Vol.2016-DBS-163 No.2
Vol.2016-IFAT-123 No.2
2016/9/13

GPU を活用する DBMS における リソース競合を考慮したクエリスケジューリング

高橋 祥平[1,a)]　山田 浩史[1,b)]

概要：クラウドコンピューティング環境において，1 つのデータベース管理システム（DBMS）で複数ユーザのデータベースを管理するマルチテナント方式が多くのサービスで採用されている．1 つの DBMS にデータベースを集約することで，サーバ台数の減少による消費電力の削減などのメリットが得られるため，DBMS 単体の性能向上が必須となる．そこで，1 台のマシン上で DBMS の性能を上げる方法として，DBMS へのリクエストを GPU を用いて高速に処理する技術が研究されている．GPU へ処理をオフロードさせる技術は処理の高速化や CPU の負荷の軽減をもたらすが，GPU に処理が集中することで GPU のリソースが不足し，性能が劣化する場合がある．本論文では GPU を活用する DBMS におけるリソース競合を考慮したクエリスケジューリングを提案する．状況に応じて GPU での実行を CPU に切り替え，GPU への処理の集中を抑制することで性能の劣化を防ぐ．具体的には，CPU での処理時間が長い処理を優先的に GPU 上で処理させることで全体のターンアラウンドタイムの短縮を図る．提案手法を PG-Strom，PostgreSQL 9.5 に対して実装した．評価実験では性能劣化を抑制しデフォルト環境と変わらない時間で処理を完了できた．また，拡張前の PG-Strom との比較においては実行時間の短縮が確認できた．

1. Introduction

クラウドコンピューティング環境において，1 つのデータベース管理システム（DBMS）で複数ユーザのデータベース（DB）を管理するマルチテナント方式が多くのサービスで採用されている．以前はマシン 1 つあたりのスペックが低く，クラウドサービスを提供する上で多数のサーバが必要であったが，性能が向上しマシンのリソースに余裕が出てくると複数のサーバを 1 つのサーバに統合する動きが見られるようになった．実際に Oracle が提供する Oracle Database 12c[1] ではマルチテナント方式を用いたサーバ統合を行っている．

1 台のマシン上で DBMS の性能を上げる方法として，DBMS へのリクエストを Graphics Processing Unit(GPU) を用いて高速に処理する技術 [4], [5], [6], [7] が研究されている．GPU を画像処理以外に応用する技術である General Perpose GPU (GPGPU)[2], [3] を用いて DBMS へ送られてきたリクエストを GPU へオフロードすることによって，GPU が持つ並列処理性能によりリクエストの処理時間を短縮させるだけでなく CPU への負荷の軽減が可能になる．

GPU へ処理をオフロードさせる技術は処理の高速化や CPU の負荷の軽減をもたらすが，GPU に処理が集中することで性能が劣化する場合がある．GPU は CPU が得意とする条件分岐を伴う複雑な処理を苦手とするため GPU を活用できる処理は限られる．そのため，リクエストの内容によって，CPU もしくは GPU のどちらで処理するかが固定されており，GPU リソースを超えるリクエストを処理する際に大きな性能劣化が発生してしまう．DBMS の 1 つである PostgreSQL[8] の拡張モジュールの PG-Strom[9] を用いて実験をしたところ，4 つの集約処理を GPU で並列処理すると標準の PostgreSQL と比べて実行時間が約 3 倍に増加している．

本論文では，GPU 上のリソース競合を考慮したクエリスケジューリングを提案する．提案手法では GPU で処理するクエリに対してスケジューリングを行い，GPU でのクエリの処理の集中を抑制することで全体の性能劣化を防ぐ．スケジューリングの際には各クエリの CPU での処理時間を見積もり，CPU 上での処理時間が最も長いクエリを優先的に GPU で処理させることによって全体の実行時間の短縮を図る．

