から 220 倍の性能向上に成功した。本論文では、この手法を基に Dijkstra 法の GPU による実装を行う。

本論文では対象にしないが、Dijkstra 法以外のグラフ処理アルゴリズムも GPU による高速化が行われている。Merrill らはグラフ探索アルゴリズムの一つである幅優先探索の高速化を GPU で行い、CPU に対して最大 29 倍の性能向上に成功している [2]。また、Nobart らはグラフの最小スパニングツリーを求めるアルゴリズムの高速化を GPU で行い、CPU に対して最大 14 倍の性能向上に成功している [3]。

2.2 グラフ型データベースの分割

Alex らがグラフ型データベースの一つである Neo4j を対象に、グラフ分割アルゴリズムをグラフデータベースに適応することが可能であることを示した [6]。彼らは三つのグラフ分割アルゴリズムが Neo4j に適応可能であると示し、それらのアルゴリズムはグラフデータベースを使用しながら合間に実行可能であると示した。これらの分割アルゴリズムによって、分割されたグラフ間を跨るトラフィックを、ランダムに分割する場合に比べて最大 90%削減に成功した。本論文では、対象のグラフ型データベースとして彼らの提案手法によって分割された Neo4j を想定する。

2.3 グラフ型データベースの GPU による高速化

我々の過去の研究で、小規模なグラフ型データベースの GPU による高速化を行った [7]。Facebook の次数分布を基にした 100,000 ノードのグラフにおいて、Dijkstra 法はオリジナルの Neo4j に対して 26.2 倍、A*法は 32.8 倍高速化に成功した。本論文では、以前は対象にしなかった大規模で複数台に分割されたグラフ型データベースを対象とする。

3. グラフ型データベース

本論文で、高速化の対象とするグラフ型データベースは Neo4j である。Neo4j は Java 言語で実装されたオープンソースかつ代表的なグラフ型データベースの 1 つである。3.1 節から 3.5 節では Neo4j に実装されているグラフ探索アルゴリズムについて説明する。

3.1 Dijkstra 法

Dijkstra 法は、2 章で述べた通り、重み付きグラフにおいてある始点からの最短経路を求めるアルゴリズムである。始点から重みの小さい順に探索を行い、最短経路が判明しているノードを増やしていくことで全てのノードへの最短経路を求める。

3.2 A*法

A*法は、Dijkstra 法を改良したアルゴリズムである。それぞれのノードに目的ノードまでの重みの推定値を与え、その推定値が小さい順に探索を行う。

3.3 Shortest Path

Shortest Path はある 2 ノード間のホップ数が最小となる経路を全て求めるアルゴリズムである。始点からホップ数が少ない順に全ての経路を探索し、目的ノードに到達した時点で終了する。

3.4 All Path

All Path はある 2 ノード間で、設定したホップ数以下の経路を全て求めるアルゴリズムである。Shortest Path の終了条件をホップ数に変更したものと同じである。

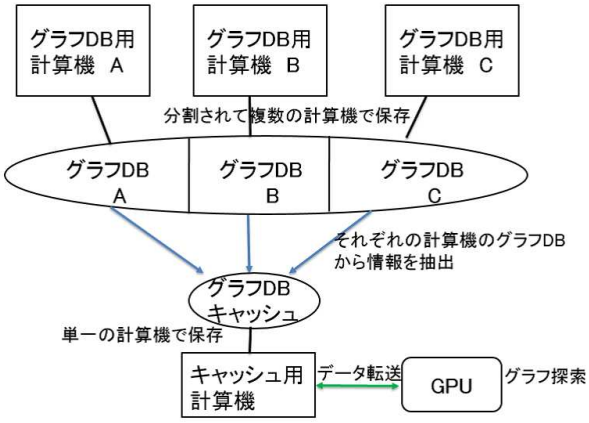

図 1　想定するシステム全体の概観

3.5　All Simple Path

All Simple Path は All Path で求まる経路のうち、同じノードが二回現れる経路を除外したものである。

A*法は Dijkstra 法を改良したものであるため、推定値に基づく点を除けば Dijkstra 法と同様の探索方法である。また、Shortest Path、 All Path、 All Simple Path は終了条件を除くと、全ての辺の重みを 1 にした Dijkstra 法と探索方法が似ている。このように、これらのアルゴリズムは全て Dijkstra 法を基にしていると言える。そのため、本論文では Dijkstra 法を対象とする。

