DEIM Forum 2021 B33-4

並列プリミティブを活用したGPU上での SQL/Row Pattern Recognitionの処理手法

大原　翼† 宮崎　純†

† 東京工業大学情報理工学院情報工学系　〒 152–8552 東京都目黒区大岡山 2–12–1
E-mail: †ohara@lsc.cs.titech.ac.jp, ††miyazaki@cs.titech.ac.jp

あらまし　本研究では，並列プリミティブを始めとした既に実装されている GPU の技術を組み合わせ，SQL/Row Pattern Recognition の高速な問い合わせ処理を実現する処理手法を提案する．評価実験では，PostgreSQL を用いた SQL/RPR の実装と提案手法を比較した．評価実験の結果から並列プリミティブなどの技術を組み合わせることで，GPU 上で SQL/RPP の処理を効率的に行うことが可能であることが判明した．また，実行時間において提案手法は比較手法と比べ約 7.1-22.6 倍の性能向上を達成した．

キーワード　GPU，並列プリミティブ，大規模データ処理，SQL/RPR，シーケンスデータ，行パターンマッチング

1 はじめに

近年，情報通信技術，センサ技術の発展に伴い，日々大量のデータが生成されている．こうしたデータの中には，データの連続性が重要な意味を持つシーケンスデータが多く含まれている．シーケンスデータは，時系列など順序に従い並んでいるデータのことである．例えば，“株価チャート”やコンピュータシステムに対する“アクセスログ”がシーケンスデータとして挙げられる．

このシーケンスデータに対して，ユーザが指定した特定のパターンが含まれるかどうかを検出する技術として行パターンマッチングが存在する．例えば，“株価チャート”に対し，「価格が一度下がり始め，その後価格が上昇した瞬間」を抽出したい場合，行パータンマッチングを行うことで抽出することができる．このようにシーケンスデータに対する行パターンマッチングという技術は非常に有効であり，社会においても重要なものである．

データベースに蓄積されたシーケンスデータに対する行パターンマッチングは，SQL/RPR (Row Pattern Recognition) [1] として SQL:2016 において標準化されている．SQL/RPR の主な処理としては次の四つが挙げられる．(1) 処理対象データのグループ化，(2) 各グループ毎にユーザが指定した順序に基づき並び替える，(3) 抽出したいパターンを正規表現を用い定義する，(4) 各グループ毎にパターンマッチングを行う．この行パターンマッチングは単純に実装した場合，その処理には多くの時間がかかる．そのため行パターンマッチングを効率的に，また高速に処理する方法が求められている．

一方，シーケンスデータをはじめとする大規模なデータを処理する技術として MapReduce [2] が存在する．MapReduce は分割された大量のデータをクラスタで分散処理するためのプログラミングモデルおよびフレームワークのことである．そのプログラミングが容易であることから，クラスタ上ではなくマルチコア CPU 上で実装した Phoenix [3] や，MapReduce を GPU 上で実装するためのフレームワーク Mars [4] が存在する．また，MapReduce を行うためのフレームワークとして Apache Hadoop [5] も存在する．

また，大規模データ処理の技術としてメニーコアプロセッサである GPU を汎用計算に利用する GPGPU(General-purpose Computing on Graphics Processing Units) が存在する．本来，GPU は膨大な単純処理を必要とする画像処理に用いられていたプロセッサであるが，その並列計算能力の高さから GPU を汎用計算に活用する研究が行われてきた．GPU を汎用計算に活用する研究として，並列プリミティブが存在する．並列プリミティブはソートや配列の総和を求めるリダクションといった汎用計算を GPU 上で実装したものである．並列プリミティブが実現している計算は原始的ではあるが，GPU のハードウェアアーキテクチャに最適な形で処理されるよう最適化されている．

現在，SQL/RPR を実装しているプラットフォームとして Oracle Database 12c [6] などが存在するが，SQL/RPR の処理において上記 GPU の技術を活用している手法はまだ存在していない．

本研究では，並列プリミティブを始めとした GPU の技術を組み合わせ，処理の高速化，最適化を目的とした GPU 上での SQL/RPR の処理手法を提案する．特に先述した SQL/RPR の四つの主な処理を GPU 上で実装する．評価実験では，まず比較手法として PostgreSQL 上で SQL/RPR を実装する．そして SQL/RPR のユースケースのクエリを用い提案手法と比較手法を比較することで性能を評価する．

2 関連研究

2.1 並列プリミティブ

並列プリミティブはソートやリダクションといった汎用計算を GPU 上で実装したものである．この並列プリミティブを実装したライブラリとして現在，ModernGPU [7] や CUDPP [8]

などが知られている．本研究で用いた並列プリミティブの一部について以下に述べる．

hashtable

この並列プリミティブはハッシュテーブルを GPU 上で実現したものである．GPU 上のハッシュテーブルに key と value のペアを挿入し，挿入時 key はハッシュ関数によってハッシュ化される．ある key に対し複数の value を格納することも可能であり，例えば value を取り出す際 key1 を指定すると key1 のペアに対応する全ての value を出力する．

