Vol.2011-HPC-130 No.26
2011/7/28

GPU援用カラムストアデータベースの設計と評価

上村純平[†1]　柏木岳彦[†1]　鳥居隆史[†2]

近年，レコードデータをカラム毎に保存，処理を行うカラムストアデータベースにおけるアクセラレータとして，GPGPU を用いる研究が行われている．本稿では，GPU でカラムストアデータを扱うのに適したデータベースのアーキテクチャを提案する．提案するアーキテクチャでは，データ転送と処理を GPU のストリーム処理によって多重処理する．ストリーム処理を効率的に行うために転送と処理がバランスするようにカラムデータをチャンクに分割し，データ圧縮を行う．さらに，GPU 上にカラムデータをキャッシュすることによって転送時間のオーバーヘッドをなくす．提案アーキテクチャを用いてスキャン処理を実装し， NVIDIA GTX480 と Intel Xeon CPU で評価した．結果，CPU 1 スレッドでの処理時間に比べ，GPU を用いることで 3-25 倍の性能向上が確認できた．

Design and Evaluation of a GPU Accelerated Column Store Database

Junpei Kamimura,[†1] Takehiko Kashiwagi[†1]
and Takashi Torii[†2]

GPGPU appears to be promising as an accelerator of column store databases, which store and execute data by columns. In this paper, we propose a GPU accelerated database architecture suited to handle column of data in a database table. In our proposal, data transfers between host machine, GPU and kernel executions are overlapped by streaming feature of GPU. The column of data is split into smaller chunks to balance the data transfer and execution time. Each chunk data is compressed to reduce the overhead of data transfer. And data caching on GPU allows the system to skip the step of data transfer. We have implemented scan operation on a PC with NVIDIA GTX480 and Intel Xeon CPU. Our GPU implementation is about 3-25 times faster than that on a single-threaded CPU.

1. はじめに

DWH などの読み取り性能が求められるデータベースにおいて，IO 効率のよいカラムストアストレージ[1] を採用したデータベースが注目されている．[2)–5)] カラムストアデータベースでは，論理的な表データを列単位で保存し，クエリ処理を行う．これに対して，従来のロウストアデータベースは，行単位でデータを保存し，クエリ処理を行う．カラムストアデータベースでは，大量のレコードデータに対して，少数の列に条件をつけて絞り込む場合や集約演算を行う場合に IO 効率がよい．同じ処理をロウストアデータベースで行う場合，行全体を読み出し，対象の列データを取り出す必要があるため IO，処理の効率が悪い．一方，レコードデータの挿入や更新に関しては，列数分のデータ更新が必要なカラムストアデータベースよりも 1 回のデータ更新で実行可能なロウストアデータベースの方が効率が良い．それぞれの特徴により，カラムストアデータベースは DWH のような大量データに対する分析処理，ロウストアデータベースは電子商取引のようなトランザクション処理に適している．

また，近年のメモリの大容量化により，全てのテーブル情報をメモリ上に保持するオンメモリデータベースが広く使われるようになっている．オンメモリ，カラムストアのデータベースにより，これまでのディスク IO のボトルネックが解消され，高速なクエリ処理が可能となってきた．さらなる高速化のためには CPU のデータ処理速度に対してメモリのデータ供給速度が小さいことに対応する必要があり[6)]，オンメモリのカラムストアデータベースや，GPU や FPGA などのハードウェアを活用する研究が行われている．

GPU は高いメモリバンド幅と大量のスレッドによる高い並列計算能力を有しており，計算あたりの電力効率の良さや GPU 自体の低廉化により，コスト面でも有望であると考えられる．GPU を用いてデータベースを高速化する研究は盛んに行われており，データベース処理の基本要素であるスキャン，ソート，ジョイン，集約などで効果が報告されている[7)–10)]．また，カラムストアデータベースでは同じ型のデータが連続して並ぶデータ構造になるため，GPU の演算スタイルとマッチしやすく，様々な研究が行われている[11),12)]．しかしながら，大量のデータを扱うデータベースでは，ホスト，GPU 間の限られた帯域でのデータ転送のオーバヘッドが大きく，CPU に対する処理速度のメリットが低下してしまう．

