Vol.2017-DBS-164 No.6
2017/1/17

ローカルおよびリモートGPUを用いたSpark RDD操作の高速化

大野 泰弘, 森島 信, 松谷 宏紀[1]

概要：Apache Spark はビッグデータ向けの分散処理フレームワークである。Spark の処理においては、インメモリ RDD がデータとして利用される。本論文では、GPU を用いて Spark RDD 操作の計算インテンシブ処理を高速化することを提案する。GPU で RDD を処理するために、RDD を配列データに変換し、その配列を GPU い転送し GPU で処理を行う。配列を作成する際、元の RDD の ID と作成された配列をホストメモリにキャッシュする。これにより同一 RDD のデータ変換オーバヘッドを削減することができる。また、転送したデータを GPU デバイスメモリで保持し続けることで、GPU へのデータ転送オーバヘッドを削減することができる。また、本論文では PCIe を通して 10Gbps Ethernet 越しに接続された GPU デバイスをリモート GPU として活用する。リモート GPU へのデータ転送オーバヘッドを削減するため、ローカル GPU とリモート GPU に対する RDD キャッシュポリシーも提案する。我々は、複数のリダクション操作と変換操作の GPU 処理を実装し、通常の Spark 操作と比較して最大 21.4 倍の性能向上を達成した。また、複数の RDD キャッシュポリシーによる性能向上率の比較も行った。

Performance Improvement of Spark RDD Operations Using Local and Remote GPUs

Yasuhiro Ohno, Shin Morishima, Hiroki Matsutani[1]

1. はじめに

近年、インターネットの利用拡大やモバイルデバイスの普及などにより扱われるデータ量が増大してきている。そのような大規模データセットを処理、解析するためにデータの分散処理フレームワークが注目されている。

Apache Hadoop[1] は代表的なオープンソースの分散処理フレームワークの一つである。Hadoop はストレージとして複数ノードのディスクにデータを分散保持する HDFS(Hadoop Distributed File System) を利用している。各プロセスにおいて、HDFS ベースでデータの読み書きを行うため、機械学習の様な同一データを複数回利用する処理ではディスクアクセスがボトルネックとなる。

Apache Spark[2] もオープンソースの分散処理フレームワークであるが、中間データを分散共有メモリに保持する点で Hadoop と異なる。Spark では、データセットは RDD(resilient distributed dataset)[3] というデータ形式で表現される。RDD は使用時にメモリに保持され、ディスクアクセスが最初と最後のデータの読み込み、書き込みのみのためディスクへのアクセスオーバヘッドが少ない。そのため、機械学習などの同一データに対する反復処理に有効である。

Spark RDD 処理の一部は計算インテンシブなため、本論文ではそれらの処理を GPU(Graphics Processing Unit) デバイスを用いて高速化する。ホストマシンと GPU の間のデータ転送がボトルネックとなりうるため、我々は各 RDD を GPU デバイスメモリにキャッシュしてデータ転送オーバヘッドを削減する。しかし、ホストマシンに直接接続可能な GPU(ローカル GPU) の数は限られるため、PCIe から 10Gbps イーサネット越しに接続された GPU(リモート GPU) を利用する。これにより、利用可能な GPU の数とデバイスメモリ容量を増やすことができる。一方で、リモート GPU へのデータ転送オーバヘッドはローカル GPU への転送オーバヘッドよりも大きい。ローカル GPU とリモート GPU を効率的に利用するため、利用頻度や転送サイズを考慮して、RDD を各 GPU に割り当てる必要がある。本論文では、ローカル GPU とリモート GPU に対して 3 つの RDD キャッシュポリシーを提案する。

本論文の構成は以下の通りである。まず、2 章にて、背景と関連研究を説明する。3 章にて、Spark RDD 処理の GPU オフローディングについて述べる。4 章にて、ローカルとリモート GPU に対する RDD キャッシュポリシーの詳細を説明する。5 章にて評価結果を示し、6 章にて本論文をまとめる。

2. 背景と関連研究

2.1 Spark 及び RDD の概要

Apach Spark はデータセットを RDD として扱う分散処理フレームワークである。主な言語は Scala である。中間データを RDD としてメモリ内に保持するため、機械学習の様な同一データを繰り返し使う処理に有用である。Spark では、ドライバノードがタスクやデータを複数のワーカーノードの割り当て、各ワーカーノードは与えられたデータを処理する。最終的にそれらの結果をドライバノードで統合し、処理を完了する。

RDD とは、Spark で採用されているデータ形式である。RDD は使用時にメモリに保持されるため、機械学習などの同一データに対する反復処理に効果的である。RDD の操作には主にアク