CUDPP の hashtable のハッシュ化は (1) で表される式により計算される．

$$\begin{aligned} \text{ハッシュ値} = &((\textit{random}\text{定数 } \textit{1}\hat{}\textit{key}) \\ &+ \textit{random}\text{定数 } \textit{2}) \ \% \text{ 素数定数} \end{aligned} \tag{1}$$

ここで，異なる値の二つの key から生成したハッシュ値が同じ値になることをハッシュ衝突と呼ぶ．本研究では，key の種類数とハッシュ値の種類数を比較しハッシュ衝突が発生したか確認し，ハッシュ衝突が発生した場合ランダム定数 1 および 2 を変更し再度ハッシュテーブルを構築する処理を例外処理として追加した．

segmented sort

この並列プリミティブは配列を複数のセグメントに区切り各セグメント内でソートを独立に行う．入力にはセグメント毎にソートを行いたい配列とセグメント間隔を定義する配列の二つである．

2.2 PG-Strom

PG-Strom [9] は PostgreSQL 向けに設計された拡張モジュールであり，GPU を利用することで大規模なデータセットに対する集計・解析処理やバッチ処理向けの SQL ワークロードを高速化することができる．PG-Strom は，SQL 命令から自動的に GPU プログラムを生成するコードジェネレータと，SQL ワークロードを GPU 上で非同期かつ並列に実行する実行エンジンからなる．また，GPU 処理にアドバンテージがある場合には PostgreSQL 標準の実装を置き換えることで、ユーザやアプリケーションからは透過的に動作する．

2.3 Spark SQL 上での SQL/RPR の実現

中挾ら [10] はデータベースなどに格納されたシーケンスデータに対する行パターンマッチングの処理コストを削減する手法を提案した．処理コストを削減する手法として Sequence Filtering と Row Filtering を提案した．Sequence Filtering は PARTITION BY 句によりグループ分けされたのち，DEFINE 句において PREV( ) や NEXT( ) といった行パターン変数を使用しないマッピング条件によりマッピングされる行が一行もないグループは行パターンマッチングの対象外とする手法である．Row Filtering は DEFINE 句における行パターン変数のマッピング条件によりマッピングされた行とその前後数行だけを残し，その以外の行はフィルタリングする．これにより行パターンマッチングの対象となる行数が削減され，処理コストが削減される．評価実験では，SQL/RPR を PostgreSQL と Spark に実装し，提案手法が行パターンマッチングの処理コストを削減したことを示した．

Table

item	tradedate	price
item1	2021-02-05	152.00
item1	2021-02-15	142.00
item1	2021-02-17	142.00
item1	2021-02-18	138.00
item1	2021-02-19	144.00
item1	2021-02-21	150.00
item1	2021-02-22	152.00
item2	2021-02-13	335.00
item2	2021-02-14	329.00
item2	2021-02-19	329.00
item2	2021-02-20	326.00
item2	2021-02-22	320.00
item2	2021-02-25	326.00
item2	2021-02-27	328.00
item3	2021-02-05	148.00
item3	2021-02-10	153.00
item3	2021-02-13	151.00
item3	2021-02-20	326.00

図 1　データベース例

3 SQL/RPR の機能

本節では，SQL:2016 において標準化されている SQL/RPR の機能について述べる．図 1 にデーターベースの例を示す．データベースは，item(取引された商品のカテゴリ)，tradedate(商品が取引された年月日)，price(商品の取引価格) の三つのカラムを保持している．図 2 は図 1 のデーターベースに対し SQL/RPR のクエリを発行した例を示している．図 2 のクエリでは，取引価格が一回以上下がり続け，その後取引価格が一回以上上がり続けるパターン，いわゆる V 字型のパターンが抽出される．

SQL/RPR の機能を用いるためには MATCH_RECOGNIZE 句を使用する．MATCH_RECOGNIZE 句内では，PARTITION BY 句，ORDER BY 句，MEASURES 句，ONE ROW PER MATCH 又は，ALL ROWS PER MATCH，AFTER MATCH SKIP 句，PATTERN 句，SUBSET 句，DEFINE 句を用いてパターンの抽出を行う．それぞれの句についてその機能を述べる．

- PARTITION BY 句：指定したカラム名の値が同じレコード同士をグループにする．
- ORDER BY 句：PARTITION BY 句でグループ分けされたグループ毎に ORDER BY 句で指定したカラムに基づきソートする．
- MEASURES 句：結果出力テーブルのカラムとなる行パターン測定列を定義する．
- ONE ROW PER MATCH(又は，ALL ROWS PER MATCH)：抽出したパターン毎に結果を 1 行に集約した形で (又は，パターンとして抽出された行を全て) 出力する．
- AFTER MATCH SKIP 句：空でないパターンの抽出に成功した後，次のパターンマッチングを再開する場所 (行の

```
SELECT *
FROM Table
   MATCH_RECOGNIZE
   (
      PARTITION BY item
      ORDER BY tradedate
      MEASURES
         CLASSIFIER() AS classy,
         A.tradedate AS startdate,
         A.price AS startprice,
         LAST(B.tradedate) AS bottomdate,
         LAST(B.price) AS bottomprice,
         LAST(C.tradedate) AS enddate,
         LAST(C.price) AS endprice,
         MAX(U.price) AS maxprice
      ALL ROWS PER MATCH
      AFTER MATCH SKIP PAST LAST ROW
      PATTERN (A B+ C+)
      SUBSET U = (A, B, C)
      DEFINE
         B AS B.price < PREV(B.price),
         C AS C.price > PREV(C.price)
   )
```