†1 NEC 第三ＩＴソフトウェア事業部
3rd IT Software Division, NEC Corporation
†2 NEC システムプラットフォーム研究所
System Platform Laboratory, NEC Corporation

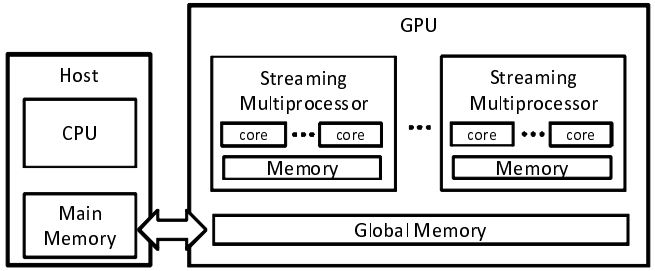

図 1 GPU hardware architecture

本稿では，オンメモリカラムストアデータベースのアクセラレータとして GPU を活用するためのアーキテクチャを提案し，その評価を行う．提案アーキテクチャでは，ホスト，GPU 間の通信路である PCI Express がボトルネックとなるようハードウェアを選択し，データ転送を削減するために GPU 上のメモリにカラムデータをキャッシュする．また，データ転送と処理をストリームによって多重させ，さらにストリーム処理を効率よく行うために，ストリームスケジューラによるカラムデータの分割とデータ圧縮を行う．本稿の貢献は次の通りである．

- GPU をアクセラレータとして用いるカラムストアデータベースにおいて，データ供給の観点からどのハードウェアを使うかの指針を示し，クエリを高速に処理するためのソフトウェアアーキテクチャの提案を行った．
- 提案アーキテクチャを用いて，Fermi 世代の GPU を用いてカラムスキャン処理の性能を評価し，以下のような結果を得た．
  - データ供給の速度律速となる処理において，提案アーキテクチャによるカラムデータのチャンク化，圧縮，ストリーム処理により，処理全体のスループットを PCI Express の転送レートまで近づけられることを示し，1 スレッドの CPU に対して 3-15 倍の性能向上を得た．
  - 提案アーキテクチャによって GPU 上にデータをキャッシュすることでデータ転送のオーバヘッドをなくし，1 スレッドの CPU に対して 25 倍の性能向上を得た．

2. GPGPU

近年の計算機において，GPU はコモディティハードウェアとして広まっている．NVIDIA による統合開発環境 CUDA などによって，GPU を科学計算，行列演算，データベースなどのアプリケーションのアクセレータとして利用する GPGPU (General-purpose computing on GPU) の研究が盛んに行われている．ここでは CUDA 対応 GPU のアーキテクチャについて説明する．なお，世代や型番により，GPU の構造は若干異なる．

図 1 に GPU ハードウェアのアーキテクチャの概略を示す．GPU の基板上には，32 個の CUDA コア群から成る Streaming Multiprocessor (SM) が複数配置されている．例えば GTX480 では 16 基の SM が配置され，内 15 基分の計 480 の CUDA コアが動作する．CUDA コアは GPU の最小の演算処理ユニットでスケジューラなどを備えておらず，GPU は SM 内のスケジューラが発行した同じ命令を CUDA コアが並列に実行する SIMD(Single Instruction Multiple Data) 型のデータ処理を行う．カーネル関数と呼ばれる CUDA のプログラムは各 SM 上で 32 スレッドずつの Warp と呼ばれる単位で実行される．