[1] 慶應義塾大学大学院 理工学研究科
Graduate School of Science and Technology, Keio University

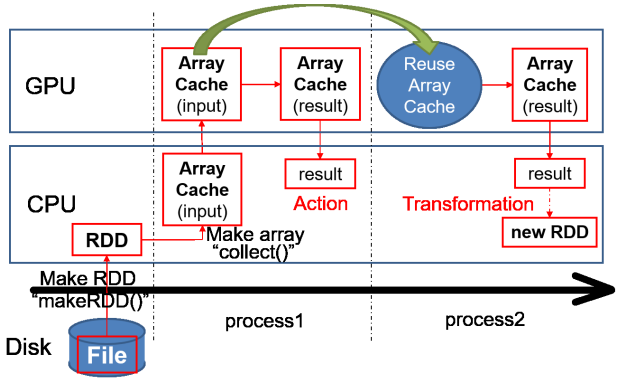


図 1　配列キャッシュ作成から GPU 処理の流れ

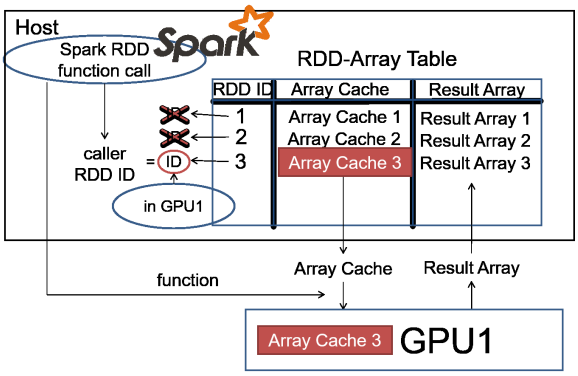


図 2　Spark からの GPU カーネル呼び出し

ション処理とトランスフォーメーション処理の 2 つがある。アクション処理は RDD に対し集約演算を行いスカラデータやベクタデータを結果として返す処理で、トランスフォーメーション処理は RDD を別の RDD に変換する処理である。

本論文では、Spark の `reduce` 処理などのアクション処理や `sortByKey` 処理などのトランスフォーメーション処理に加え、それらの複合処理を GPU を用いて高速化する。

2.2　関連研究

HeteroSpark[4] は Spark の各ノードに GPU を接続する研究である。cuSpark[5] は単一マシンに GPU を接続して並列処理を行う、Spark のようなフレームワークである。しかし、これらの研究は RDD の GPU 処理の過程や GPU デバイスに対する RDD のキャッシュ方法の詳細に言及していない。本論文では GPU 処理の際、RDD を配列キャッシュと呼ばれる GPU 処理に適したフォーマットに変換している。この配列キャッシュを必要に応じて GPU に転送して処理を行う。また、転送した配列キャッシュを GPU デバイスメモリに保持し続けることで、後続の処理の際データ転送オーバヘッドを削減する。

Spark 以外では、従来の MapReduce フレームワークに GPU を組み合わせるものがある [6][7][8]。Mars[6] は GPU ベースの MapReduce フレームワークの一つで、GPU のスレッド並列性を活用するために CPU ベース MapReduce に似た API を提供している。

本研究では、デバイスメモリに RDD の配列キャッシュをキャッシュすることに加え、ローカル GPU とリモート GPU の併用についても言及する。ローカル GPU とはホストマシンの PCIe に直接接続された GPU で、リモート GPU とはホストマシンの PCIe に 10GbE のネットワーク越しに接続された GPU である。ネットワーク越しの接続において、我々は NEC ExpEther[9][10] を使用する (図 3 に示す)。ExpEther を使うことで、GPU デバイスなどのハードウェアリソースへの PCIe パケットはイーサネットフレームにカプセル化され、10GbE で転送される。

3.　Spark 処理の GPU オフローディング

3.1　システム概要

図 1 に、Spark から GPU を用いてリダクションとトランスフォーメーションを行う流れを示す。まず、既存の Spark RDD メソッドの RDD を作成するメソッド (makeRDD) を用いてデータセットの RDD を作成する。その後、配列化メソッド (collect) を用いて RDD を GPU 処理に適した配列に変換する。この配列を以後、配列キャッシュと呼ぶ。作成された配列キャッシュは GPU に転送され Spark メソッドに対応する CUDA カーネルが実行される。ここで、RDD から配列に変換する際のオーバヘッドを削減するため、一度 RDD から作成された配列キャッシュはホストメモリに格納され 2 回目以降の処理で再利用される。加えて、GPU デバイスメモリに転送された配列キャッシュを別の配列キャッシュが転送されるまで保持させることで、2 回目以降の処理の際、転送オーバヘッドを削減する。GPU 側で配列キャッシュを処理した後、処理結果を CPU 側に返して処理が完了する。