4.　グラフ型データベースキャッシュの作成

4.1　想定するシステム全体像

図 1 に想定するシステム全体の概観を示す。グラフ型データベースの規模は 1 台の計算機に収まるものよりも大きいものを想定しており、図の上部に示す通りグラフ型データベース用の複数台の計算機に分割されて保存されている。この例では、3 台の計算機に分割されている。これらのデータは元々は一つのグラフ型データベースであるため、1 台の計算機に収まる大きさにデータを抽出することができれば、分割する必要はなくなる。グラフ探索に必要最低限の情報のみを抽出することで、データサイズを大幅に小さくすることができる。この抽出後の 1 台の計算機に収まる大きさのグラフを 1 台の計算機に保存し、グラフ型データベースのキャッシュとして利用する。これによりグラフ探索は複数台に跨って行う必要がなくなる。さらに、キャッシュ用の計算機に GPU を搭載し、GPU によってグラフ探索を行うことでグラフ探索を高速化する。

4.2　グラフ型データベースキャッシュのデータ構造

Neo4j は Java 言語で書かれており、ノードや辺に関する情報をまとめ、それらを様々な用途に使用できるよう、ノードや辺はクラスとして定義されている。このデータ構造は、グラフ型データベースの運用には適しているが、グラフ探索のみを考慮すると、冗長な情報が多い。

グラフ探索に必要な情報は、ノード ID、辺の両端、辺の重み、辺の方向であり、これらの情報のみを抽出する。また、単純に抽出するだけではなく、GPU による高速化を行うため、GPU の処理に適したデータ構造に変換を行う。必要最低限の情報を用いたデータ構造で主なものは、隣接行列と隣接リストである。隣接行列の情報量はノード数の 2 乗に比例し、隣接リストの情報量は辺の数に比例する。完全グラフのような一部のグラフを除き、隣接リストの方が情報量が小さくなるため、隣接リストを使用する。しかし、隣接リストの一般的な実装は、ポインタを多用しており、GPU による実装には向かない。そこで、GPU の処理に適した、配列を用いて隣接リストを表現する。

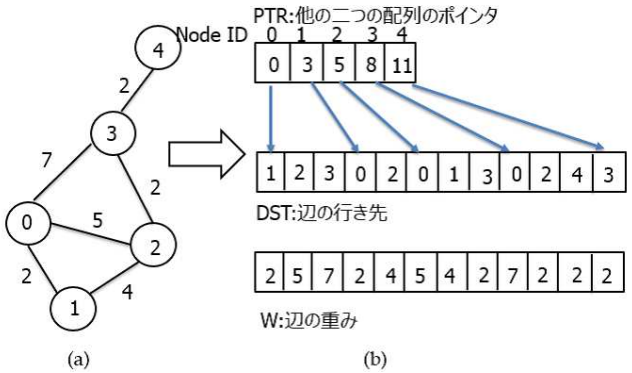

図 2　隣接リストの配列による表現

図 2 にグラフを隣接リストの配列表現で表現を示す。図 2(a) のグラフのノードの中の数字がノード ID、辺の横の数字が辺の重みを表す。辺の方向は全て双方向とする。図 2(b) は図 2(a) のグラフの隣接リストを三つの配列を用いて表現したものである。配列は、他の二つの配列のポインタの役割をする配列 PTR、各辺の行き先を表す配列 DST、各辺の重みを表す配列 W の三つである。各配列の要素の内容を以下に示す。