GPU のメモリには Shared, Global, Local, Constant, Texuture, Register といった容量，アクセス遅延，アクセススコープの異なる様々なメモリ種類があり，パフォーマンスを引き出すために開発者がそれらの特徴を意識する必要がある．ここでは，Register, Shared Memory , および Global Memory について説明する．Register は，スレッド内のみからアクセス可能であり，アクセス遅延は小さい．Shared Memory は SM 内のスレッド間で共有可能であり，最大で 48KB と容量は小さいが，アクセス遅延は Register と同様小さい，Global Memory は SM をまたがる全てのスレッドから共有可能であり，容量は数 GB である．Global Memory のバンド幅は大きいがアクセス遅延が Register の 100 倍程度も大きい．Warp 内のスレッドが連続する Global Memmory 領域にアクセスする場合，これらのアクセスは結合 (Coalescing) され 1 回のメモリアクセスとなる．アクセス遅延の大きい Global Memory を利用する際にはこの結合アクセスの活用が非常に重要である．

ホストと GPU 間のデータのやり取りは Global Memory を介して行う．データの転送性能は PCI Express 2.0 x16 の転送レートに制限される．データ転送とカーネル関数の実行時間を合わせたトータルの処理性能の低下を防ぐために，ホストと GPU 間のデータ転送量と転送回数をできるだけ抑える必要がある．

GPU はストリーム処理が可能である．すなわち，ホストと GPU 間のデータ転送とカーネル関数の実行を同時に行うことができる．ストリーム処理を行うには，転送中のデータとカーネル関数の依存関係を開発者が明示的に示す必要があり，異なるストリームに属するデータ転送とカーネル関数の実行が同時並列的に行われる．

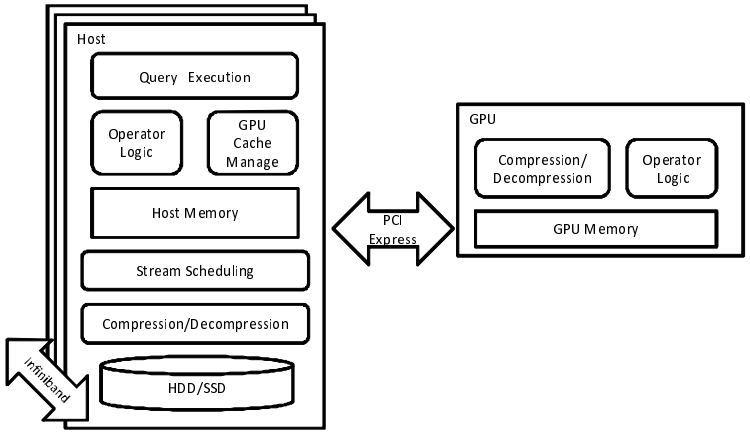

図 2 GPU 援用カラムストアデータベースアーキテクチャ

3. 提案アーキテクチャ

本システムの目的は，カラムストアデータに対するクエリを高速に処理することである．そのためのハードウェア，およびソフトウェアのアーキテクチャについて説明する．

図 2 に本システムのアーキテクチャを示す．なお，GPU を効率的に使うためのコンポーネントのみに注目し，通常のデータベースに含まれるメモリマネジメントやカタログ管理などの機能は省略する．

3.1 データ供給の観点から見たハードウェアの選択

データ供給の速度の観点からデータを保持するべきハードウェアついて考える．GPU を用いたデータベース処理では，ホストのメモリ，ディスク，あるいは別マシンに蓄えられたカラムデータを GPU に転送する必要があるため，GPU でどれだけ高速に処理を行っても全体の処理速度はデータ供給の速度律速になりやすい．なお，計算律速の処理については，より計算性能のよいハードウェアを選択するかアルゴリズムを工夫する他に余地はないため，ここでは議論しない．