3.2　RDD の配列キャッシュ

図 2 に Spark から GPU 利用メソッドが呼ばれた際の、ホストメモリに RDD をキャッシュし GPU カーネルを実行するまでの流れを示す。図 2 に示すように、一度使われた配列キャッシュは GPU デバイスメモリにキャッシュされ後続の処理で再利用される。ここで、ホスト側に各 RDD と各配列キャッシュの関係を保持するテーブルを導入する。このテーブルを RDD-Array テーブルと呼ぶ。このテーブルは Spark から参照可能で、RDD から配列キャッシュが作成される際、元の RDD の ID と作成された配列キャッシュはこのテーブルに同一エントリの要素として登録される。また、トランスフォーメーション処理実行時に、結果が結果配列として登録される。また、GPU デバイスメモリにキャッシュされている配列キャッシュをもつエントリの RDD の ID が記憶され、デバイスメモリ内の配列キャッシュの有無の判別を行う。GPU 利用メソッドが Spark から呼ばれると、処理対象 RDD がすでに配列キャッシュに変換されているか RDD-Array テーブルを調べる。もしテーブルになければ対象 RDD は配列キャッシュに変換されテーブルに登録される。テーブル内にある場合、対象 RDD の配列キャッシュが GPU デバイスメモリに転送済みかを調べ、デバイスメモリになければ転送、あれば転送せずに再利用する。

GPU 側では、Spark メソッドに対応する CUDA カーネルが実行される。Scala や Java 言語で構成された Spark から CUDA カーネルを呼び出すため、我々は jcuda[11] を用いる。

3.3　Spark 処理の GPU 実装

Spark のアクション処理の一つである `reduce` 処理を GPU にオフロードして行う Sum プログラムの詳細について説明する。

`reduce` 処理は入力 RDD の各要素の和などを計算する際に使用される。通常 Spark での RDD の要素和の計算は、以下のように行われる。

(1) RDD を作成する。
(2) 入力 RDD に `reduce` 処理を行い要素和を計算する。

Spark の `reduce` 処理に対応する CUDA カーネルの実装は文献 [12] をもとに行った。Spark から GPU を呼び出して処理を

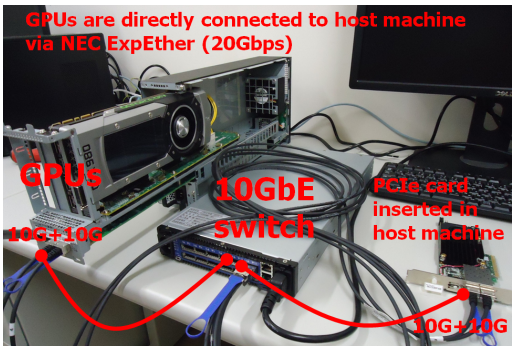


図 3　10GbE スイッチ越しにネットワーク接続されたリモート GPU

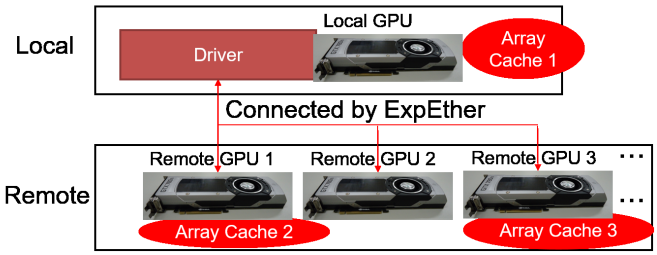


図 4　ローカルとリモート GPU を接続した Spark

行う際、処理過程は以下のように変化する。
(1) RDD を作成する。
(2) RDD を配列化し RDD-Array テーブルに登録する。
(3) 配列キャッシュを GPU デバイスメモリに転送した後、GPU 側で `reduce` メソッドに対応する CUDA カーネルを実行する。

我々は、Sum プログラム以外にも、RDD の行数を数える `count` 処理に対応する LineCount プログラムや RDD の要素の最大値を見つける `max` 処理に対応する Max プログラムを CUDA カーネルで実装した。これらの処理過程は上記の Sum プログラムと概ね同様である。

また、トランスフォーメーション処理において、キー値で要素をソートする SortByKey プログラムと小文字大文字の変換を行う WordConversion プログラムを CUDA カーネルで実装した。加えてアクション処理とトランスフォーメーション処理の複合処理として RDD の全要素の分散の値を求める Variance プログラムの実装も行った。これらの処理過程もアクション処理と概ね同様である。

4. ローカル GPU とリモート GPU

4.1 システム概要

この章では、Spark からローカル GPU とリモート GPU を利用することを提案する。図 4 にシステムの概要図を示す。ホストマシンに接続可能な GPU の数は PCIe スロット数により制限されるため、GPU デバイスメモリにキャッシュ可能な RDD の数やサイズも制限される。GPU デバイスメモリの空き容量が十分でないとき、RDD の配列キャッシュはデバイスメモリから頻繁にスワップされる。頻繁なデバイスメモリ内データのスワップは性能の低下につながりうる。そこで、我々は少数のローカル GPU に加え、多くのリモート GPU を PCIe を通して 10GbE ネットワーク越しに接続することでデバイスメモリ容量を拡大する。また、GPU 数を増やすことで多くのタスクを GPU で並列に処理することも可能になる。