今回は PostgreSQL の拡張モジュールである PG-Strom を拡張する形で実装し，Linux 3.13.0，PostgreSQL 9.5 を使用して実験を行った．劣化が発生する条件下では性能劣

[1] 東京農工大学
TUAT, Koganei, Tokyo 101-0062, Japan
[a)] shoheit@asg.cs.tuat.ac.jp
[b)] hiroshiy@cc.tuat.ac.jp

化を抑制しデフォルト環境と変わらない時間で処理を完了できた．また，拡張前の PG-Strom との比較においても実行時間の短縮が確認できた．

本論文では，第 2 章で本研究の動機となった技術である GPGPU と GPU を活用した DBMS について述べ，既存の手法が抱える問題について触れる．第 3 章ではその解決策となる手法を提案する．第 4 章では提案手法を実現するための設計を述べ，第 5 章では実装について述べる．第 6 章で提案手法の評価を行い，第 7 章で関連研究，第 8 章で現状と今後の展望について述べる．

2. Motivation

2.1 GPGPU

GPGPU(General-Purpose computing on GPU) とは GPU を用いた汎目的演算のことであり，画像処理以外の様々な演算処理に GPU を活用する技術である．GPGPU を活用することで，CPU 上で実行される処理の一部を GPU の並列処理性能を用いて高速に処理することが可能になる．

GPGPU ではまず，GPU 上で実行する処理が記述されたカーネルを GPU 上にロードする．続いて，CPU は GPU 上のメモリ領域を確保し，演算に必要なデータを作成して GPU へコピーする．データの転送が完了したら，CPU から GPU 上のカーネルを起動することによって GPU 上で処理が開始される．GPU での処理が完了したら，演算結果を GPU から CPU へコピーし，演算結果を取得する．演算結果の取得後は使用したメモリ領域の解放を行う．

2.2 GPGPU を活用した DBMS

GPGPU を活用した DBMS は，一連の処理を全て GPU へオフロードする DBMS と，CPU と連携し一連の処理の一部のみを GPU へオフロードする DBMS の 2 種類が存在する．本論文では一連の処理を全て GPU へオフロードする DBMS を対象にしている．

2.2.1 GPU に処理を全てオフロードする DBMS

DBMS ではテーブルに対して Scan や Join などの処理を実行するが，こうした処理を全て GPU 上で実行することで処理を高速化する手法 [10], [11], [12] が存在する．PG-Strom [9] は PostgreSQL における Scan, Hash-Join, Aggregation を GPU にオフロードすることで処理を高速化する．PG-Strom では対象となるテーブルを指定したサイズでチャンクと呼ばれるまとまりに区切り，チャンクごとに GPU 上で高速並列処理することで実行時間の短縮を実現している．GPU から受け取った実行結果は CPU により処理される．Ocelot [13] は列指向 DB である MonetDB[14] をベースとしている．既存手法は GPU を使用する際にハードウェアに合わせて hand-tune を施す必要があるが，Ocelot では OpenCL を用いてハードウェアを抽象化することで hand-tune を施さずに様々なデバイスへの対応を可能としている．クエリの実行においては Selection や Join, Aggregation に加えて GROUP BY を用いた際の処理の高速化を提供する．こちらも最終的な処理は CPU が実行する．

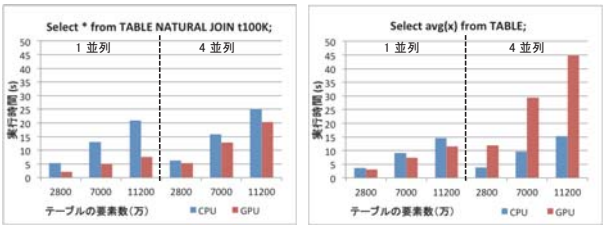

図 1 CPU, GPU 上での並列処理による影響

2.2.2 CPU と GPU が連携して処理を行う DBMS

DBMS において一部の処理がスループット向上のボトルネックとなる場合に，その処理のみを GPU が担当し，CPU と GPU が連携して処理を行うことでスループット向上を図る手法 [16], [17], [18], [19] が存在する．Mega-KV[15] では，Key-Value Store における Value を CPU, Key を GPU で管理し，GPU の並列処理性能によって index operation を高速処理する．Index operation のみを GPU で実行し，その他の処理は CPU で実行される．