図 2　SQL/RPR のクエリ例

位置) を指定する．

- PATTERN 句：正規表現を用い抽出したいパターンを定義する，正規表現には行パターン変数と {+，＊ } などの量指定子を用いる．
- SUBSET 句：行パターン変数の集合リストを定義する．
- DEFINE 句：行に対しどの行パターン変数をマッピングするかの条件を定義する，行パターン変数のマッピング条件には，PREV( )，NEXT( ) といった 1 レコード前や後を指定する関数，AVG( )，MAX( )，COUNT( ) といった集約関数などを用いることができる．

4　提 案 手 法

本節では，GPU を用い SQL/RPR の高速な問い合わせ処理を実現する手法について提案する．特に SQL/RPR の主な処理である (1) 処理対象データのグループ化，(2) 各グループ毎にユーザが指定した順序に基づき並び替える，(3) 抽出したいパターンを正規表現を用い定義する，(4) 各グループ毎にパターンマッチングを行う，の四つ処理を GPU 上で実行するための手法について述べる．提案手法のワークフローを図 3 に示す．提案手法のワークフローは次の通りである．CPU 上でデータベースに格納されている処理対象データを GPU 上に転送し，そのデータをパーティションする．その後，パーティションされた各データグループ毎にユーザが指定した順序に従いソートする．各データグループの各行に行パターン変数を DEFINE 句に基づき暫定的にマッピング (以下，暫定マッピングと呼ぶ) する．ユーザが指定した AFTER MATCH SKIP 句のオプションに従い，各データグループ毎に並列でパターンマッチングを行う．このパターンマッチングによりパターンにマッチした“連続している行”のオフセットを取得する．そのオフセットとデータベースを照らし合わせマッチしたパターンを CPU 上に表示する．

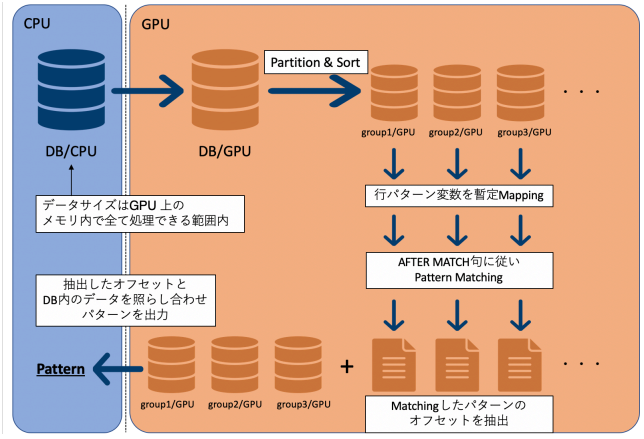

図 3　提案手法のワークフロー

GPU 上に転送された処理対象データはデータベースのカラム毎に配列として保持する．ここでは，CPU 上のデータベースでの id を保持する配列を ID 配列と呼ぶことにする．

また，本研究では，処理対象のデータサイズは GPU 上のメモリ内で全て処理できる範囲内であるとする．

4.1　データベースのパーティション

SQL/RPR における PARTITION BY 句の機能に相当する GPU 上での処理について述べる．データベースのパーティションを GPU 上で実行するために，ハッシュテーブルを用いたパーティションを行う．データベースのパーティションはユーザが指定したカラム名 (以下，パーティションカラムと呼ぶ．) の値が同じ列をグループとしてまとめる．パーティションを行うためにパーティションカラムを保持した配列と ID 配列の二つを用いる．パーティションカラムの値を key，CPU 上のデータベースにおける id を value としハッシュテーブルに格納する．並列プリミティブである CUDPP の hashtable を用いることでパーティションカラムの値毎にグループ分けを行う．CUDPP の hashtable は二つの配列を出力する．一つ目の配列は，(2) で表される通りまとめられたグループが連続で並んだものを表している．

$$ID\text{ 配列 } = (group1\_1, group1\_2, \ldots, group2\_1, group2\_2, \ldots, group3\_1, group3\_2, \ldots) \quad (2)$$

二つ目の配列は，(3) で表される通り一つ目の配列における各グループの先頭要素の配列のインデックスを表したものである．しかし先頭グループの先頭要素のインデックスは出力されない．この配列を以下 SegHead 配列と呼ぶことにする．

$$SegHead\text{ 配列 } = (group2\_1\text{ の }ID\text{ 配列における }index, group3\_1\text{ の }ID\text{ 配列における }index, \ldots) \quad (3)$$