表 1 に主なインターコネクトの規格とデバイスの転送レートの理論値を示す．GPU (GTX480) の内部バスの転送レートは 177.4 GB/s ととても高いが，GPU はホストと PCI Express 2.0 x16 を介して接続されているため，転送を含めた処理速度の上限は 8 GB/s に制限される．次世代の GPU では PCI Express 3.0 x16 が採用されるが，同様に 16GB/s に制限される．また，データの供給元として SATA などのディスクを使うとディスク IO が大きなボトルネックになり，CPU での処理に対して GPU での速度向上のメリットが失われてしまう．以上より，筆者らは GPU へのデータ供給に関して次の優先順で考える．

- GPU のメモリ上にカラムデータをキャッシュする．
- メインメモリの搭載量を増やし，そこからデータを供給する．
- PCI 接続の SSD からデータを供給する．
- Infiniband で接続された他マシンのメインメモリ上のデータを供給する．
- 複数の HDD や SSD から並列にデータを供給する．

上記の優先順でデータ供給を行うことにより，キャッシュを利用できる場合以外はできるだけ PCI Express をボトルネックとする．

表 1 転送レート (理論値, 1GB=10^9B)

Interface/Device	Throughput (GB/s)
GPU(GTX480)	177.4
PCI Express 2.0 x16	8
PCI Express 3.0 x16	16
QPI (3.2GHz)	12.8
SATA 300	0.3
SATA 600	0.6
Infini Band(QDR) x4	4
Infini Band(EDR) x4	12.6

```
1: bmp1 = table.filter(EQUAL, col1, 1);
2: bmp2 = table.filter(BEETWEEN, col2, 20, 29);
3: bmp3 = bmp1 & bmp2;
4: result  = table.sum(col3, bmp3)
```

図 3 ” SELECT SUM(col3) WHERE col1 = 1 AND col2 BETWEEN 20 AND 29” を処理する擬似コード

3.2 GPU を用いたクエリ処理

GPU を用いたクエリ処理について述べる．カラムストアでのクエリ処理はカラム単位の operator-at-a-time になるため，中間処理結果のデータを保存する必要がある．ソート，

ジョイン，スキャンといった演算の種類によって中間処理結果のデータ構造は異なるが，それらは処理後直ちに GPU からホストへ転送するのではなく，GPU 上で計算可能な一連の計算が終了するまで GPU 上に残す．

例えば，特定列に対する条件による絞り込みなどの述語処理はカラムのスキャン処理として実行するが，その処理結果のデータ構造としてビットマップを使う．ビットマップは，0/1 の真偽値からなる配列であり，AND/OR などの複合条件を GPU で処理するのにも非常に適している．**図 3** にビットマップを用いて，条件に合致する列の総和を求めるクエリを処理する疑似コードを示す．1 行目で条件節 ”col1 = 1” を満たすビットマップ bmp1 を GPU 上に作る．同様に 2 行目で条件節 ”col2 BETWEEN 20 AND 29” を満たすビットマップ bmp2 を GPU 上に作る．3 行目で bmp1 と bmp2 の AND を満たすビットマップ bmp3 を GPU 上に作る，4 行目で，bmp3 を用いてカラム col3 の値の総和を求める．この例では全ての処理を GPU 上で行うため，中間処理結果のビットマップをホストに転送する必要はない．

3.3 GPU メモリ上のキャッシュ管理

全てのデータを GPU 上に保持するのが最も良いのは明らかだが，現在の GPU のメモリ容量は 1.5-6GB 程度と限られているため現実的ではない．本システムでは．カラム単位で GPU 上にデータをキャッシュする．なお，13) で述べられているように処理結果自体をキャッシュし，次のクエリ処理に利用することは非常に有効であり，GPU を使う場合にも同様のことが言える．

3.4 ストリームスケジューラ

GPU 上のメモリに処理対象データがキャッシュされていない場合に，ストリーム処理によって転送時間，あるいは処理時間を隠蔽する．そのためにカラムデータを適切な大きさのチャンクに分割し，転送と処理をストリーム処理によって多重させる．