2.3 Performance Issue

PG-Strom や Ocelot など，DBMS において GPU を活用した処理の高速化の手法は既に存在するが，これらの手法では GPU における複数のクエリやワークロードの実行が考慮されていない．クエリの内容によって CPU, GPU のどちらで処理するか固定されているため, GPU 上で処理するクエリが複数送られてきた場合は GPU 上で複数のクエリを同時に処理することになる．GPU 上での複数のクエリの並列処理は GPU のリソースの競合をもたらし, 性能が劣化してしまう恐れがある．

GPU 上での複数クエリの並列処理による性能劣化を確認するために Join, Aggregation の SQL を 1, 4 並列で実行した．図 1 は PostgreSQL を用いた CPU のみでの実行時間, PG-Strom を用いた GPU のみでの実行時間を示している．実行した SQL はグラフの上部に示している．Join の並列処理では GPU 上での並列処理は CPU での並列処理よりも短時間での実行が可能であるため，Join については常に GPU で処理しても問題ない．しかし，Aggregation の並列処理においては GPU 上で 1 つの Aggregation を処理する場合は CPU の約 1.5 倍の速度で処理できるが，4 並列における GPU での実行時間は CPU の約 3 倍以上に増加してしまう．Ocelot においては TPC-H No.18 のクエリを 3 並列以上で処理すると，単体では CPU より高速だった GPU での実行時間が CPU を上回る．

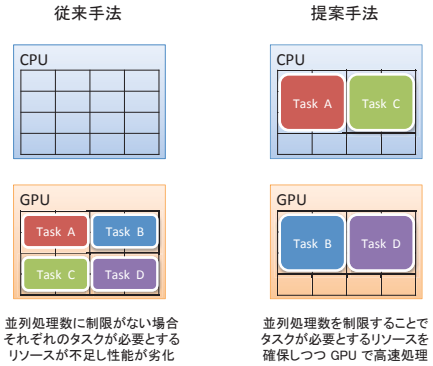

図 2 従来手法と提案手法

3. Approach

3.1 GPU リソース競合を考慮したスケジューリング

複数のクエリを GPU 上で並列処理することで発生する性能劣化を防ぐ手法として，GPU 上のリソース競合を考慮したクエリスケジューリングを提案する．既存の手法には GPU 上で実行されるクエリの実行順序を制御するスケジューリング [11] は存在したが，GPU 上での実行が決定したクエリを CPU での実行に切り替えるスケジューリングはなかった．GPU のクエリスケジューリングによって GPU のリソースの不足を防ぐことができるが，GPU での処理が集中した場合クエリの実行は GPU が利用可能になるまで待機しなければならない．待機時間が長くなると GPU での高速処理による恩恵が得られなくなる．GPU へのクエリの処理のオフロードでは GPU 上での処理の実行中は CPU のリソースに余剰が発生するため，GPU へのクエリの処理が集中する場合はクエリの一部を CPU で処理することでリソースを有効活用する．提案手法では GPU 上のクエリに対してスケジューリングを実行し，GPU による恩恵が小さいと判断した場合一度 GPU での実行が決定していたクエリを CPU での実行に切り替える．

従来手法と提案手法の違いを図 2 に示す．提案手法では GPU のリソース不足を防ぐために，GPU 上でのクエリの同時処理数を制限する．DBMS は本来 GPU の使用を想定していないため，複数のユーザから同時にクエリが送られてきた場合，GPU の使用状況に問わず GPU へのオフロードを開始することで GPU 上のリソースが不足し，性能劣化が発生する．各ユーザのクエリを監視し，後述するスケジューリングポリシーに従って GPU 上で実行するクエリを決定し，その他のクエリを CPU で処理することで GPU でのクエリの処理の集中を抑制し，性能劣化を防ぐ．