- PTR の n 番目の要素 $PTR[n]$ は n 番目のノード n までの累積次数を表す。すなわち、D_i をノード i の次数としたとき、$PTR[n] = \Sigma_{i=0}^{n} D_i$ である。
- p の範囲を $PTR[n] \leqq p < PTR[n+1]$ とするとき、$DST[p]$ はノード n を始点とする辺の終点を表す。
- $W[p]$ は $DST[p]$ に対応する辺の重みを表す。

例えば、図 2(b) において、ノード 0 はノード 1、ノード 2、ノード 3 への辺と隣接しており、それぞれ重みは 2、5、7 である。

このデータ構造を用いて GPU でグラフ探索を行ったのち、結果を Neo4j に返し、Neo4j で処理することで、Neo4j で直接実行した場合と同じ出力形式で結果が得られる。

5.　GPU によるグラフ探索の高速化

5.1　キャッシュされたグラフの分割

グラフ型データベースの情報を抽出し、一台の計算機にキャッシュするが、GPU のメモリ容量は計算機のメモリ容量やディスクの容量に比べて小さい。そのため、キャッシュしたグラフサイズが GPU のメモリ容量よりも大きい場合、グラフを GPU のメモリ容量よりも小さくなるように分割する必要がある。

グラフ探索をする際、あるノードに隣接する辺を全て参照する操作を多用するため、各区分にはあるノードに隣接する辺は同じ区画に含まれるように分割するのが望ましい。上述した配列を用いたデータ構造では、各ノードに隣接する辺がまとまって格納されており、配列 PTR によって管理されているので、

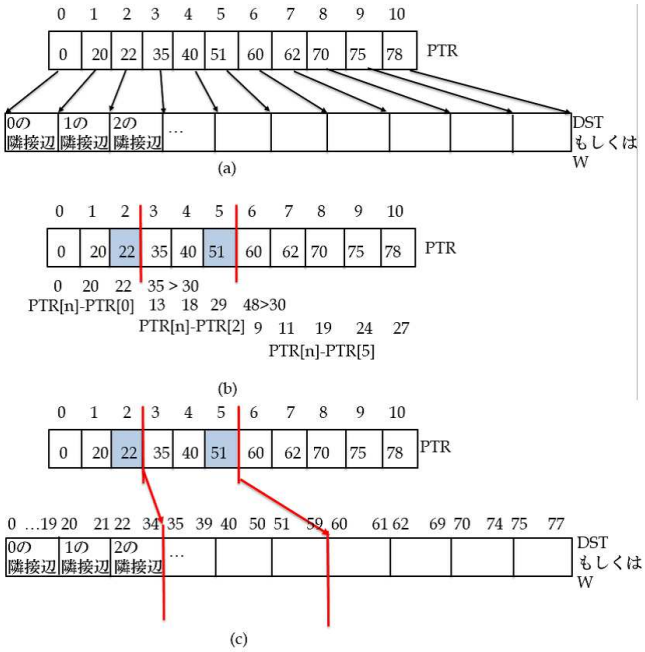

図 3　グラフ分割の例

配列 PTR を基に他の二つの配列を分割することで、あるノードに隣接する辺を全て含む分割は容易に実装できる。また、配列 PTR の要素の値は、グラフの累積次数を表しているため、配列 PTR の値を調べることで分割後のグラフサイズを求めることができる。そのため、GPU のメモリ容量に収まる大きさに設定してグラフを分割することが可能である。

図 3 に分割の例を示す。グラフのノード数は 10 であり、分割後に各グラフの辺の数が 30 以下にするという条件で分割を行う。配列 DST と配列 W は同様に分割するため、図には配列 PTR と配列 DST あるいは W のどちらかに相当する配列を示す。図 3(a) は分割前の配列を示している。まず、図 3(b) のように、配列 PTR の値を調べ、分割後のグラフが条件を満たすかどうかを確かめる。この例では、各区画の辺の数を 30 以下にするため、各区画の累積次数が 30 を越えたら、その一つ前の要素を分割の基準としており、2 番と 5 番の要素が分割の基準となる。次に、図 3(c) のように、配列 DST と配列 W を PTR の指す場所を基に分割する。この場合各区画の辺の数は、22、29、27 となり、条件である辺の数が 30 以下を満たす。

