Spark の GPGPU を用いたグラフ処理高速化方法の提案と評価

M2013SE002 稲本 裕貴

指導教員 青山 幹雄

1. はじめに

グラフ構造のビッグデータを高速処理するための並列分散処理の必要が高まっている．このため Apache Hadoop[7]に対してインメモリ型の Apache Spark[1,5,10]が提案された．また，並列分散処理に GPGPU を用いることで高速化を実現する提案がある．しかし，Spark を GPGPU で高速化する具体的な提案はされていない．本稿では，Spark の GPGPU を用いた大規模グラフ処理の高速化方法を提案する．

2. 研究課題

(1) 大規模グラフ処理

Spark と GPU という異なる並列計算環境を組み合わせた大規模グラフに対する処理方法の具体的な提案はされておらず，高速化の効果が明らかでない．

(2) メモリ上のデータ割り当て

RDD オブジェクトは Device Memory 上で処理ができない．また，Device Memory は記憶領域が Host Memory よりも少ないので大規模データを Device Memory 上に置くことができない．

3. 関連研究

3.1. Apache Spark

Apache Spark(以下 Spark)は Scala ベースのインメモリ型並列分散処理オープンソースフレームワークである．マスタワーカ型で実行し，複数ワーカ毎の Executer プロセスで並列処理を行う．抽象データセットである RDD (Resilient Distributed Datasets) [10]を実装している．

RDD はインメモリ型の大規模な分散クラスタのための抽象データセットである．Hadoop のような分散処理フレームワークでは反復アルゴリズムやインタラクティブなデータマイニング処理は非効率的であるため，インメモリで処理することで高速化する．

RDD は次の制約を満たしている．

(1) Immutable(Read Only)
(2) パーティションで分割可能
(3) なるべくインメモリで保持
(4) ストレージ上のデータもしくは他の RDD から生成
(5) 遅延評価

RDD はインメモリでフォールトトレラント性を確保するために，各 RDD の生成元データと生成時の処理内容のデータを生成している．このデータを用いることで RDD の一部が失われた時，RDD の再生成が可能である．

3.2. GPGPU

GPGPU はメニーコアの GPU を汎用計算に用いる並列計算技術である．GPU を用いるための環境として CUDA がある．CUDA では GPU が持つメモリを Device Memory と呼び，CPU が持つメモリを Host Memory と呼ぶ．

3.3. Spark による高速グラフ処理

Spark の RDD をグラフデータに対して拡張を行い，グラフ処理の高速化を実現した GraphX[8,9]がある．しかし，拡張されたデータは GPU での処理に適していない構造のため，GPU を用いた高速化ができない．

3.4. GPU を用いた MapReduce の高速化

Hadoop の MapReduce を GPU で高速化する提案がある[2,6]．しかし，インメモリ型の Spark に対して具体的な提案はされていない．

4. アプローチ

Spark のグラフ処理を GPGPU で高速化する．

しかし，GPGPU の Device Memory 上で Spark の RDD オブジェクトを扱えない．RDD の制約下で，Device Memory へコピーすることが可能なコレクションを生成する必要がある．RDD の Read Only であり分割処理可能である制約下で GPU を用いた処理を行うために，Device Memory 上のコレクションを Read Only である入力元データ，処理結果または中間データの 2 種類に分ける．これにより，RDD の制約を満たし GPU で RDD の処理が可能となる．この結果，RDD が Spark のプロセス並列処理に加えて，GPU を用いたスレッド並列処理が可能となり，処理の並列度を向上させ処理の高速化が可能となる．アプローチの図を図 1 に示す．