ストリーム処理を効率よく行う条件について考える．まず，チャンクサイズが小さい場合は，PCI Express の転送効率が落ちてしまうため，チャンクサイズにはある程度の大きさが必要である．次に，転送待ち (**図 4**-b)，処理待ち (図 4-c) が発生しないよう，チャンクの転送と処理に要する時間がバランスしている (図 4-d) とよい．データの転送時間は，チャンクサイズに対して単調増加するが，圧縮によって短縮することができる．圧縮データの展開も含めた処理時間はアルゴリズムにより変わるが，チャンクサイズに対して線形以上に増加する場合もあるため，処理に応じて好適なチャンクサイズがあると考えられる．そのため処理に応じてチャンクサイズを変化させることが有効であると考えられる．

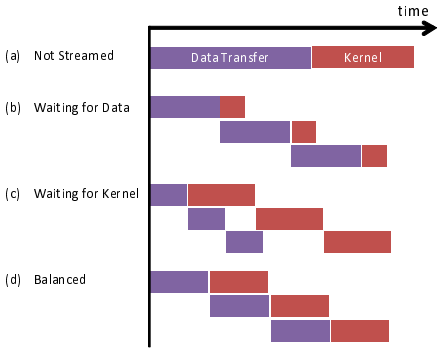

図 4 ストリーミング処理

上記の議論では処理結果の転送時間が考慮されていないが，チャンクの部分処理結果をストリーム転送する場合は，次のチャンクの転送時間に含めることができる．しかし，スキャンなど部分データに対しての処理のみで結果が得られる処理の他に，ソートなど全てのデータが揃わなければ結果が得られない処理がある．後者の場合は，転送と同時並列的に行う部分処理と，すべてのデータが揃って行われる全体処理を組み合わせて問題を解決するよう，アルゴリズムに工夫が必要である．

3.5 データ圧縮

データの転送時間が支配的な処理では，GPU 上でのデータの展開時間のコストを支払ったとしても，データ圧縮によって転送データ量を減らすことで全体の処理時間を減らすことができる．展開を行わず処理が可能な圧縮アルゴリズムも多数存在する．また，データ圧縮によってキャッシュにのるデータ量を増やすことができるため，キャッシュヒット率の向上も期待できる．ディスクとメインメモリ間の転送データの圧縮[14] と共に，メインメモリと GPU 間のデータ圧縮[11] も有効である．

4. 実　　装

本稿では，提案アーキテクチャを用いてスキャン処理を実装した．**表 2** にハードウェアの構成を示す．

4.1 カーネル関数

スキャン処理は，カラムデータである 4 byte の固定長数値型のコード値の配列を入力と

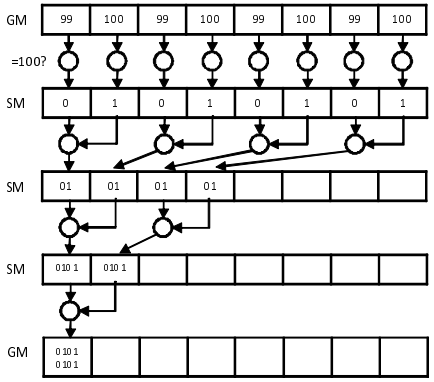

図 5 GPU によるスキャン処理

して，条件を満足たす箇所のビットを 1 にしたビットマップを出力する．

図 5 にスキャン処理の様子を示す．スキャン処理の実装は，Parallel Reduction[15] をベースに行った．各スレッドは Global Memory からコード 1 つ分のデータを取り出し，スキャン対象のデータと比較し，結果を Shared Memory に記録する．次に Shared Memory 上でリダクションを行い，最後に Global Memory に書きだす．