データベースのパーティションの処理例を図 4 に示す．図 4 では，パーティションカラムを item(取引された商品のカテゴリ) としている．

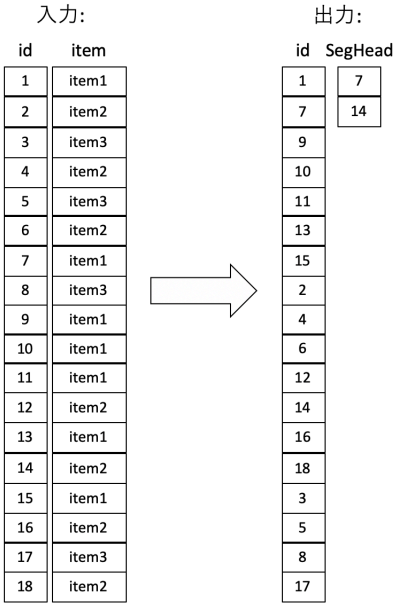

図 4　データベースのパーティションの処理例

CUDPP の hashtable により，グループ毎に並んだ ID 配列と SegHead 配列を出力し，データベースのパーティション処理が完了する．

4.2　グループ毎のデータ並び替え

SQL/RPR における ORDER BY 句の機能に相当する GPU 上での処理について述べる．パーティションされた各データグループをユーザが指定したカラム名 (以下，ソートカラムと呼ぶ．) に従った並び替えは，ソートカラムを保持する配列と SegHead 配列，さらに 4.1 節で出力されたグループ毎に並んだ ID 配列の三つを用い処理を行う．並び替えは各データグループ内でそれぞれ独立に行われる必要がある．そのため並列プリミティブである ModernGPU の segmented sort を用いる．segmented sort はセグメント毎にソートを行いたい配列 A とセグメント間隔を定義する配列 B を入力に取り，各セグメント内でソートを行った配列 A を結果として出力する．また，配列 A のインデックスも同時にセグメント毎にソートされ結果として出力される．

本手法では，入力の配列 A にソートカラムを保持する配列，配列 B に SegHead 配列を選択する．これによりソートカラムを保持する配列がセグメント毎にソートされる．またソートカラムを保持する配列のインデックスがセグメント毎にソートされた配列も出力され，この配列に従い 4.1 節で出力されたグループ毎に並んだ ID 配列を並び替える．これによりグループ毎に並び，かつグループ内がソート済みである ID 配列が出力され，グループ毎のデータ並び替え処理が完了する．

4.3　正規表現を用いた抽出パターンの定義

SQL/RPR における DEFINE 句の機能に相当する GPU 上での処理について述べる．SQL/RPR の内部処理では，一番最初の行から順に (1)DEFINE 句に基づく行パターン変数の暫定マッピング，(2) この時点でパターンにマッチする可能性があるか確認，(3) パターンにマッチした場合暫定マッピングされた行パターン変数を正式にマッピングする，あるいはマッチしなかった場合暫定マッピングされた行パターン変数を取り消す，といった一連の処理を一つのパターンのマッチに成功するか失敗するまで行う．しかし，本手法では，(1)DEFINE 句に基づく行パターン変数の暫定マッピング，を (2) パターンにマッチする可能性があるか確認，の前にあらかじめ全ての行に対し並列に行う．暫定マッピングは GPU 上の各スレッドが処理対象データの各行に対して一対一で処理を行うことで並列計算を実現する．

```
AFTER MATCH SKIP PAST LAST ROW
PATTERN (A+ B+)
DEFINE    A AS PREV (A.Price + A.Tax) < 100
          B AS PREV (B.Price) >= 100
```

図 5　クエリのユースケース 1

ユーザが定義する行パターン変数のマッピング条件は SQL:2016 において指定されている複数の関数を組み合わせて行う．そのため DEFINE 句のパターン数は無数に存在する．しかし，本研究では考え得るすべてのパターンには対応していない．そこで SQL:2016 において指定されている関数を用いた DEFINE 句のユースケースを五つ準備した．ユースケースは図 5 から図 9 に示す．これらユースケースは SQL:2016 において指定されている関数をおおよそ用いている．そのためユーザが定義する DEFINE 句は無数に存在するが，それらは準備したユースケースの DEFINE 句に近い形になると考えることができる．本研究では，ユースケースの DEFINE 句をそれぞれ実装し，また評価実験を行う．

各行への暫定マッピングは各データグループ毎に独立して行う必要がある．データグループ間をまたぐ暫定マッピングを防ぐために 4.2 節で出力された ID 配列と 4.1 節で出力された SegHead 配列を用い，各行並列にかつ各グループ独立に行パターン変数の暫定マッピングを行う．