また、この分割方法ならば、分割後の配列を別に作成して保存する必要はなく、GPU に転送する際に、分割後の範囲に区切って転送すればよい。そのため、分割にかかる計算時間は分割箇所を決める計算時間のみでグラフ探索や GPU へのデータ転送の計算時間に比べて非常に短い。

5.2　分割グラフを用いた Dijkstra 法の実装

2 章で述べた Ortega-Arranz らの実装方法を、分割グラフに対して適応する。Algorithm1 にこの実装の擬似コードを示す。まず始めに、前節で述べたように配列 PTR を調べて分割箇所を決める。この操作は計算時間が GPU への転送時間よりも短いため、CPU で行う。

次に 2 章で述べた並列に探索する対象を決定するための式 1 で表される Δ を求めるため、カーネル delta は、未探索の各ノードに対して、ノードの重みとその隣接辺の重みの最小値の和を求める。カーネル delta の実装の擬似コードを Algorithm2 に示す。カーネル delta では、i 番目の分割グラフ G_i に対して探索を行っている。GPU には配列 DST と配列 W の一部しか

Algorithm 1 分割グラフに対する Dijkstra 法の実装 <<< *kernel* 名 >>> は GPU カーネル呼び出し

```
配列 PTR を調べて分割箇所を決める。分割数を n とする。
while 1 do
  for i = 0 to n do
    分割グラフ G_i を転送 (CPU → GPU)
    <<< delta >>>
  end for
  Δ =<<< min >>>
  if Δ = ∞ then
    break
  end if
  for i = 0 to n do
    分割グラフ G_i を転送 (CPU → GPU)
    <<< update >>>
  end for
end while
```

Algorithm 2 delta カーネル

```
s ← 分割グラフの配列 PTR の始まりの索引番号
e ← 分割グラフの配列 PTR の終わりの索引番号
c[j] ← ノード j の始点からの距離
Δ_j ← ノード j における Δ の候補、初期値は ∞
W_i ← 分割グラフの G_i の配列 W
j = スレッド ID + ブロック ID × ブロックの大きさ + s
while j < e do
  if c[j] ≠ ∞ かつ j は未探索のノード then
    for k = PTR[j]toPTR[j + 1] do
      d_j = min(d_j, W_i[k − PTR[s])
    end for
    d_j = d_j + c[j]
  end if
  j+ = ブロックの大きさ × ブロック数
end while
```

Algorithm 3 update カーネル

```
s ← 分割グラフの配列 PTR の始まりの索引番号
e ← 分割グラフの配列 PTR の終わりの索引番号
c[j] ← ノード j の始点からの距離
DST_i ← 分割グラフの G_i の配列 DST
W_i ← 分割グラフの G_i の配列 W
j = スレッド ID + ブロック ID × ブロックの大きさ + s
while j < e do
  if c[j] ≦ Δ then
    j を探索済みノードにする
    for k = PTR[j]toPTR[j + 1] do
      BEGINATOMICREAGION
      min(c[DST[k − PTR[s]]], c[j] + W[DST[k − PTR[s]]])
      ENDATOMICREAGION
    end for
  end if
  j+ = ブロックの大きさ × ブロック数
end while
```