4.2 圧縮アルゴリズム

圧縮アルゴリズムはシンプルなバイト単位差分符号化を実装した．バイト単位差分符号化では，まずチャンク中のコードの最小値 min と最大値 max を探し，$max - min$ を表現可能な最小のバイトサイズ n を求める．次にチャンク中の各コード値 x を $x - min$ として，それぞれ n バイトで表現する．圧縮されたチャンクデータの他に，min と n をメタ情報として管理する．本実装では未圧縮のコードは 4 byte のため，n は 1, 2, 3, 4 のいずれかの値となる．

バイト単位差分符号化による圧縮データは，複雑な展開処理をする必要がない，また GPU の各スレッドがアクセスするデータが連続で結合アクセスが行えるため，GPU での演算に適している．圧縮効率については，同じ値がバースト的に発生するデータや，カーディナリティの低いカラムデータに対しては効率がよいと考えられる．

4.3 GPU メモリ上のキャッシュ管理

GPU のメモリ上に，一度処理に用いたカラムデータをカラム単位でキャッシュするよう

表 2 Hardware configuration

	CPU	GPU
Model	Intel Xeon (E5502)	GTX480
Cores	1.86GHz x 2 (logically4)	1401MHz x 480
Memory	16GB	1536MB
Memory Clock	400MHz	1848 MHz

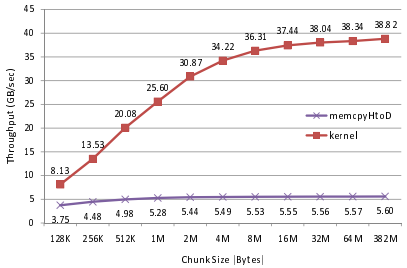

図 6 チャンクサイズと転送スループット，処理スループットの関係

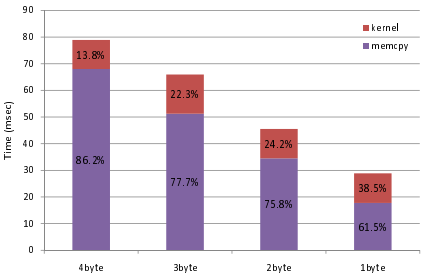

図 7 チャンクサイズと転送，処理の時間比

LRU (Least Recently Used) のアルゴリズムを実装した．

4.4 チャンクサイズの決定

図 6 に 1 億件分のデータに対するスキャン処理におけるチャンクサイズと転送，処理それぞれのスループットの関係を示す．全体のデータサイズは 382MB であり，桁計算は 2^{10} を基にしている．95%の効率性を目安とするとデータ転送は 1MB 以上, データ処理は 8MB 以上 のチャンクサイズが望ましいことがわかる．4.2 に示した圧縮アルゴリズムを用いると，チャンクサイズは最小で 1/4 倍になる．その際にも転送効率が 95%を達成できるよう，チャンクサイズを 4M として設定する．**図 7** にチャンクサイズが 4MB の場合に, 1-4 バイトでチャンクを圧縮した場合の転送時間と処理時間の比率を示す．どの場合においても処理の比率は転送に対して小さく，ストリーム処理時に処理待ちは発生しない．すなわち，スキャン処理においては，チャンクサイズ を 4MB とした場合に，圧縮時の処理のスループットは最適とはならないが，そのことが全体の処理時間に影響を及ぼさないと言える．また，チャンクの圧縮率が上がると処理時間が転送時間に近づいていくため，ストリーム処理による効果が高くなると予想される．