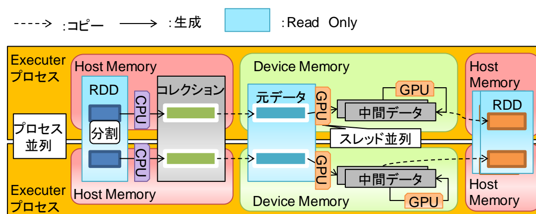

図 1 アプローチ

また，データ格納領域が少ない Device Memory を考慮し

てグラフのデータ構造はデータ量を削減しながら，GPU 処理に適したグラフ表現を用いる．

5. GPGPU を用いた Spark の高速化

5.1. GPGPU を用いた Spark アーキテクチャ

GPGPU を用いた並列高速化 Spark アーキテクチャの構成を図 2 に示す．

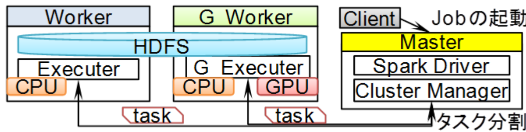

図 2 GPGPU を用いた Spark の構成

Worker 内の Executer プロセスで GPU を用いて高速処理を行う．GPU を用いる Worker は G_Worker，G_Worker 内の Executer を G_Executer と名付ける．GPU を Spark で用いるためには Host Memory 上の RDD オブジェクトを Device Memory にコピーする必要がある．しかし，RDD は GPU の Device Memory 上でデータ変換を行うことができない．そのため，メモリへのデータ割り当ての拡張を行う．

5.2. メモリへのデータ割り当ての拡張

Spark 実行時に Device Memory 上に配置するデータを不変データと可変データの 2 種類に分ける．これにより，RDD の Read Only であり分割処理可能である制約下で Device Memory 上にデータを配置することが可能である．2 種類のデータの利用方法を述べる．

(1) 不変データ

変換元データとして用いる．不変データの生成元は Host Memory 上の RDD である．不変データ生成の振る舞いを図 3 で示す．RDD は Device Memory にコピーすることができない．このため，RDD データから Device Memory へ転送可能であるコレクションを Host Memory 上に生成し，コレクションの大きさだけ Device Memory の領域を確保する．その後，コレクションを Device Memory へコピーする．Device Memory にコピーされたデータを不変データとする．

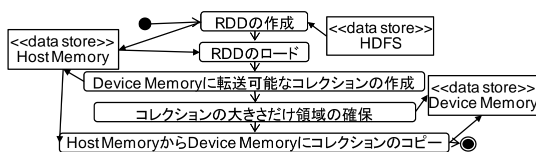

図 3 不変データの生成

(2) 可変データ

可変データを処理結果と中間データとして用いる．生成の振る舞いを図 4 に示す．可変データは Device Memory 上に生成される．事前にデータ格納領域を Device Memory 上に確保する．確保した Device Memory の領域に GPU を用いた処理結果を格納する．処理結果は Host Memory にコピーされ，RDD に変換される．

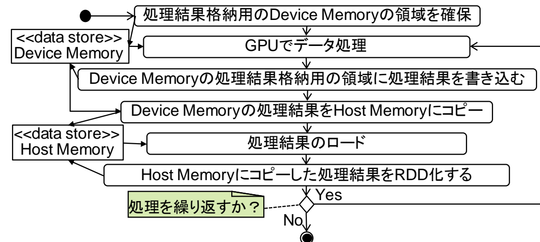

図 4 可変データの生成

5.3. GPU を用いたグラフ処理方法

GPU を用いた Spark のグラフ処理方法を述べる．

(1) グラフ構造

グラフには隣接リストと隣接行列で表現する方法がある．ポインタを用いる隣接リストでのグラフ表現では GPU で用いるには適していない．また，隣接行列での表現では GPU で扱うことは容易であるが，隣接リストよりもデータ量が増大する．そこで，Device Memory 上ではグラフをリスト表現でのデータ構造を配列で表現した隣接リスト配列[4]でグラフを表現する(図 5)．