ここで，ユースケース 3，4 において用いられている関数 First( ) 関数や集約関数は正式に行パターン変数がマッピングされている行に対し操作を行う関数である．このような関数が DEFINE 句で用いられている場合は，パターンマッチング前に暫定マッピングを行うことができない．この場合は暫定マッピングを行うことができる条件のみあらかじめ暫定マッピングを行う．ユースケース 3 においては行パターン変数 A の暫定マッピングを，ユースケース 4 においては行パターン変数 A と B の暫定マッピングをあらかじめ実行する．図 7 のユースケース 3 における行パターン変数 B の暫定マッピング，図 8 のユースケース 4 における行パターン変数 C の暫定マッピングなどあらかじめ行うことができないマッピングは次の処理ステップであるパターンマッチング時に行う．その処理方法に関しては 4.4 節で述べる．

4.4　処理対象データからパターン抽出

SQL/RPR における PATTERN 句の機能に相当する GPU

```
AFTER MATCH SKIP PAST LAST ROW
PATTERN (A+ B+)
DEFINE    A AS NEXT (A.Price + A.Tax) < 100
          B AS NEXT (B.Price) >= 100
```

図 6　クエリのユースケース 2

```
AFTER MATCH SKIP PAST LAST ROW
PATTERN (A+ B+)
DEFINE    A AS A.Price < 200
          B AS B.Price + B.Tax > FIRST (A.Price) + 200
```

図 7　クエリのユースケース 3

```
AFTER MATCH SKIP PAST LAST ROW
PATTERN (A* B+ C)
DEFINE    A AS A.Price + A.Tax > 200
          B AS B.Price < 100
          C AS C.Tax < MAX (A.Tax)
```

図 8　クエリのユースケース 4

```
AFTER MATCH SKIP PAST LAST ROW
PATTERN ( (A | B) + C )
DEFINE    A AS A.Price >= 150
          B AS B.Price < 100
          C AS CASE
                    WHEN PREV (CLASSIFIER ()) = 'A'
                    AND C.Price >= 100 AND C.Price <130
                      THEN 1
                    WHEN PREV (CLASSIFIER ()) = 'B'
                     AND C.Price >= 130 AND C.Price <150
                      THEN 1
                    ELSE 0
```

図 9　クエリのユースケース 5

上での処理について述べる．行パターン変数が暫定マッピングされた処理対象データからパターンを抽出する処理では CUDA-grep [11] のソースコードを改変したものを用いる．CUDA-grep は一つのマッチング対象データに対し，複数の正規表現パターンのマッチングを並列に行う．しかし本研究では，パーティション化された複数のグループに対し PATTERN 句で定義した一つの正規表現パターンのマッチングを行う．そのため，CUDA-grep を複数のマッチング対象データに対し GPU 上のスレッドが並列にパターンマッチングを行うように変更した．

パターンマッチング前にあらかじめ暫定マッピングできない行パターン変数がユースケース内に存在する場合の処理について述べる．図 7 のユースケース 3，図 8 のユースケース 4 がそれに該当する．

ここでは，処理例としてユースケース 4 の行パターン変数 C の暫定マッピングを用い説明する．ユースケース 4 の抽出パターンは (A* B+ C) であり，既に行パターン変数 A，B は暫定マッピングされている．まず，パーティションされたグループ内の一番上の行から順に処理していきパターン (A* B+) にマッチするまでパターンマッチングを行う．そして直前の行までパターン (A* B+) にマッチしており，かつ現在処理している行に行パターン変数が何もマッピングされていない場合に MAX( ) 関数の計算を行う．ユースケース 4 の MAX( ) 関数は現在行パターン変数 A が暫定マッピングされている全ての行の中で一番高い Tax の値を求める関数である．この時，並列プリミティブである ModernGPU の MaxReduce を用いる．MaxReduce は入力配列の中で最も大きい値を出力する並列プリミティブである．これにより行パターン変数 C のマッピング条件を計算でき，現在処理している行に行パターン変数 C をマッピングするか判断することが可能になる．以上がパターンマッチング前にあらかじめ暫定マッピングできない行パターン変数が存在する場合の処理となる．

パターンマッチングが最終行まで完了した時，どの行がマッチしたかを示すオフセット配列を出力する．このオフセット配列は各パターン毎にパターン番号が振られている．パターン番号はパターンにマッチするたびに「1」インクリメントされるシーケンシャルな番号であり，マッチしたパターン毎にそのパターンに含まれる行全てに振られる．また，パターン番号はグループ毎に独立している．つまり，最後に振られたパターン番号がそのグループ内のパターンの総マッチ数となる．

オフセット配列とデータベースを照らし合わせマッチしたパターンを最終結果として出力し，全ての処理が完了となる．

5　評 価 実 験

本節では，提案手法と PostgreSQL 上で SQL/RPR の処理を実装した比較手法を比較し評価を行う．

5.1　比 較 手 法

比較手法として用いる PostgreSQL 上での SQL/RPR の処理手法の実装方法について述べる．比較手法の実装には以下の五つのタスクが存在する．

Partition&Sort
PARTITON BY 句と ORDER BY 句に従い処理対象データをグループ化，その後各グループ内でソート．

Mapping RPV
行毎に DEFINE 句に基づき行パターン変数の暫定マッピングを行う．

Concat RPV
行毎に暫定マッピングされた行パターン変数をグループ毎に連結し，長大な文字列に変換する．このタスクの処理例について図 10 に示す．

Pattern Matching
連結された行パターン変数列に対し，パターンマッチングを行う．Concat RPV タスクで出力されたグループ毎の行パターン変数列を入力とし，1 から始まる連番である *id*，パターンの出現位置にパターン番号が代入された *is_match* の二つのカラムを保持するテーブルを出力する．