スキャン処理においては，どのチャンクサイズにおいても転送時間が支配的である．スト

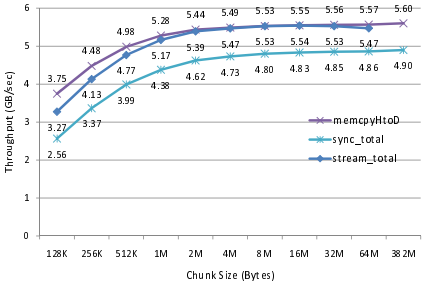

図 8 チャンクサイズと転送スループット，トータル処理スループットの関係

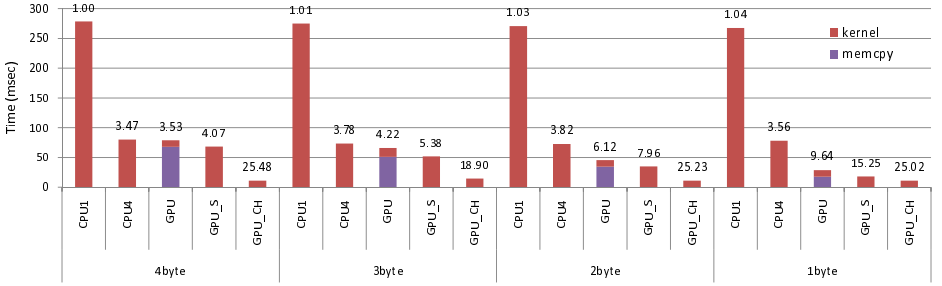

図 9 GPU vs CPU, Execution time and sppedup

リームによる転送と処理の重なりを考慮すると GPU での処理時間にはまだ余裕があるため．データの展開時間により GPU での処理時間が伸びるとしても，より複雑な圧縮アルゴリズムを用いてデータサイズを減らせば転送時間が減少し，全体の処理時間を短縮できると考えられる．

5. 評　　価

図 8 に未圧縮時の 1 億件のデータにおけるスキャン処理のスループットを示す．グラフはそれぞれ，ホストと GPU 間の転送スループット (memcpyHtoD)，ストリームを用いない GPU 処理のトータルスループット (sync_total)，ストリームを用いた GPU 処理のトータルスループット (stream_total) を示す．図よりチャンクサイズを適切に設定し，ストリーム処理をすることでトータルのスループットを転送スループットに近づけられていることが分かる．すなわち，スキャン処理はデータ転送のスループットで実行可能であると言える．

図 9 に 1 億件のデータにおけるスキャン処理の時間を示す．それぞれのグラフは全てのコードが 4，3，2，1 byte に圧縮された場合における CPU 1 スレッドでの処理 (CPU1), CPU 4 スレッドでの処理 (CPU4)，ストリームを用いない GPU での処理 (GPU)，ストリームを用いた GPU での処理 (GPU_S)，キャッシュヒットした場合の GPU での処理 (GPU_CH) の時間を示している．また，グラフ上の数値は，4 byte, CPU 1 スレッドの値を 1.00 とした場合のスピードアップを示す．スキャン処理はデータ転送時間が支配的なため，GPU での処理は圧縮率が上がるほど高速化していることが分かる．ストリームを用いない場合で最大 9．64 倍の高速化，ストリームを用いる場合で最大 15.25 倍の高速化が確認できた．また，圧縮率が上がるほど転送時間と処理時間のバランスがよくなるため，ストリームを用いた場合の効果が高くなっていることが確認できる．

一方，圧縮率が低い場合には，CPU 4 スレッドと比べて GPU でのスピードアップは小さく，メニーコアの CPU でさらにスレッドを増やしたり，ベクトル命令を用いることで，GPU を上回る性能を得られると考えられる．これは，CPU でのスキャン処理レートが PCI Express 2.0 x16 の性能を上回ることを意味する．しかし，GPU でキャッシュヒットした場合には，CPU 1 スレッドに対して 25 倍程度の性能が得られており，そのような CPU 実装を上回る性能を得られていると考えられる．さらに，次世代の GPU で PCI Express 3.0 x16 になることで，転送を含む GPU のスキャン性能は 2 倍近く向上すると考えられ，GPU は優位であると言える．

6. ま と め

本稿では，GPU をアクセラレータとして用いるカラムストアデータベースのアーキテクチャを提案した．提案アーキテクチャでは，データ転送のオーバヘッドの影響を少なくするために，カラムデータのキャッシュ，圧縮を行い，ストリーム処理に適したチャンクサイズを用いる．