RDD の配列キャッシュを GPU デバイスメモリにキャッシュするにあたり、キャッシュ先の GPU の選択が重要である。リモート GPU は数が多い一方で、10GbE の通信オーバヘッドが大きい。また、ローカル GPU は数は少ないが 10GbE の通信オーバヘッドはかからない。そのため、ローカル GPU とリモート GPU の効率的な使い分けは Spark の高速化に大きく影響する。

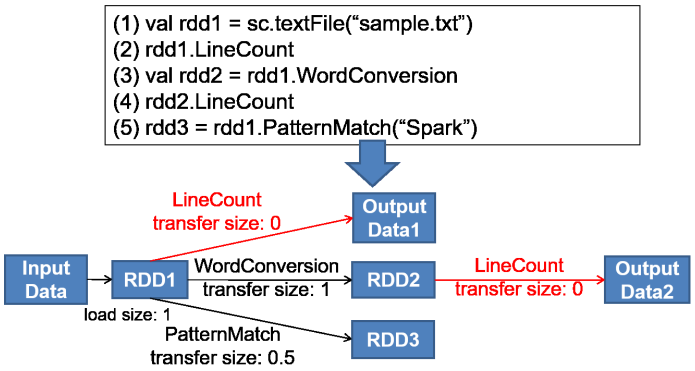


図 5　RDD グラフの作成

4.2 各 GPU に対する RDD キャッシュポリシー

ローカル GPU とリモート GPU の特性を考慮した RDD のキャッシュポリシーは性能に影響する。本論文では、我々は以下の 3 つの RDD キャッシュポリシーを提案する。
(1) Policy-1:最も利用頻度の高い RDD の配列キャッシュをローカル GPU にキャッシュする。
(2) Policy-2:ホストとリモート GPU 間のデータ転送回数を最小化する。
(3) Policy-3:ホストとリモート GPU 間のデータ転送総量を最小化する。

Policy-1 はリモート GPU での CUDA カーネル実行オーバヘッド削減を目的としている。Policy-2 と Policy-3 はホストとリモート GPU 間のデータ転送の削減を目的としており、特に Policy-2 では転送回数、Policy-3 では転送量に焦点を当てている。データ転送はホストからリモート GPU への入力データ転送と、リモート GPU からホストへの出力データ転送の二つがある。出力データサイズは実行処理の種類に依存する。アクション処理ではスカラデータかベクタデータを出力するのに対し、トランスフォーメーション処理では RDD を出力するため出力データサイズはアクション処理のものより大きい。同一入力データに対し CUDA カーネルが実行されるとき、2 回目以降の入力データの転送は行われない。

3 つの RDD キャッシュポリシーは対象アプリケーションのソースコードの静的解析の後、適用される。我々のコード解析プログラムを使うことでアプリケーションコードから、ノードが RDD でエッジが RDD 処理の RDD グラフが抽出される。RDD グラフのノードとエッジの数を数えることで、対象アプリケーションにおける各 RDD のアクセス回数と各 RDD の転送サイズを見積もることができる。

RDD の各ノードは処理結果サイズの情報を保持する。我々の静的解析プログラムは各エッジでの処理結果サイズをスカラサイズ、元 RDD より小さいサイズ、元 RDD と同程度のサイズ、元 RDD より大きいサイズのいずれかに分類する。例えば、アクション処理の結果はスカラサイズとして分類される一方、トランスフォーメーション処理の `sortByKey` 処理の結果は要素がソートされた同一サイズの RDD であるため元 RDD と同程度のサイズとして分類される。また、`filter` 処理後は RDD の要素数が減少するため、元 RDD より小さいサイズと分類される。

図 5 に静的解析プログラムによる RDD グラフ抽出の具体例を示す。RDD1 のアクセス回数は 3 回で RDD2 のアクセス回数は

表 1　評価環境

CPU	Intel Xeon E5-2637 v3
Number of CPU cores	4
Operating frequency	3.50GHz
Memory capacity	256GB
GPU	GeForce GTX 980 Ti
OS	CentOS 6.7
CUDA version	7.5
jcuda version	0.7.5
Spark version	1.6.0 (Stand alone mode)
Scala version	2.10

1 回である。入力データセットが GPU デバイスメモリに転送されるとき、入力データサイズは “1” と記録される。アクション処理が行われるとき、処理結果はスカラ値のため “0” が結果サイズとして記録される。トランスフォーメーション処理が行われるとき、処理結果サイズが元 RDD より小さければ “0.5” が、元 RDD と同程度ならば “1” が、元 RDD より大きければ “1.5” が記録される。ここでは、LineCount プログラムはアクション処理のため結果サイズに “0” が記録される。WordConversion プログラムは同一サイズの RDD を生成する処理のため、結果サイズに “1” が記録される。PatternMatch プログラムは対象文字列を抽出した RDD を生成する処理のため、結果サイズには “0.5” が記録される。入力時のデータサイズと出力データサイズの合計から、RDD1 の合計転送サイズは “2.5” と記録される。