3.2 スケジューリングポリシー

提案するスケジューリングでは，全体のターンアラウンドタイムを最小にすることをスケジューリングポリシーとする．クエリごとに CPU での実行時間を見積もり，GPU 上で処理するクエリの内 CPU での実行時間が最も長いクエリを GPU で処理し，それ以外のクエリを CPU で処理する．GPU 上で処理するクエリを CPU での実行時間が最も長いクエリに絞ることによって 1 つのクエリの処理にリソースを注ぎ，より短時間で GPU 上の処理を完了させることでスループットを向上させる．また，GPU 上の処理時間を短縮することによって特定のクエリによる GPU リソースの占有時間を減らし，多くのクエリが GPU へ処理をオフロードできる機会を得られるようにする．

4. Design

4.1 スケジューリングの開始

提案するスケジューリングでは，DBMS が GPU へ処理をオフロードするタイミングでスケジューリングを開始する．GPU 上のクエリに対してスケジューリングを実行し，今回は CPU 上で処理するクエリに対してはスケジューリングを行わない．スケジューリング開始後多少の猶予時間を設け，その間は GPU へ処理をオフロードするクエリを受け付ける．猶予時間内で受け付けたクエリに対してスケジューリングを行い，GPU 上で処理するクエリを決定する．猶予時間が経過したら，スケジューリングにより決定したクエリを GPU 上で実行する．GPU での処理が完了したらスケジューリングを停止し，新たに GPU へオフロードするリクエストが到達するまで待機する．

4.2 GPU 上で実行するクエリの決定

全体のターンアラウンドタイムを最小にするために，GPU 上で実行するクエリは予め予測した CPU での処理時間を基に決定する．CPU での処理時間は，リクエストの処理内容や対象とするテーブルの行数を基に予測する．予測に必要なパラメータは事前に集計・入力しておく．

クエリの決定はスケジューリング開始のきっかけとなったクエリを基準に実行する．キューへのクエリの登録によりスケジューリングが開始される．猶予時間内に新たにスケジューリングの対象となったクエリは，自身の CPU での処理時間とキュー内のクエリの CPU での処理時間を比較する．比較の結果，CPU での処理時間が長いクエリをキューに登録し，処理時間が短いクエリは CPU での処理に切り替え，即座に処理を開始する．猶予時間経過後，キュー内に存在するクエリを GPU で実行する．

スケジューリングアルゴリズムの概略図を図 3 に示す．SQL A が先にスケジューリングされ，その後 SQL B がスケジューリングされた場合を考える．SQL A の情報から CPU での実行時間を見積もり，Task A としてキューへの登録することでスケジューリングが開始される．スケジューリング開始後，SQL B がスケジューリングの対象となると，SQL B の CPU での実行時間を見積もる．スケジューリングはすでに実行中であるため，Task B とし

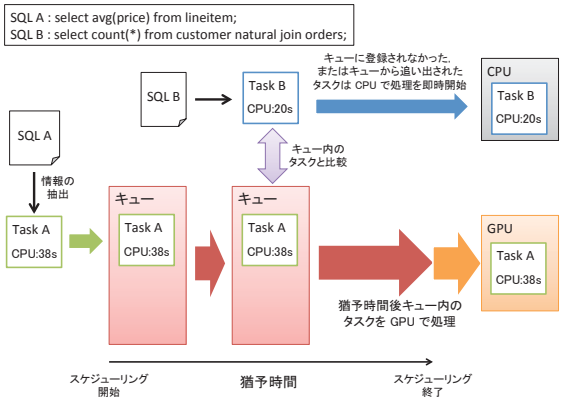

図 3 スケジューリングアルゴリズムの概略図

てキューへの登録を試みる．キュー内のタスクと実行時間を比較した結果，Task B は Task A と比べて GPU で処理する必要性がないと判断し，CPU での実行に切り替えられ，SQL B は CPU 上で処理される．猶予時間経過後，キュー内の Task A は GPU での実行が決定し，SQL A は GPU 上で処理される．