Collecting Results
Pattern Matching の結果に従い、最終結果を出力する．Pat-

item	tradedate	price	RPV
item1	2021-02-05	152.00	A
item1	2021-02-15	142.00	B
item1	2021-02-17	142.00	A
item1	2021-02-18	138.00	B
item1	2021-02-19	144.00	C
item1	2021-02-21	150.00	C
item1	2021-02-22	152.00	C
item2	2021-02-13	335.00	A
item2	2021-02-14	329.00	B
item2	2021-02-19	329.00	A
item2	2021-02-20	326.00	B
item2	2021-02-22	320.00	B
item2	2021-02-25	326.00	C
item2	2021-02-27	328.00	C
item3	2021-02-05	148.00	A
item3	2021-02-10	153.00	C
item3	2021-02-13	151.00	B
item3	2021-02-20	326.00	C

item	RPVS
item1	ABABCCC
item2	ABABBCC
item3	ACBC

図 10　Concat RPV タスクの処理例

tern Matching タスクで出力されたテーブルと Mapping RPV タスクで出力されたテーブルを元に最終結果を出力する．

上記の五つのタスクの中で Pattern Matching 以外のタスクは SQL クエリのみで全ての処理を行っている．Pattern Matching タスクに関しては SQL クエリのみでは実装できないため本研究では C++で実装した．SQL/RPR を処理する比較手法をまとめた擬似 SQL クエリを Algorithm1 に示す．

5.2 実験準備

評価実験に用いたマシンの構成を表 1 に示す．

表 1　実験に用いたマシンの構成

CPU		Intel Core i7-6800K (3.4GHz，6 コア)
RAM		32GB
GPU		NVIDIA TITAN X (Pascal)
	CUDA コア数	3584
	ベースクロック	1.53GHz
	メモリ量	12GB
CUDA		CUDA 8.0
OS		CentOS 7.7
データベース		PostgreSQL 9.2

また，本実験で用いた PostgreSQL はパラメータをチューニングしている．変更後のパラメータの値を表 2 に示す．

表 2　PostgreSQL のパラメータ

パラメータ	変更前	変更後
max_connections	100	4
shared_buffers	32MB	10GB
work_mem	設定なし	5GB

5.3 実験内容

本実験では，実行時間の計測範囲として CPU 上のデータベースにデータが読み込まれている状態から，計算結果が CPU 上のメモリに書き込まれるまでとした．そのため CPU-GPU 間のデータ転送時間も実行時間に含まれる．

CPU 上のデータベースにはテストテーブルを設計している．テストテーブルは四つのカラムを保持しており，それらは item(取引された商品のカテゴリ)，tradedate(商品が取引された年月日)，price(商品の取引価格)，tax(商品の取引価格の税金) である．テストテーブルに読み込むデータには人工データを用いる．各カラムのデータはそれぞれ一様分布に従う乱数とし，全体サイズは 870MB，1.7GB，10GB の三種類となるようにした．三種類のデータベースのレコード数はそれぞれ 17,350,000，34,700,000，208,200,000 である．また，item カラムのデータの種類は 10 種類となるようにした．

実験で用いるクエリを図 11 に示す．

AFTER MATCH SKIP 句，PATTERN 句，DEFINE 句には，図 5 から図 9 に示した五つのユースケースを用いた．

5.4 実験結果

提案手法における実行時間を各タスク毎に計測した．タスクに

Algorithm 1 SQL/RPR を処理する比較手法の擬似 SQL クエリ

```
Require: input_table
  // Partition&Sort
  table1 ← SELECT * FROM input_table
           ORDER BY partition_column, sort_column;
  // Mapping RPV
  table2 ← SELECT *,
              CASE WHEN [map_condition1] THEN 'rpv1'
                   WHEN [map_condition2] THEN 'rpv2'
                                         :
              // map_condition には rpv のマッピング条件を
                 記述
                   ELSE 'E'
              END AS rpv
           FROM table1;
  ALTER TABLE table2 ALTER rpv TYPE varchar;
  //次タスクでは varchar でしか扱えないため
  // Concat RPV
  table3 ← SELECT partition_column,
                  string_agg(rpv, '') AS rpvs
           FROM table2
           GROUP BY partition_column ORDER BY partition_column;
  // Pattern Matching
  table4 ←pattern_matching(table3)
  // table4：連番である id，パターンの出現位置にパターン番号が
     代入された is_match を保持する
  // Collecting Results
  table5 ← SELECT *, row_number() OVER() FROM table2;
  //連番を振り直す
  output_table ←SELECT partition_column, sort_column,
                       other_column, rpv, is_match
              FROM table5
              JOIN table4 ON table5.row_number = table4.id
              WHERE is_match != 0;
              // is_match = 0 の行はパターンにマッチ
                 していない行，最終結果には表示しない
  return output_table
```

```
SELECT *
FROM Table
  MATCH_RECOGNIZE
  (
    PARTITION BY item
    ORDER BY tradedate
    MEASURES
      CLASSIFIER() AS classy,
    ALL ROWS PER MATCH
    AFTER MATCH SKIP clause
    PATTERN clause
    DEFINE clause
  )
```