RDD グラフにおける各 RDD のアクセス回数や入出力サイズを基に、RDD キャッシュポリシーが適用され、各 RDD がローカル GPU とリモート GPU に振り分けられる。例えば、Policy-1 では利用回数最大の RDD1 がローカル GPU にキャッシュされる。これらの RDD キャッシュポリシーごとの実行時間比較ついては 5 章にて述べる。

5. 評価

5.1 アクション処理の性能評価

この章では、アクション処理の実行時間について、CPU のみの Spark とローカル GPU を用いた Spark、そしてリモート GPU を用いた Spark の比較を行う。表 1 に評価環境を示す。Java ヒープ領域は 200GB とした。リモート GPU において、各 GTX 980 Ti GPU はホストマシンに NEC ExpEther10G で、2 つの SFP+モジュールを介して接続される。

アクション処理の性能評価には、Sum プログラム、Max プログラム、LineCount プログラムを用いた。評価にあたり、各プログラムを 5 回連続で実行するのを 1 セットとし、それを 50 セット実行し平均値をとった。5 回連続の実行の最初の実行には Spark 起動と RDD 作成のオーバヘッドがかかる。それに加え、GPU 実行では配列キャッシュの作成時間もかかる。現在、配列キャッシュ作成は最適化されておらず、他のオーバヘッドに比べ大きな時間がかかる。この問題点に関しては、6 章で言及する。図 6 に各プログラムの実行時間比較を示すが、この図には Spark の起動時間、RDD 作成時間、配列キャッシュ作成時間を含めていない。これらのオーバヘッドは、RDD が新たに作成された時にしか発生しないためグラフからは省略する。これら 3 つのオーバヘッドの内、配列キャッシュ作成時間のみ “overhead” としてグラフのキャプションに示す。

図 6(a) に CPU、ローカル GPU、リモート GPU による Sum プログラムの実行時間を示す。要素の値がランダムの 1GB の Int 型配列を入力データとした。また、Spark の起動に 1.9 秒、RDD 作成に 8.4 秒、配列キャッシュ作成に 15.6 秒かかった。ここで、GPU での実行時間は GPU 処理時間と、ホストと GPU デバイス間のデータ転送時間の合計である。1 回目の実行の際はデータ転送オーバヘッドがかかっているが、2 回目以降ではデバイスメモリ内データを再利用するため転送オーバヘッドがかからず、全体の実行時間は短くなった。Sum プログラムにおいて、ローカル GPU 実行は CPU 実行に対し 1 回目は 11.4 倍の性能向上、2 回目以降は 14.4-15.7 倍の性能向上を達成した。また、リモート GPU 実行は CPU 実行に対し 1 回目は 4.7 倍、2 回目以降は 14.5-15.7 倍の性能向上を達成した。

図 6(b) に Max プログラムの実行時間比較を示す。要素の値がランダムの 1GB の Int 型配列を入力データとした。図 6(c) に LineCount プログラムの実行時間を示す。ランダムに生成された 1GB のテキストファイルを入力データとした。

5.2 トランスフォーメーション処理の性能評価

トランスフォーメーション処理の性能評価には、SortByKey プログラムと WordConversion プログラムを用いた。

図 6(d) に、CPU とローカル GPU とリモート GPU によるキーソートの実行時間比較を示す。1GB のキーバリュー配列を入力データとした。ただし、キーは 512MB の Int 型データ、バリューは 512MB の Int 型データである。また、GPU 処理の際、キーのみを転送したため転送データサイズは 512MB である。既存 Spark メソッドを用いて RDD からキーを抜き出し配列化しているため、配列キャッシュ作成時間は大きく、改善の余地がある。

2 回目以降の GPU 実行では、転送済みデータを再利用するため、転送オーバヘッドがかからない。しかし、図 6(d) でローカル GPU とリモート GPU における 1 回目と 2 回目以降の実行時間にはほとんど差が見られない。これは、SortByKey プログラムの GPU 処理の時間が大きく、転送時間が相対的に小さくなるためである。結果的に、ローカル GPU 実行は CPU 実行に対し 1 回目は 16.0 倍の性能向上、2 回目以降は 15.7-20.0 倍の性能向上を達成した。また、リモート GPU 実行は CPU 実行に対し 1 回目は 14.2 倍、2 回目以降は 14.6-18.9 倍の性能向上を達成した。

図 6(e) に、WordConversion プログラムの実行時間比較を示す。ランダムに生成された 1GB のテキストファイルを入力データとした。転送時間の割合が多いため、リモート GPU の CPU に対する性能向上率は控えめなものとなった。