4.3 複数のクエリの並列処理

CPU での処理時間が最も長いクエリを GPU で処理するスケジューリングを行うが，状況に応じて GPU での複数のクエリの並列処理を考慮する．

クエリ間の CPU での実行時間の差が大きい場合，CPU での実行時間が最も長いクエリのみを GPU で処理する．CPU での実行時間が最も長いクエリにリソースを集中させることで，GPU 上で短時間で処理を行い全体のターンアラウンドタイムを大きく短縮することができる．

一方でクエリ間で CPU での実行時間が小さい場合は並列処理を考慮する．CPU での実行時間が最も長いクエリに併せて，その実行時間との差が小さいクエリについても GPU で処理する．GPU リソースの分配によって GPU 上での実行時間は増加するものの，CPU での実行時間が長いクエリを高速に処理し全体のターンアラウンドタイムを大きく短縮することが可能になる．

GPU 上での複数のクエリの並列処理により更なるターンアラウンドタイムの短縮が可能になるが，一方で GPU リソースの不足による性能劣化を招く恐れがある．従って，GPU 上での最大並列処理数はクエリの内容や使用するメモリ量などに応じて制限する．

5. Implementation

本論文では提案手法を PG-Strom 上に実装した．カーネルは linux-3.13.0，DBMS は PostgreSQL 9.5alpha2 を用いる．クエリの処理方法が GPU へオフロードを含むものであるかどうかを調べるために PostgreSQL 内にフック関数を導入し，GPU へのオフロードを含むクエリに対してのみスケジューリングを実行する．

PG-Strom による GPU へのオフロードは，リクエストの内容が Scan, Join のみの場合は最大で 4 つまでの並列処理に制限し，Aggregation のみの場合は単一での実行に制限する．Scan, Join と Aggregation を同時に GPU 上で並列処理する場合は最大で 2 つまでの並列処理に制限する．この並列処理の最大数は事前に GPU 上での並列処理数を変えて実行時間を計測した結果を基に設定している．

表 1 比較する実験環境

略称	設定	SQL の実行
Default	標準の PostgreSQL による実行	CPU
PG-Strom	標準の PG-Strom による実行	GPU
Custom	提案手法による実行	CPU & GPU

表 2 マイクロベンチマークに使用するテーブル

テーブル名	サイズ	要素数（万）	容量（MB）
t28m	Small	2800	968
t70m	Middle	7000	2420
t112m	Large	11200	3873

6. Experiments

6.1 目的

本論文では GPU での処理を CPU に切り替えるスケジューリングアルゴリズムを実装した．ここでは表 1 に示した各環境下での実行時間を比較することで，スケジューリングによる改善が行えているかを調査する．

6.2 実験環境

マシンを 2 台用意し，クライアントとサーバを分けて実験を行う．サーバは DELL PowerEdge T620 を用いた．メモリは 64 GB, CPU は Intel Xeon E5-2620 2.10 GHz である．GPU は NVIDIA Quadro K5000 を使用する．

6.3 マイクロベンチマークによる評価

簡単な SQL を用いたベンチマークによる提案手法の評価を行う．使用するテーブルは ID と値の 2 つの属性を持ち，値には 1〜10000000 の整数値がランダムに格納されている．表 2 の 3 つのテーブルを使用する．SQL は PG-Storm での並列処理による性能劣化の影響が小さい Join と 性能劣化の影響が大きい Aggregation の 2 種類を用いる．実行する Join, Aggregation は 2.3 節で使用したクエリを用いる．Join に使用するテーブル t100k は 10 万の要素を持つ．