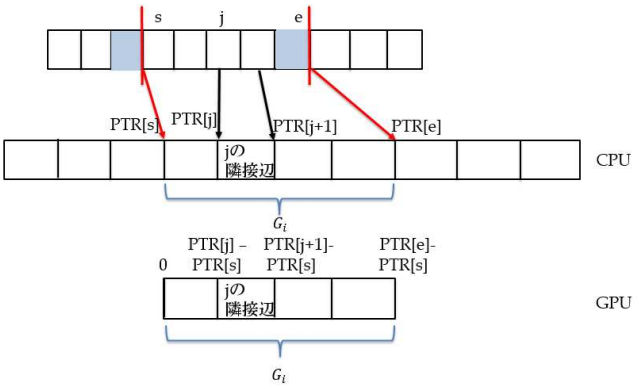

図 4　分割後のグラフの索引番号

送らないため、この 2 つの配列の索引番号が配列 PTR の値と異なることに注意が必要である。その関係を図 4 に示す。この図では、一つ目の配列が配列 PTR、二つ目の配列が CPU における配列 DST もしくは配列 W、三つ目の配列が GPU における配列 DST もしくは配列 W を表す。配列 PTR の s と e の間の部分が分割後のグラフ G_i である。CPU においては、配列 DST と配列 W の全て記憶されているが、GPU の場合、G_i に相当する部分のみしか記憶していない。そのため、配列の索引番号が分割前のグラフとずれることになる。図に示す通り、そのズレ幅は $PTR[s]$ の値に等しく、分割後のグラフで配列 DST と配列 W を参照する時は元の索引番号から $PTR[s]$ を引いた値を用いる。配列 PTR に関しては、配列の大きさが配列 DST や配列 W に比べて小さいため、分割を行わずに全体を転送し、GPU で全体を保持する。

カーネル delta で各ノードに対して探索を行った後は、その結果の最小値をカーネル min で求める。カーネル min は、単純に最小値を求める問題であり、最小値を求めるのに一般的なリダクション演算を用いているため、カーネルの詳細説明は省略する。カーネル min の結果を CPU に転送し、その結果が ∞ の場合、ループから抜けだし、アルゴリズムは終了する。

そうでない場合、Δ の値を用いて、カーネル update で各ノードの重みを更新する。カーネル update の擬似コードを Algorithm3 に示す。カーネル update はカーネル delta と同様に、分割グラフ G_i に対して実行する。あるノードの重みが Δ の値よりも小さい場合、そのノードと隣接するノードの重みを更新する。並列に更新を行うため、一つのノードの重みを同時に二つのスレッドから変更する可能性がある。この競合を避けるために、この操作は各ノード毎の不可分操作とする。異なるノードに対する更新は同時に行うことができるため、不可分操作にすることによる計算時間の増加は少ない。

カーネル update で更新を終えた後は、while 文の始まりに戻り、再び Δ の計算を行う。全てのノードに対して探索を終える、もしくは始点からのパスが通っていないノードのみが未探索ノードの場合 Δ の値は ∞ になり、アルゴリズム 1 の break 文によってループから抜けだし、アルゴリズムが終了する。

6. 評　　価

6.1 対象グラフ

対象グラフには、ランダムグラフを用いた。代表的な SNS である Facebook の平均次数がおよそ 200 [4] であるため、ランダムグラフの平均次数は 200 とした。辺の数の合計がノード数 × 200 となるまで辺を作成することでグラフを作成した。メモリ使用量の評価では、ノード数は 50,000 ノードから 150,000 ノードまで 50,000 ノードずつ変化させて評価を行い、キャッシュしたデータに対する計算時間の評価ではノード数を $2^{20} \times 1$ から $2^{20} \times 10$ まで 2^{20} ノードずつ変化させて評価を行った。

6.2 評価環境

評価に用いた CPU は Intel Xeon E5-1620 v2 であり、動作周波数は 3.7GHz、CPU コア数 8、メモリ容量は 128GB、プログラミング言語は C 言語を用いた。GPU の評価では NVIDIA GeForce GTX 780 Ti を用いた。GPU の主な性能は表 1 に示す。

表 1　GPU の主な性能

性能項目	GeForce GTX 780 Ti
コア数	2,880
コアクロック	875MHz
メモリクロック	1,750MHz
メモリバス幅	384bit
メモリバンド幅	336GB/s
メモリ容量	3GB

6.3 Neo4j とキャッシュのメモリ使用量の比較

Neo4j で作成したデータを抽出してキャッシュすることでメモリ使用量をどの程度削減出来るか調べるため、Neo4j でグラフを作成したときのメモリ使用量とデータを抽出して作成したキャッシュのメモリ使用量を比較した。