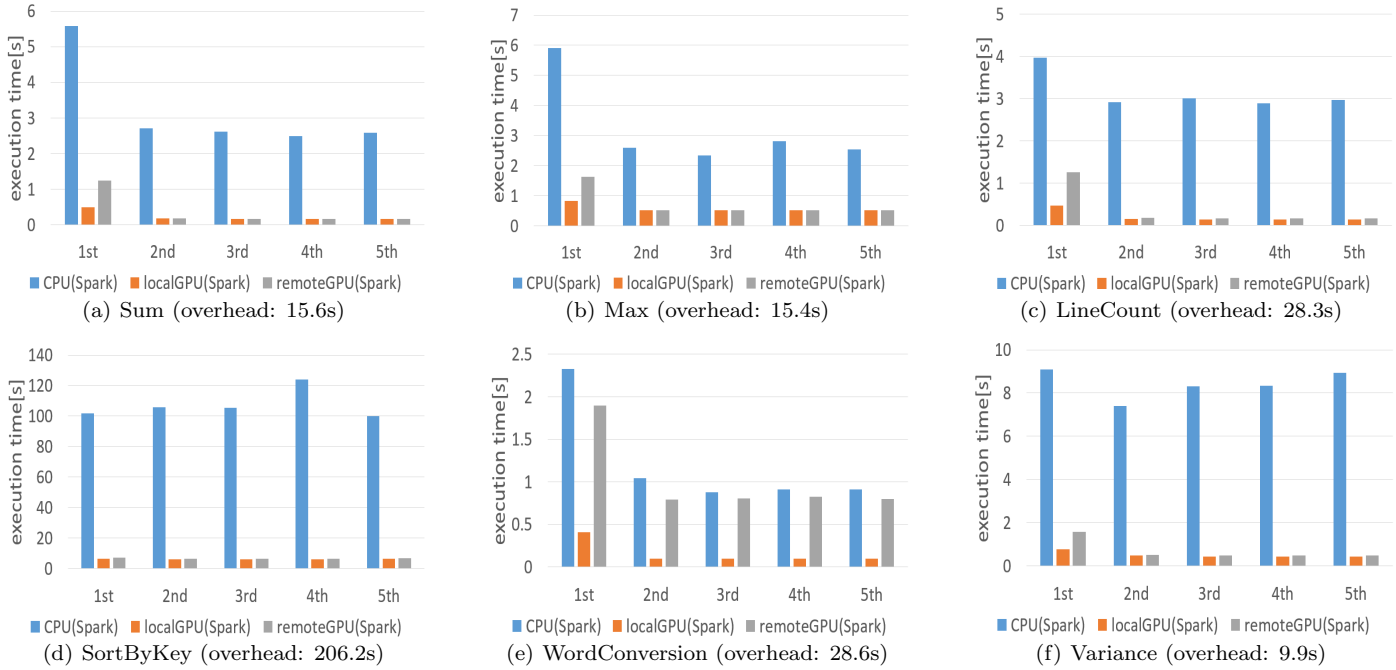
5.3 複合処理の性能評価

アクション処理とトランスフォーメーション処理の複合処理の性能評価には、Variance プログラムを用いた。

図 6(f) に CPU とローカル GPU とリモート GPU による Variance プログラムの実行時間比較を示す。要素がランダムの 1GB の Int 型配列を用いた。ローカル GPU 実行は CPU 実行に対し 1 回目は 11.5 倍の性能向上、2 回目以降は 15.2-19.9 倍の性能向上を達成した。また、リモート GPU 実行は CPU 実行に対し 1 回目は 5.8 倍、2 回目以降は 14.3-18.5 倍の性能向上を達成した。

5.4 RDD キャッシュポリシーごとの性能評価

前節までに示したように、様々なプログラムの実行時間を CPU、ローカル GPU、リモート GPU にて測定した。その測定結果をもとに 4.2 節で提案した RDD キャッシュポリシーによる性能比較のシミュレーションを行った。シミュレーションのため、我々が実装した各プログラムを実行する 3 つのアプリケーションを想定した。1 つ目と 2 つ目のアプリケーションは単純な処理のため、ローカル GPU1 つ、リモート GPU1 つと想



(a) Sum (overhead: 15.6s) (b) Max (overhead: 15.4s) (c) LineCount (overhead: 28.3s)

(d) SortByKey (overhead: 206.2s) (e) WordConversion (overhead: 28.6s) (f) Variance (overhead: 9.9s)

図 6 CPU、ローカル GPU、リモート GPU による各プログラムの実行時間

定した。3 つ目のアプリケーションは処理量が大きいため、ローカル GPU1 つ、リモート GPU7 つと想定した。また、各 GPU デバイスメモリに一度にキャッシュ可能な配列キャッシュは 1 つとした。RDDj が RDDi に依存するとき、これらの RDD は逐次的に処理され、依存性がなければ並列に処理される。

1 つ目のアプリケーションは、1GB の Int 型データを RDD1、1GB テキストデータを RDD2、別の 1GB テキストデータを RDD3 としてロードする。それらは以下の手順で処理が行われる。

(1) RDD1.Sum
(2) RDD2.LineCount
(3) RDD1.Variance
(4) RDD4 = RDD3.WordConversion
(5) RDD1.Max
(6) RDD5 = RDD2.WordConversion
(7) RDD1.Min