6.3.1 実験方法

各 SQL が扱うテーブルのサイズが異なる場合の実行時間を比較する．並列処理数は 6 に設定する．サイズごとに 2 つの SQL を発行し，合計で 6 つの SQL を並列処理する．実行する SQL のセットを表 3 に示す．表 1 に示す各環境下で 6 つの SQL を同時に実行し，CPU での実行時間，

表 3 マイクロベンチマークで使用する SQL セット

セット名	Small	Middle	Large
JJJ	Join × 2	Join × 2	Join × 2
JJA	Join × 2	Join × 2	Aggr × 2
JAA	Join × 2	Aggr × 2	Aggr × 2
AAA	Aggr × 2	Aggr × 2	Aggr × 2
AAJ	Aggr × 2	Aggr × 2	Join × 2
AJJ	Aggr × 2	Join × 2	Join × 2

GPU での実行時間，全体のターンアラウンドタイムを測定する． 計測は各 SQL セットにつき 10 回行う．

6.3.2 実験結果

実験結果を図 4 に示す．Large テーブルに対して Aggregation を実行する JJA, JAA, AAA においては，Custom は Default と同様のターンアラウンドタイムを達成した．GPU 上での Aggregation の処理を 1 つに制限し，性能の劣化を抑制している．PG-Strom では複数の Aggregation を GPU で処理することで性能劣化が発生し，ターンアラウンドタイムは最大で Default の約 3 倍まで増加した．

Large テーブルに対して Join を実行する JJJ, AAJ, AJJ では，Custom は最も短いターンアラウンドタイムを達成した．GPU 上で並列処理するクエリを最大で 4 つに制限し，GPU リソースを CPU での実行時間が長いクエリに集中させることで GPU 上での実行時間を短縮し，PG-Strom よりもターンアラウンドタイムを短縮できた．

スケジューリングアルゴリズムを導入する事によって，Default よりスループットが低下することを防ぎつつ，PG-Strom よりも高いスループットを達成できた．

6.4 パブリックベンチマークによる評価

意思決定支援システムとしての性能を測定するベンチマークである TPC-H を用いて評価実験を行う．パブリックベンチマークによる測定を行うことで，実システムとして有効であるかを評価する．

6.4.1 実験方法

TPC-H のクエリを並列処理した際のスループットを比較する．TPC-H はデータセットサイズを 20 に設定している．クエリは 12, 13, 14, 16 を使用する．実行するクエリは PG-Strom による GPU 上での動作が確認された中から選出している．4 つのクエリを同時に実行し，各クエリの実行時間，全体のターンアラウンドタイムを測定する．

ベンチマークを実行したところ，実装したスケジューリングアルゴリズムによる切り替えが制御できず，Custom においてクエリ 16 を CPU ではなく GPU で実行していた．そのため，新たな環境として Desired を設定し，スケジューリングアルゴリズムによる切り替えが正しく実行できたと仮定した状態の測定を行い，表 1 の環境と比較する．

6.4.2 実験結果

各環境下においてそれぞれのクエリの処理を担当したコアを表 4 に示し，実験結果を図 5 に示す．クエリ 12,14 では，GPU 上で処理することで実行時間の短縮が可能になるが，Desired では GPU リソースを 2 つのクエリのみに集中させることで PG-Strom と比べ GPU での実行時間を短縮できた．一方でクエリ 13,16 においては，Default と比較すると他の 3 つはすべて実行時間が増加している．クエリ 13,16 の実行時間が増加した原因として，クエリを単一で実行する場合，クエリ 13,16 は CPU での実行時間が GPU での実行時間より短いためであると考えられる．提案手法による切り替えが正しく実行された状態である Desired は全ての環境下の中で最も短いターンアラウンドタイムを達成できた．

表 4 TPC-H の各クエリの処理を担当したコア

環境	sql 12	sql 13	sql 14	sql 16
Default	CPU	CPU	CPU	CPU
PG-Strom	GPU	GPU	GPU	GPU
Custom	GPU	CPU	GPU	GPU
Desired	GPU	CPU	GPU	CPU