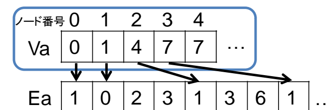

図 5 隣接リスト配列

(2) グラフ処理

グラフ処理では不変データはグラフに関するコレクションは隣接配列，各ノードの入出次数となる．このグラフのデータを用いてグラフ処理を行う．グラフ処理の結果を可変データとして生成し，Host Memory に転送され RDD として格納される．提案のグラフ処理の振る舞いを図 6 に示す．

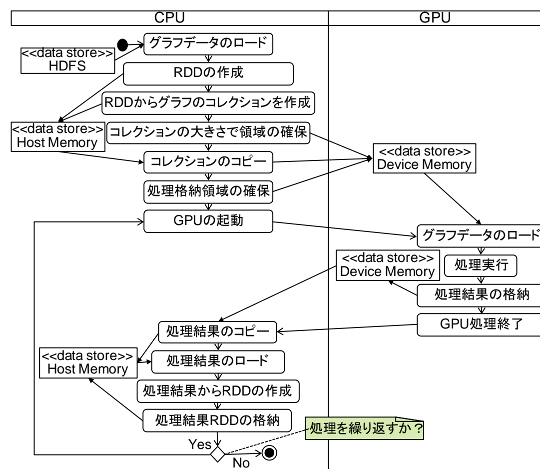

図 6 グラフ処理の振る舞い

6. プロトタイプの構築

提案の評価を行うためプロトタイプを構築した. 実行マシンの構成を表 1 に示す. 実行マシンは 1 台であるが, 複数コアを用いて擬似分散させる.

表 1 プロトタイプの構成

OS	Ubuntu 14.04
Spark	1.1.0
CUDA	6.5
CPU	Intel Corei5-4460(4Core,4Thread, Meory:16GB)
Graphic Board	NVIDIA GeForce GTX745 (CUDA Core:384Core, Device Memory:4GB)

例として用いるグラフ処理プログラムは静的ページランクとした. 実装は Python で GPU を用いるため PyCUDA API を用いた. 実装プログラムの規模は 171(LOC)である.

実装したプログラムの振る舞いを図 7 に示す. グラフのデータを不変データとして Device Memory へコピーする. 次に, 初期化用のコレクションを生成する. このコレクションは実数値の 1.0 がグラフのノードの個数だけ格納されている. 可変データとして contribution(以下 contrib.)とページランク格納領域を確保する. 2 つの領域に初期化用コレクションをコピーする. 次に GPU で contrib.を算出するカーネル関数を起動する. contrib.算出カーネル関数は対象ノードに入るエッジを持つノード(入ノード)のページランクを入ノードの出次数で割る処理を行う. その後, 残りの処理を行うカーネル関数を起動してページランクを得る. 処理結果であるページランクを Host Memory にコピーし, RDD にする. 繰り返し処理を行う場合は contrib.の初期化処理まで戻る.

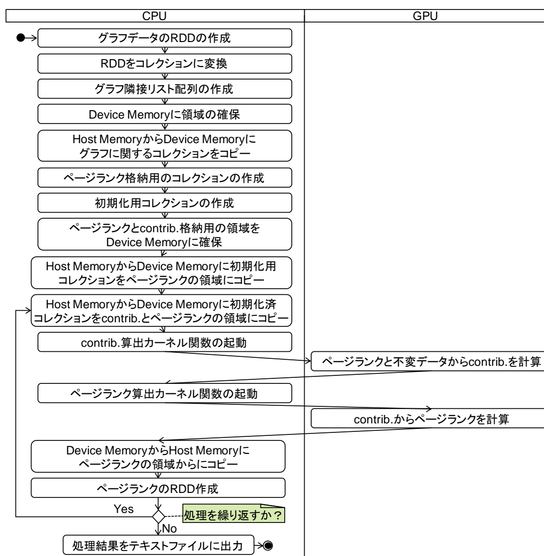

図 7 プロトタイプ上のページランク処理の振る舞い

7. 例題への適用と評価

7.1. 例題への適用

例題として, 静的ページランクプログラムを, Graph500[3] の Generator で生成した Kronecker Graph に適用した. 適用した Kronecker Graph のパラメータは SCALE4 から 23, edge factor は 16 とする. これにより生成されるグラフは 2 の SCALE 乗のノード数, (総エッジ数 / 総ノード数)が 16 の有向グラフが生成される.