このアプリケーションでは、RDD1 が利用回数最大の RDD と判別される。

2 つ目のアプリケーションは、1GB キーバリューデータを RDD1 としてロードする。その後以下の手順で処理が行われる。

(1) RDD2 = RDD1.SortByKey
(2) RDD3 = RDD2.values.WordConversion
(3) RDD2.keys.Max
(4) RDD2.keys.Min
(5) RDD3.LineCount
(6) RDD3.Min

このアプリケーションでは、RDD グラフに沿って逐次的に RDD が作成される。最も頻繁に利用される RDD は RDD2 である。

RDD キャッシュポリシーを処理量の大きいアプリケーションで評価するため、ランダムな 50 回のメソッドをランダムに選ばれた RDD に実行するアプリケーションを、3 つ目のアプリケーションとして用いた。初期の RDD は 3 つとし、各 RDD のサイズは 1GB とした。規模が大きく、ランダムに生成したためアプリケーションの詳細は省略する。表 2 と 3、そして表 4 に 3 つのアプリケーションのシミュレーション結果を示す。

1 つ目のアプリケーションでは、RDD 間の依存関係がない状況を想定している。そのため、リモート GPU を使って処理が並列に実行され、性能が向上につながる。Policy-1 を適用すると最も使われる RDD1 をスワップなしでローカル GPU により処理できる。しかし、リモート GPU では RDD2 と RDD3 が交互に処理されるため、ホストとリモート GPU 間のデータ転送量が大きい。結果的にリモート GPU 実行はローカル GPU 実行の 2 倍以上の実行時間がかかり、全体性能が悪化した。Policy-2 を適用した場合、ホストとリモート GPU の間のデータ転送回数は 2 回のみになる。結果的に、Policy-1 よりもデータ転送オーバヘッドが少なくなり性能が向上した。Policy-3 を適用すると、転送量の多い RDD2 と RDD3 はローカルで処理される。結果的に転送オーバヘッドが大きく削減され、Policy-3 は他のポリシーよりも性能がでた。

2 つ目のアプリケーションでは、RDD 間に依存関係がある状況を想定している。依存関係があると待ち時間が発生し、リモート GPU を多く使う利点が小さくなる。結果的に、ローカル GPU のみを使った時の性能が相対的によくなった。しか

表 2 アプリケーション 1 のシミュレーション結果

RDD caching policy	RDD1	RDD2	RDD3	Execution time
Policy-1	Local	Remote	Remote	4.987 (sec)
Policy-2	Local	Local	Remote	3.504 (sec)
Policy-3	Remote	Local	Local	2.777 (sec)
Local GPU only	Local	Local	Local	4.223 (sec)

表 3 アプリケーション 2 のシミュレーション結果

RDD caching policy	RDD1	RDD2	RDD3	Execution time
Policy-1	Remote	Local	Remote	8.258 (sec)
Policy-2	Remote	Local	Local	8.387 (sec)
Policy-3	Local	Local	Remote	7.450 (sec)
Local GPU only	Local	Local	Local	7.579 (sec)

表 4 アプリケーション 3 のシミュレーション結果

RDD caching policy	Execution time
Policy-1	20.278 (sec)
Policy-2	19.811 (sec)
Policy-3	18.576 (sec)
Local GPU only	46.479 (sec)

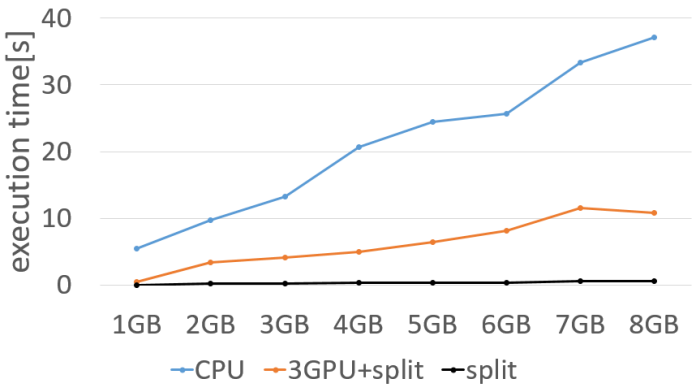


図 7　Sum プログラムの RDD サイズごとの実行時間

し、Policy-3 では RDD3 生成後に RDD2 と RDD3 をローカル GPU とリモート GPU で並列して処理するため、最高の性能となった。

3 つ目のアプリケーションのような規模の大きいアプリケーションの場合はリモート GPU を使った配列キャッシュと並列処理が性能向上につながることが分かる。ここでも Policy-3 が最大性能である。以上の結果から、並列実行可能なアプリケーションに対してはより多くのリモート GPU を用いることで、デバイスメモリからの RDD スワップ回数とデータ転送オーバヘッドが減り性能が向上する。