7. Related Work

MemcachedGPU [12], [20] では Memcached[21] への GPU の適用を提案している．Memcached における read は write のように排他処理を必要とせず並列して実行することができるため，MemcachedGPU では少量の write を CPU 上で処理し，大量の read を GPU 上で並列処理することによって処理の高速化を図っている．しかし，GPU のリソースを超える量の read 命令が発行された場合でも GPU 上で処理を実行しようとするため，GPU 上でリソース競合が発生し性能が劣化する恐れがある．

MultiQx-GPU [11] は，GPU 上での複数クエリの並列処理機能をサポートする機構を提案する．GPU 上のメモリやクエリの実行順序を管理することで，GPU のリソースをクエリが必要なメモリを確保しつつ効率的に GPU を活用し，GPU 上での複数のクエリの実行における性能劣化を抑制する．こちらの場合でも GPU で処理するクエリが大量に発行された場合，CPU での実行に切替えることなく全て GPU で処理しようとするため，クエリの実行待ちが発生するだけでなく CPU リソースの余剰が発生し，リソースを有効に使用できない状態に陥ってしまう．

今回提案した手法では状況に応じて GPU での実行を CPU に切替えることによって GPU 上のリソース競合を防ぎ，性能の劣化を抑制している．

8. Current Status

本論文では DBMS の複数のクエリを GPU 上で並列処理することで発生する性能劣化を防ぐ手法として，GPU を活用する DBMS におけるリソース競合を考慮したクエリスケジューリングを提案した．PG-Strom 上に提案手法を

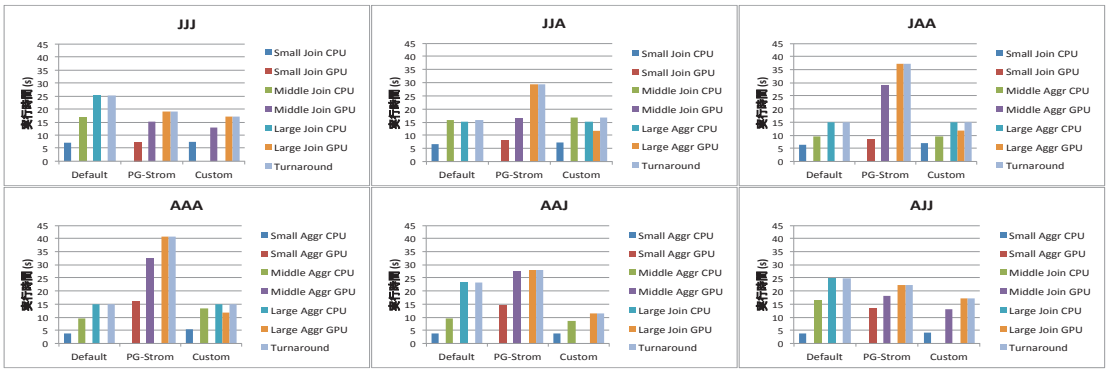

図 4 マイクロベンチマーク 実験結果

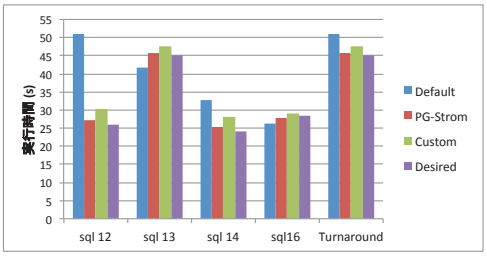

図 5 パブリックベンチマーク 実験結果

実装し，実験から GPU での実行を CPU に切替えるクエリスケジューリングの有用性を確認できた．今後は異なる GPU を活用する DBMS に対して提案手法を適用しつつ，GPU を活用する多くの DBMS に適用できるスケジューリングフレームワークの実現を目指す．