表 2　Neo4j とキャッシュのメモリ使用量

ノード数	Neo4j	キャッシュ
50,000	10630.8MB	76.3MB
100,000	20935.9MB	152.6MB
150,000	31186.5MB	228.9MB

表 2 に平均次数 200、ノード数 50,000 から 150,000 のグラフの Neo4j とキャッシュのメモリ使用量を示す。Neo4j に対しては、グラフ作成に必要最低限の情報しか与えずに評価した。そのため、実際の運用で様々な属性がノードや辺に対して与えられると、Neo4j のメモリ使用量はさらに大きくなる。

表 2 に示した通り、キャッシュのメモリ使用量は Neo4j に比べて大幅に削減され、$\frac{1}{139}$ から $\frac{1}{136}$ になった。この大幅なメモリ使用量の削減により、複数台の計算機に保存されている Neo4j のグラフを一台の計算機に収まる大きさにすることが可能である。

6.4 グラフ探索の実行時間

6.4.1 CPU と GPU の実行時間の比較

キャッシュに対するグラフ探索として、Dijkstra 法を GPU と CPU で実行し、その実行時間を比較した。また GPU においては、GPU のメモリ容量は GPU の種類によって異なることを考慮して分割後のグラフサイズを変えて評価を行った。GPU は 5. 章で述べた実装方法を用い、CPU は一般的な隣接リストに対する Dijkstra 法の実装方法である優先度付きキューを用いた方法を用いた。

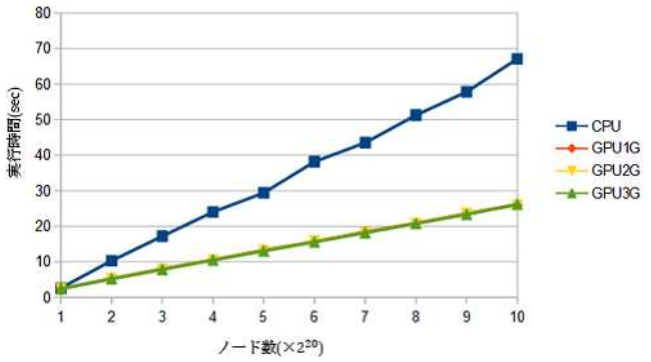

図 5　CPU と GPU の実行時間

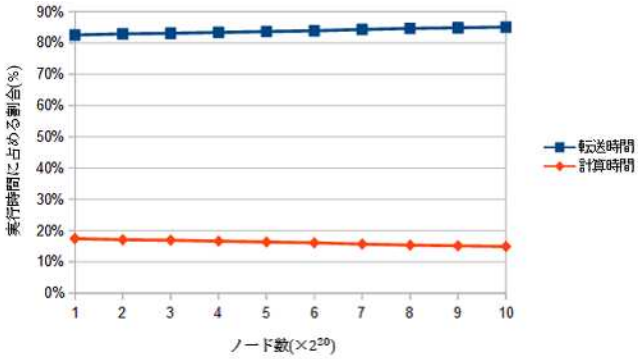

図 6　実行時間に占める計算時間と転送時間の割合

図 5 に、次数 200、ノード数が $2^{20} \times 1$ から $2^{20} \times 10$ までのグラフのキャッシュに対する Dijkstra 法の実行時間を示す。CPU は CPU の実行時間、GPU1G、GPU2G、GPU3G はそれぞれ GPU の実行時間で、グラフサイズが 1G、2G、3G の時間である。ただし、2^{20} ノードのグラフはおよそ 1.6GB であるため、2GB と 3GB の GPU メモリの評価では、分割する必要がない。そのため、5. 章で示したアルゴリズムのようにループ毎にグラフの転送を行う必要はないが、今回は比較のためループ毎の転送を行った。

図 5 をみると、GPU の 3 つの評価結果は重なっており、分割後のグラフサイズは実行時間に影響を与えないことがわかる。転送するデータ量が多い場合、グラフの転送にかかる時間は転送の回数ではなく、合計転送量に比例するためである。