7.2. 結果

各 SCALE の Kronecker Graph について, Spark 単体, GraphX, 提案実装でのページランクプログラムの実行時間を計測した. ページランクの繰り返し回数は 50 回行った. 実行時間のグラフを図 8 に示す. 計測不可の箇所がある. 理由として, CPU 利用では SCALE20 以降は 24 時間以上経過しても処理が終了せず, 複数回試しても同様の結果であったので計測不可とした. SCALE23 以降の GPU 利用での計測不可の原因はメモリ不足である.

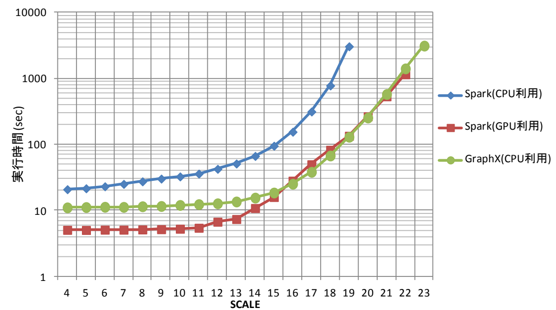

図 8 実行時間グラフ

図 9 は加速倍率を表したグラフである. 加速倍率は(CPU の Spark の実行時間 / GPU の Spark 実行時間または GraphX の実行時間)である.

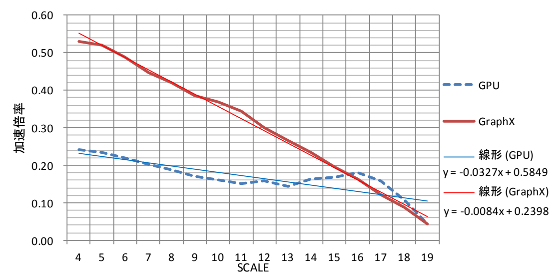

図 9 加速倍率グラフ

7.3. 評価

例題の条件では CPU 利用の Spark 実行と比較して, GPU 利用と GraphX 利用が同様に高速化された. 提案方法と GraphX はどちらも SCALE11 まで実行時間の増加が少ない. SCALE4 から SCALE14, SCALE21, SCALE22 で提案実装が最も高速に実行された. CPU 利用の Spark の加速倍率は, 一定の値になると予想したが,

提案方法と GraphX どちらもグラフ規模の増大に伴い変化の割合が増加した．GraphX は線形近似したグラフと同じ傾きとなった．提案方法は SCALE16 以降で近似グラフから逸脱した変化をしている．

8. 考察

8.1. GPU の効果に関する考察

Spark は RDD の処理を行う場合，複数のプロセスで並列計算を行う．ハードウェアの観点では，分割されたデータが並列に CPU による逐次処理を行うことである．提案方法では，GPU を用いることで Spark の RDD を細かい粒度，例えば配列の 1 要素レベルで処理の並列化による高速化が可能となった．GPU を用いることで，CPU を複数用いることなく処理の高速化を実現することが可能である．これにより，処理のレスポンス時間は用いる CPU の個数が少数でも高速化が実現可能と考える．

8.2. インメモリ処理の効果に関する考察

Spark は処理データをメモリ上にキャッシュすることで高速化を実現している．しかし，グラフ規模が大きくなるとメモリキャッシュ量が増大するためにインメモリ処理でボトルネックが生じた．SCALE18 のメモリ使用量の遷移のグラフ(図 10)を見ると，CPU 利用の Spark は他の実行よりも使用メモリ量が多い．また，520 秒以降使用メモリ量が減少している．ここではキャッシュされたメモリ解放を行っていることがコンソール上の実行ログで確認できた．ここから，使用メモリ量が一定の値になるとメモリ解放の時間がボトルネックになると考えられる．