図 11　評価実験に用いるクエリ

は copyHostToDevice，Partition，Sort，Mapping RPV，Pattern Matching，copyDeviceToHost がある．copyHostToDevice は CPU 上のデータを GPU に転送するタスク，Partition は GPU 上で処理対象データのパーティションを行うタスク，Sort はパーティションされた各データグループ毎をソートカラムに従い並び替えるタスク，Mapping RPV は DEFINE 句で定義されている行パターン変数のマッピング条件に従い処理対象データの各行に行パターン変数を暫定マッピングするタスク，Pattern Matching は処理対象データからパターンを抽出するタスク，copyDeviceToHost は GPU 上の計算結果を CPU に転送するタスクである．また，比較手法における実行時間も 5.1 節で述べた各タスク毎に計測した．

ここでは，図 5 に示したユースケース 1 に対する各タスクの実行時間と全タスクの合計時間を図 12 から図 14 に示す．各タスクの実行時間と全タスクの合計時間の図はデータベースのサイズ毎に示す．提案手法は比較手法と比べ約 7.1-16.9 倍の性能となっている．

また，その他のユースケースにおいては，ユースケース 2 では，提案手法は比較手法と比べ約 7.5-17.8 倍の性能，ユースケース 3 では，提案手法は比較手法と比べ約 12.0-22.6 倍の性能，ユースケース 4 では，提案手法は比較手法と比べ約 10.5-22.5 倍の性能，ユースケース 5 では，提案手法は比較手法と比べ約 10.3-20.2 倍の性能となっている．

提案手法では Sort タスクと Pattern Matching タスクの実行時間がデータベースサイズの変化により大きく変化が見られる．比較手法では全てのタスクでデータベースサイズの増加に伴い実行時間も増加している．

提案手法では全体の実行時間の内 Pattern Matching タスクが支配的となっている．表 3 に提案手法のユースケース毎の Pattern Matching タスクの実行時間を示す．ユースケース 4 の実行時間はユースケース間において僅かに差が見られるが，抽出パターンがその他のユースケースと違うことが一つの要因として考えられる．

また，提案手法の Partition，Sort，Mapping RPV タスクは全体の実行時間に及ぼす影響は少なかった．一方，比較手法では Partition，Sort，Mapping RPV タスクは全体の実行時間に比べ無視できる範囲ではない．従い，Partition，Sort，Mapping RPV タスクは提案手法で極めて高速化されたと言える．

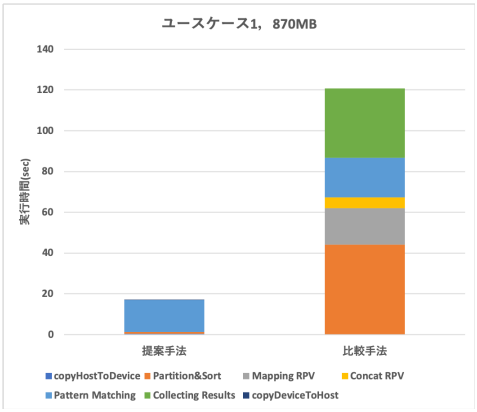

図 12　ユースケース 1，870MB

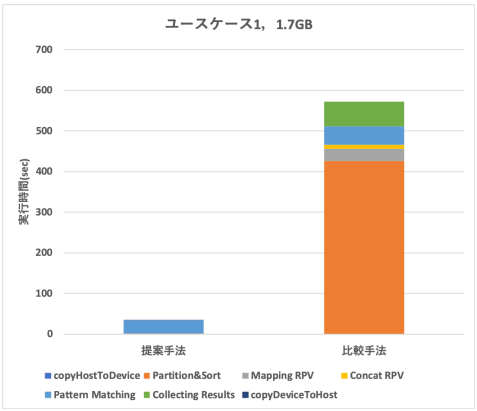

図 13　ユースケース 1，1.7GB

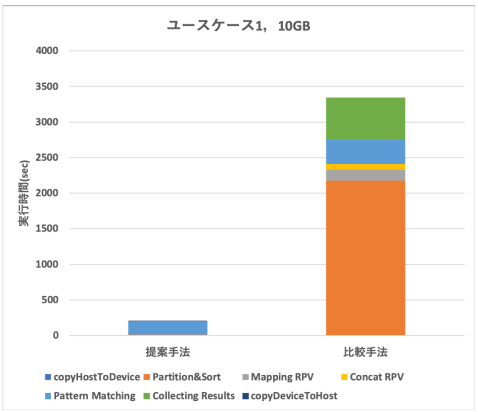

図 14　ユースケース 1，10GB

表 3　各ユースケースにおける提案手法の Pattern Matching の実行時間

実行時間 (sec)	データベースのサイズ		
	870MB	1.7GB	10GB
ユースケース 1	15.617	31.228	186.957
ユースケース 2	15.213	30.325	181.821
ユースケース 3	15.635	31.093	186.157
ユースケース 4	14.411	28.637	172.212
ユースケース 5	15.221	30.291	180.606