CPU と GPU の実行時間を比較すると、ノード数 $2^{20} \times 1$ の時を除いて、GPU のほうが高速である。$2^{20} \times 1$ の時に CPU が高速な理由は、キャッシュを有効に使えているためだと考えられる。また、ノード数 $2^{20} \times 2$ 以上では、ノード数が増えるにつれ、GPU による高速化率が増加している。$2^{20} \times 2$ の時は 2.0 倍であるのに対し、$2^{20} \times 10$ の時は 2.6 倍であり、図 5 の中で最大の高速化率である。よって、GPU による高速化はグラフの規模が大きいほど効果が大きいと考えられる。

6.4.2　GPU の実行時間の内訳

GPU の実行時間には、CPU-GPU 間のデータ転送と GPU での計算時間の二つの時間が含まれる。この節では、グラフサイズ 3GB の時のこれらの割合を示す。

図 6 に、実行時間に占める計算時間と転送時間の割合を示す。図 6 を見ると、実行時間に占める割合の大半はデータ転送時間の方である。これは、並列 Dijkstra 法の性質によるものである。並列 Dijkstra 法は、条件に合うノードを並列に探索するが、各ループ毎に条件に合うノードの数が異なるため、計算時間がループ毎に変化する。しかし、グラフの転送時間はループ毎に同時間である。そのため、計算量が小さいループでは転送時間が実行時間の大半を占め、これが全体の実行時間の割合に影響を与えていると考えられる。例えば、1 回目のループは条件を満たすノードは始点 1 つであり、計算量は非常に小さくなり、実行時間にほぼ全てをデータ転送時間が占める。ノード数が増えるにつれ転送時間の割合がわずかに増加しているが、それは 1 回目のループのようにノード数が増えても並列度が変わらないループがあるためと考えられる。

1 回目のループのように、転送時間の割合が極端に大きなループは GPU に転送せずに CPU で実行を行い、計算量が多い場合のみ GPU に転送して計算を行うことで、更なる高速化ができると考えられる。

7.　結　　論

複数台の計算機に分割して保存されているグラフ型データベースからグラフ探索に必要最低限の情報のみを抽出し、キャッシュを作成することで、データサイズを $\frac{1}{139}$ から $\frac{1}{136}$ にすることに成功した。この大幅にデータ量の削減により、大規模なグラフを 1 台の計算機に収めて計算することが可能になる。さらに、このキャッシュに対する探索を GPU によって高速化した。次数 200、ノード数 $2^{20} \times 10$ のグラフに対する Dijkstra 法において、CPU に対して 2.6 倍の高速化に成功した。

謝辞　本研究の一部は、JST 戦略的創造推進事業さきがけ「多様な構造型ストレージ技術を統合可能な再構成可能データベース技術」の補助による。

文　　献

[1] Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos and Arturo Gonzalez-Escribano “A New GPU-based Approach to the Shortest Path Problem” High Performance Computing and Simulation, pp.505-511, July 2013.

[2] Duane Merill, Michael Garland and Andrew Grimshaw “Scalable GPU Graph Traversal” Principles and Practice of Parallel Programming, pp.117-128, August 2012.

[3] Sadegh Nobari Thanh-Tung Cao Stéphane Bressan Panagiotis Karras “Scalable Parallel Minimum Spanning Forest Computation” Principles and Practice of Parallel Programming, pp.205-214, August 2012.

[4] Johan Ugander, Brian Karrer, Lars BackStrom and Cameron Marlow, “The Anatomy of the Facebook Social Graph” arXiv prepring arXiv:1111.4503 November 2011.

[5] Ian Robinson, Jim Webbem and Emil Eifrem “Graph Databases”, Oreilly, 2013.

[6] Alex Averbuch and Martin Neumann, “Partitioning Graph Databases - A Quantitative Evaluation”,arXiv1301.5121, 2013

[7] Shin Morishima and Hiroki Matsutani, “Performance Evaluations of Graph Database using CUDA and OpenMP-Compatible Libraries”, International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, June, 2014