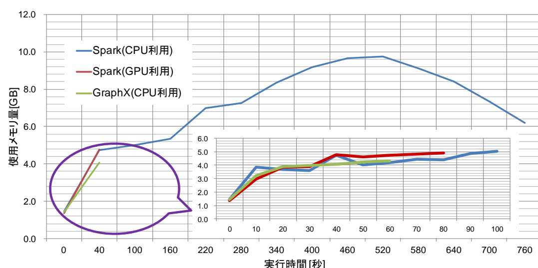

図 10 SCALE18 のメモリ使用量の遷移

また，提案方法では Device Memory に格納するデータ量の削減だけに注目していたが，メモリ解放の時間がボトルネックとなるのでアプリケーション実装時の Host Memory 上のデータ量も削減することが重要と考える．

8.3. グラフ生成アルゴリズムのスケーラビリティに関する考察

提案方法では RDD から GPU 処理可能であるコレクションに変換する．グラフ処理では RDD からグラフコレクションを生成する． SCALE11 まではグラフ規模が小さいため，提案方法，GraphX 共に実行時間の変化が少ない．GraphX の実行時間が提案方法よりも長い理由として，規模の小さなグラフでは GraphX の RDD を拡張したプロパティグラフを生成する時間が提案方法で使用するグラフコレクションの生成時間よりも長くなるためと考える．さらに，SCALE12 以降では提案方法のグラフコレクションを生成する時間がボトルネックになると考えられる．

今回実験対象としたページランク処理は提案方法では一部の計算の並列度をグラフのノード数まで向上可能であるために，ノード数が多いグラフの場合 GraphX と比べて実行時間が短縮されていると考える．

9. 今後の課題

Device Memory 上に格納できない大規模グラフに対しては，グラフを分割し複数 GPU を用いて並列処理することが効果的であると考えられるので，複数 GPU を用いた実行を考える必要がある．また，Device Memory の中で，高速に利用できるが容量の少ない Shared Memory などを用いる方法を考える必要がある．

10. まとめ

GPGPU を用いて Spark を高速化した．Device Memory 上のデータを RDD の制約下で 2 つに分ける．これにより RDD の制約を満たし GPU で RDD の処理が可能となり，Spark のプロセス並列処理に加えて，GPU を用いたスレッド並列処理を用いた高速化が可能となる．静的ページランクアルゴリズムで提案方法の評価を行い小規模のグラフでも高速化を実現した．さらに，大規模なグラフであるほど高速化の効果が増大することを確認した．今後の課題として複数 GPU の場合について考える必要がある．

参考文献

[1] Apache Software Foundation, Apache Spark, http://spark.apache.org/.

[2] H. Bingsheng, et al., Mars: a MapReduce Framework on Graphics Processors, PACT'08, ACM, pp. 260-269.

[3] Graph500, http://www.graph500.org/

[4] P. Harish, et al., Accelerating Large Graph Algorithms on the GPU Using CUDA, HiPC'07, pp. 197-208.

[5] H. Karau, Fast Data Processing with Spark, Packt Publishing, 2013.

[6] K. Shirahata, et al., Hybrid Map Task Scheduling for GPU-based Heterogeneous Clusters, IEEE CloudCom'10, pp. 733-740.

[7] T. White, et al., Hadoop, O'Reilly 2013.

[8] R. S. Xin, et al., GraphX: A Resilient Distributed Graph System on Spark, GRADES'13, ACM, pp. 2.

[9] R. S. Xin, et al., GraphX: Unifying Data-Parallel and Graph-Parallel Analytics, http://arxiv.org/abs/1402.2394.

[10] M. Zaharia, et al., Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-memory Cluster Computing, NSDI'12, USENIX Association, pp. 2-2.