5.5　グループ数による実行時間の変化

データベースの item カラムのデータの種類数，つまりグループ数を変化させた時の各タスクの実行時間と全タスクの合計時間を表 4 と表 5 に示す．表 4 と表 5 はデータベースのサイズが 870MB，AFTER MATCH SKIP 句，PATTERN 句，DEFINE 句には，図 5 に示したユースケース 1 を用い計測を行った．また，グループ数が 10，1000，10000 の三パターン計測を行った．

提案手法ではグループ数の増加に伴い Pattern Matching タスクの実行時間が減少した．Pattern Matching タスクはグループ毎に並列にパターンマッチングを行っている．そのためグループ数の増加に伴い並列度の高い処理が行われ実行時間の減少につながったと考えられる．

一方，比較手法ではグループ数の増加に伴い Partition&Sort タスクの実行時間が増加した．Partition&Sort タスクの実行時間が増加した原因としてはグループ数の増加によりパーティション計算が複雑になったことが考えられる．

この結果からパーティションされるグループ数が多い程，並列度の高い提案手法がより有効な処理手法となることが判明した．

表 4　グループ数を変化させた時の提案手法の実行時間

実行時間 (sec)		グループ数		
		10	1000	10000
タスク	copyHostToDevice	0.303	0.298	0.299
	Partition	0.029	0.028	0.029
	Sort	0.999	0.992	0.974
	Mapping RPV	0.001	0.001	0.001
	Pattern Matching	15.617	12.414	8.154
	copyDeviceToHost	0.080	0.081	0.076
合計時間		17.029	13.814	9.533

表 5　グループ数を変化させた時の比較手法の実行時間

実行時間 (sec)		グループ数		
		10	1000	10000
タスク	Partition & Sort	44.129	84.768	103.966
	Mapping RPV	17.891	17.663	17.905
	Concat RPV	5.370	5.083	4.625
	Pattern Matching	19.379	19.708	18.582
	Collecting Results	34.043	38.104	35.229
合計時間		120.812	165.326	180.307

6　おわりに

本研究では並列プリミティブを始めとした GPU の技術を用い SQL/RPR の高速な問い合わせ処理を実現する手法を提案した．

評価実験では，比較手法として PostgreSQL 上で SQL/RPR を実装した．そして SQL/RPR のユースケースのクエリを用い提案手法と比較手法を比較することで性能を評価した．評価実験の結果から並列プリミティブなどの技術を組み合わせることで，GPU 上で SQL/RPP の処理を効率的に行うことが可能であることが判明した．また，実行時間において提案手法は比較手法と比べ約 7.1-22.6 倍の性能向上を達成した．

今後の課題として，ユーザが自由に定義できる DEFINE 句に対し GPU 上での処理を自動生成するコンパイラの構築，AFTER MATCH SKIP 句に存在する複数のオプションの対応，SQL/RPR のクエリの対象となるデータベース上のデータが GPU 上のメモリに全て乗り切らない場合の処理手法の考案が考えられる．

7　謝　　辞

本研究の一部は，JSPS 科研費 (18H03242, 18H03342, 19H01138A) の助成を受けたものである．

文　　献

[1] ISO/IEC 2016. Information technology - Database languages - SQL technical reports - part 5: Row Pattern Recognition in SQL. Technical report, ISO copyright office, 2016.
[2] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters”, Commun. ACM, 51(1):pp. 107-113, 2008.
[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, “Evaluating mapreduce for multi-core and multiprocessor systems”, In Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA ’07), pp. 13-24, Washington, DC, USA, 2007.
[4] B. He, W. Fang, Q. Luo, N. K. Govindaraju, T. Wang, “Mars: A MapReduce Framework on Graphics Processors”, In Proc. of PACT 2008, pp. 260-269, 2008.
[5] Hadoop, https://hadoop.apache.org/ (2021 年 1 月 20 日アクセス)
[6] “Patterns everywhere - Find them fast! SQL pattern matching in Oracle Database 12c an Oracle White Paper”, June 2013.
[7] Sean Baxter: moderngpu 2.0, https://github.com/moderngpu/moderngpu (2021 年 1 月 20 日アクセス)
[8] CUDPP, https://github.com/cudpp/cudpp (2021 年 1 月 20 日アクセス)
[9] PG-Strom, https://heterodb.github.io/pg-strom/ja/ (2021 年 1 月 20 日アクセス)
[10] 中挟 晃介, 北川 博之, “シーケンスデータに対する行パターンマッチングの効率化”, 情報処理学会論文誌, Vol.62, No.1, pp. 302-320, 2021.
[11] CUDA-grep, https://github.com/bkase/CUDA-grep (2021 年 1 月 20 日アクセス)