評価の結果，適切にチャンクサイズを選ぶことで，スキャン処理の性能を PCI Express の転送レートとほぼ同じにできることを示し，CPU シングルスレッド実装に比べ 3-15 倍の性能向上を達成できることを示した．また，カラムデータのキャッシュを利用することで，CPU1 スレッドに比べ 25 倍の性能向上を達成できることを示した．

今後は，ソートやジョインなどの処理においてもチャンク化とストリーム処理による効果を確認していきたい．また，データ供給源として Infiniband で接続した他ノードのメイン

メモリなど他のハードウェアも使用した場合の性能への影響についても評価を行う予定である．

謝辞 本研究の一部は，独立行政法人新エネルギー・産業技術総合開発機構 (NEDO) の委託事業による成果である．

参 考 文 献

1) Copeland, G. and Khoshafian, S.: A decomposition storage model, *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, Vol.14, No.4, New York, NY, USA, ACM, pp.268–279 (1985).
2) SybaseIQ: `http://www.sybase.com/products/datawarehousing/sybaseiq`.
3) Ślezak, D. and Eastwood, V.: Data warehouse technology by infobright, *Proceedings of the 35th SIGMOD international conference on Management of data*, New York, NY, USA, ACM, pp.841–846 (2009).
4) Stonebraker, Mike and Abadi, Daniel J. and Batkin, Adam and Chen, Xuedong and Cherniack, Mitch and Ferreira, Miguel and Lau, Edmond and Lin, A., Madden and Sam and O'Neil, Elizabeth and O'Neil, Pat and Rasin, Alex and Tran, Nga and Zdonik, S.: C-store: a column-oriented DBMS, *Proceedings of the 31st international conference on Very large data bases*, VLDB Endowment, pp.553–564 (2005).
5) Boncz, P., Zukowski, M. and Nes, N.: MonetDB/X100: Hyper-pipelining query execution, *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, Vol.5, Citeseer (2005).
6) Manegold, S., Boncz, P. and Kersten, M.: Generic database cost models for hierarchical memory systems, *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB Endowment, pp.191–202 (2002).
7) Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M. and Manocha, D.: Fast computation of database operations using graphics processors, *Proceedings of the 2004 international conference on Management of data*, New York, New York, USA, ACM Press, p.215 (2004).
8) He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q. and Sander, P.: Relational joins on graphics processors, *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ACM, pp.511–524 (2008).
9) He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q. and Sander, P.V.: Relational query coprocessing on graphics processors, *ACM Transactions on Database Systems*, Vol.34, No.4, pp.1–39 (2009).
10) Bakkum, P. and Skadron, K.: Accelerating SQL database operations on a GPU with CUDA, *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units - GPGPU '10*, p.94 (2010).
11) Fang, W., He, B. and Luo, Q.: Database compression on graphics processors, *Proceedings of the VLDB Endowment*, Vol.3, No.1-2, VLDB Endowment, pp.670–680 (2010).
12) Wu, R., Zhang, B., Hsu, M. and Chen, Q.: GPU-accelerated predicate evaluation on column store, *Web-Age Information Management*, pp.570–581 (2010).
13) Ivanova, M., Kersten, M., Nes, N. and Gonçalves, R.: An architecture for recycling intermediates in a column-store, *ACM Transactions on Database Systems (TODS)*, Vol.35, No.4, p.24 (2010).
14) Zukowski, M., Heman, S., Nes, N. and Boncz, P.: Super-scalar RAM-CPU cache compression, *Proceedings of the 22nd International Conference on Data Engineering*, Washington, DC, USA, IEEE Computer Society, pp.59–59 (2006).
15) Harris, M.: Optimizing Parallel Reduction in CUDA, `http://www.nvidia.com/object/cuda_sample_data-parallel.html`.