5.5　大容量 RDD 処理の性能評価

今回の実装では、GPU デバイスメモリより大きいサイズの RDD の配列キャッシュは複数の配列に分割され複数の GPU に並列処理される。簡単のため GPU デバイスメモリサイズを 1GB と仮定し、RDD サイズを 1GB から 8GB に変えた時の Sum プログラムの実行時間を評価した。我々は 1 つのローカル GPU と 2 つのリモート GPU をこの評価に用いた。つまり、1GB の分割された配列が 3 つまで並列に処理される。この並列処理はすべての分割された配列が処理されると終了する。

図 7 に CPU と 3GPU+split、そして split による 1 回目の処理の実行時間を示す。ここで、“3GPU+split” は 3 つの GPU に RDD を分割して処理する時の実行時間を示し、“split” は RDD を 1GB の配列に分割するのにかかる時間を示している。これらは 10 回の実行の平均値である。図のように、3GPU+split の実行時間はデータサイズが増えるにつれ増加した。split の時間は計算時間に比べてごくわずかだった。

6.　まとめと今後の課題

本論文では、Spark の計算インテンシブな処理を高速化するために、Spark から CUDA カーネルを呼び出し処理をオフロードするようにした。また、GPU デバイスメモリに RDD を可能な限りキャッシュすることで転送量を減らした。ローカル GPU に加え、リモート GPU も利用することでデバイスメモリ容量を拡大し、多くのタスクの並列処理も可能にした。ローカル GPU とリモート GPU に対し、3 つの RDD キャッシュポリシーの提案、評価も行った。我々は今回、Sum、Max、LineCount、SortByKey、WordConversion、そして Variance プログラムをローカル GPU とリモート GPU を活用する Spark 上に実装した。各プログラムのローカル GPU とリモート GPU での実行時間を、同じ操作が繰り返し行われる前提で評価した。評価結果から、我々の修正した Spark がローカル GPU を利用したとき、通常の Spark と比較して最大 21.4 倍の性能向上を達成した。リモート GPU を利用した場合の性能は、ローカル GPU の実行時間と比べ 1 回目の実行以外ではほとんど劣化することはなかった。リモート GPU へのデータ転送オーバヘッドのため、3 つの RDD キャッシュポリシーではリモート GPU へのデータ転送量を最小にしたものが最も高性能となった。今回我々は、単一ホストマシンにローカル GPU とリモート GPU を組み合わせて使ったが、複数ホストマシンに拡張し各マシンがローカル GPU とリモート GPU を活用できるようにすることも可能である。

本論文は Spark からローカル GPU とリモート GPU を効率的に使うことに焦点を当てているが、配列キャッシュ作成時間は削減すべきである。今回は既存 Spark メソッドの `collect` 処理を使って RDD から配列キャッシュを作成した。今後は変換オーバヘッドを削減するために、RDD から配列キャッシュを作成する独自の変換処理を実装することを計画している。今回のアプリケーションは簡単なものを実装したが、今後は洗練されたアプリケーションを実装することも予定している。

参考文献

[1] Apache Hadoop: `http://hadoop.apache.org/`.
[2] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I.: Spark: Cluster Computing with Working Sets, *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*, pp. 10–10 (2010).
[3] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. and Stoica, I.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, pp. 15–28 (2012).
[4] Li, P., Luo, Y., Zhang, N. and Cao, Y.: HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Machine Learning Algorithms, *Proceedings of the International Conference on Networking, Architecture and Storage (NAS'15)*, pp. 347–348 (2015).
[5] cuSpark - a Functional Data Processing Framework: `http://www.yaomuyang.com/cuspark/`.
[6] He, B., Fang, W., Luo, Q., Govindaraju, N. K. and Wang, T.: Mars: A MapReduce Framework on Graphics Processors, *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*, pp. 260–269 (2008).
[7] Ji, F. and Ma, X.: Using Shared Memory to Accelerate MapReduce on Graphics Processing Units, *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS'11)*, pp. 805–816 (2011).
[8] Elteir, M., Lin, H., chun Feng, W. and Scogland, T.: StreamMR: An Optimized MapReduce Framework for AMD GPUs, *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS'11)*, pp. 364–371 (2011).
[9] Suzuki, J., Hidaka, Y., Higuchi, J., Yoshikawa, T. and Iwata, A.: ExpressEther - Ethernet-Based Virtualization Technology for Reconfigurable Hardware Platform, *Proceedings of the IEEE Annual Symposium on High-Performance Interconnects (HOTI'06)*, pp. 45–51 (2006).
[10] Suzuki, J., Hayashi, Y., Kan, M., Miyakawa, S. and Yoshikawa, T.: End-to-End Adaptive Packet Aggregation for High-Throughput I/O Bus Network Using Ethernet, *Proceedings of the IEEE Annual Symposium on High-Performance Interconnects (HOTI'14)*, pp. 17–24 (2014).
[11] jcuda.org: `http://www.jcuda.org/`.
[12] Mark Harris: Optimizing Parallel Reduction in CUDA, `http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf`.