参考文献

[1] Oracle: Oracle Database, available from ⟨http://www.oracle.com/⟩ (accessed 2016-1-10).
[2] NVIDIA: CUDA, available from ⟨http://www.nvidia.com⟩ (accessed 2015-10-25).
[3] Khronos Group: OpenCL, available from ⟨https://www.khronos.org/opencl/⟩ (accessed 2016-8-17).
[4] Bakkum, P. and Skadron, K.: Accelerating SQL database operations on a GPU with CUDA, *General-Purpose Computation on Graphics Processing Units (GPGPU)*, ACM, pp. 94–103 (2010).
[5] Agrawal, S. R., Pistol, V., Pang, J., Tran, J., Tarjan, D. and Lebeck, A. R.: Rhythm: harnessing data parallel hardware for server workloads, *Architectural support for programming languages and operating systems (ASPLOS)*, pp. 19–34 (2014).
[6] He, B. and Yu, J. X.: High-throughput transaction executions on graphics processors, *Very Large Data Bases (VLDB)*, Vol. 4, No. 5, pp. 314–325 (2011).
[7] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B. and Witchel, E.: PTask: operating system abstractions to manage GPUs as compute devices, *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 233–248 (2011).
[8] PostgreSQL Global Development Group: PostgreSQL, available from ⟨https://www.postgresql.org/⟩ (accessed 2015-10-25).
[9] Kaigai, K.: PG-Strom, NEC, available from ⟨http://kaigai.hatenablog.com/entry/2014/11/11/231717⟩ (accessed 2015-10-25).
[10] Fang, W., He, B. and Luo, Q.: Database compression on graphics processors, *Very Large Data Bases (VLDB)*, Vol. 3, No. 1-2, pp. 670–680 (2010).
[11] Wang, K., Zhang, K., Yuan, Y., Ma, S., Lee, R., Ding, X. and Zhang, X.: Concurrent analytical query processing with GPUs, *Very Large Data Bases (VLDB)*, Vol. 7, No. 11, pp. 1011–1022 (2014).
[12] Hetherington, T. H., O'Connor, M. and Aamodt, T. M.: MemcachedGPU: scaling-up scale-out key-value stores, *ACM Symposium on Cloud Computing (SoCC)*, pp. 43–57 (2015).
[13] Heimel, M., Saecker, M., Pirk, H., Manegold, S. and Markl, V.: Hardware-oblivious parallelism for in-memory column-stores, *Very Large Data Bases (VLDB)*, Vol. 6, No. 9, pp. 709–720 (2013).
[14] MonetDB Developer Team: MonetDB, available from ⟨https://www.monetdb.org/⟩ (accessed 2016-8-17).
[15] Zhang, K., Wang, K., Yuan, Y., Guo, L., Lee, R. and Zhang, X.: Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores, *Very Large Data Bases (VLDB)*, Vol. 8, No. 11, pp. 1226–1237 (2015).
[16] Heimel, M. and Markl, V.: A First Step Towards GPU-assisted Query Optimization., *Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, Vol. 2012, pp. 33–44 (2012).
[17] Pirk, H., Manegold, S. and Kersten, M.: Waste not… Efficient co-processing of relational data, *International Conference on Data Engineering (ICDE)*, pp. 508–519 (2014).
[18] He, J., Zhang, S. and He, B.: In-cache query co-processing on coupled CPU-GPU architectures, *Very Large Data Bases (VLDB)*, Vol. 8, No. 4, pp. 329–340 (2014).
[19] He, J., Lu, M. and He, B.: Revisiting co-processing for hash joins on the coupled CPU-GPU architecture, *Very Large Data Bases (VLDB)*, Vol. 6, No. 10, pp. 889–900 (2013).
[20] Hetherington, T. H., Rogers, T. G., Hsu, L., O'Connor, M. and Aamodt, T. M.: Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems, *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 88–98 (2012).
[21] Danga Interactive: Memcached, available from ⟨http://memcached.org/⟩ (accessed 2016-8-17).