Vol.2023-DBS-177 No.24
Vol.2023-IFAT-152 No.24
2023/9/22

CPUとGPUを併用するIOインテンシブなデータ処理の実行方式の検討

三浦 優也[1,a)]　小沢 健史[1,b)]　合田 和生[1,c)]

概要：GPU メモリとストレージとの間の直接 IO を可能にする GPUDirect Storage の登場により，IO インテンシブなデータ管理システムに対して，CPU と GPU を併用する新たな設計が見込まれる．両方のプロセッサを十分に活用する手立てとして，本論文では，その構成要素となる GPU 直接 IO について，著者らがこれまでに取り上げてきた同期 IO・バッチ IO に加え，非同期 IO の基本性能を測定し，結果を示す．

1. はじめに

GPU は CPU に比べて，より小型で定形処理に特化したコアを多数搭載しており，高い並列性能を誇る．そのため，単純な計算をいくつも行うようなタスクにおいて，CPU よりも格段に高い計算速度と帯域幅を発揮する．一方で，ストレージ上にある大規模なデータを扱う際には，従来，CPU メモリにデータを読み込み，GPU メモリに転送するという二段階のデータの移動を伴っていた．特に，GPU に搭載されているメモリの大きさは CPU に比べて小さく，大規模なデータをメモリのみで扱うことは難しい．

NVIDIA が 2021 年に提供を開始した GPUDirect Storage [1] は，ストレージと GPU メモリとの間のデータの直接のやり取りを可能にする技術である．GPUDirect Storage の機能は CUDA API [2] の一部である cuFile API として提供されており，追加のハードウェアなしに動作させることができる．これを用いることで，コストのかかる CPU–GPU 間のメモリ転送を行うことなく，GPU 直接 IO が可能となる．

CPU と GPU を併用するヘテロジニアスなデータベースシステムは今までにも検討されてきたが，データがメモリ上に収まる想定であるか，データがストレージ上にあるとすれば，CPU で IO を行って GPU で演算処理を行い，データ転送やロードバランシングを工夫する設計が基本であった．GPU 直接 IO の登場によって，CPU と GPU の両方のプロセッサを活用するための選択肢が増え，ストレージ上の大規模なデータを更に効率的に処理するための新たな設計を見込むことができる．

本論文では，CPU と GPU を併用してストレージ上のデータを処理するための最適な機構を検討する．そのために，GPU 直接 IO をはじめとする機構の各構成要素をまとめ，定量的な性能を測定する．また，構成要素を組み合わせて機構を試験的に作成し，TPC-H 問合せの実行性能を評価する．

本論文の構成は以下の通りである．第 2 章では，CPU–GPU 併用データベースエンジンの構成要素をまとめ，それらの基本性能を測定し，結果を示す．第 3 章では，CPU–GPU 併用データベースエンジンの設計を検討し，その実装による TPC-H 問合せ実験の結果を示す．第 4 章では，GPU を用いたデータベースシステムの研究を紹介し，第 5 章において本論文をまとめる．

2. CPU–GPU 併用エンジンの構成要素の基本性能実験

2.1 構成要素の整理

CPU と GPU を併用して IO インテンシブなタスクを処理する機構を作成するにあたって，まずはその構成要素の基本的な性能を知ることが，機構全体の最適な設計の手がかりとなる．本論文では，その構成要素を**図 1** のとおりに，CPU IO，GPU IO，転送，CPU 計算，GPU 計算の 5 つに定める．

(1) CPU IO
  (a) **同期 IO**．IO を発行した後，その IO が完了するのを待つ．Linux の `pread` を用いる．
  (b) **非同期 IO**．複数の IO を発行する命令と，複数の IO を待機する命令からなる．`libaio` の

[1] 東京大学 生産技術研究所
Institute of Industrial Science, The University of Tokyo
[a)] ymiura@tkl.iis.u-tokyo.ac.jp
[b)] ozawa@tkl.iis.u-tokyo.ac.jp
[c)] kgoda@tkl.iis.u-tokyo.ac.jp

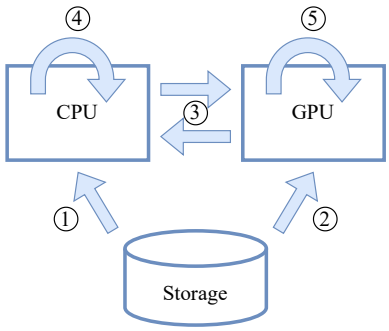
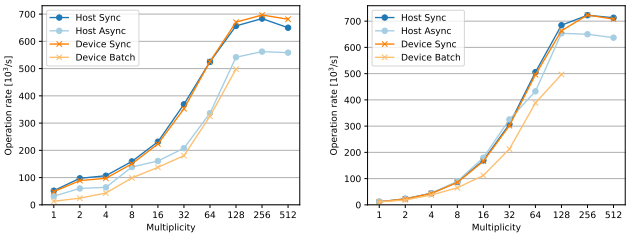


図 1: CPU–GPU 併用エンジンの構成要素

ワークステーション	HP Z4 G4 Workstation
CPU	Intel Xeon W-2245 (8 コア 16 スレッド @ 3.90 GHz)
メモリ	64 GiB
NVMe SSD	WD_BLACK AN1500
GPU	NVIDIA RTX A4000
ソフトウェア	Ubuntu 22.04 Mellanox OFED 5.8 CUDA 12.2.0

表 1: 実験環境

Table 1 Experimental environment



(a) シーケンシャルアクセス　(b) ランダムアクセス

図 2: CPU IO と GPU IO のスループット

`io_submit` と `io_getevents` を用いる．

(2) GPU IO
- (a) **同期 IO**．CPU で GPU IO を発行した後，その IO が完了するのを待つ．cuFile API の `cuFileRead` を用いる．
- (b) **バッチ IO**．CPU で複数の GPU IO を発行する命令と，CPU で複数の GPU IO を待機する命令からなる．cuFile Batch API の `cuFileBatchIOSubmit` と `cuFileBatchIOStatus` を用いる．
- (c) **非同期 IO**．GPU IO を CUDA ストリームに登録する．cuFile Stream API の `cuFileReadAsync` を用いる．

(3) CPU–GPU 間メモリ転送
- (a) **同期転送**．CPU と GPU 間のメモリ転送を発行した後，その転送が完了するのを待つ．CUDA API の `cuMemcpyHtoD` と `cuMemcpyDtoH` を用いる．
- (b) **非同期転送**．CPU と GPU 間のメモリ転送を CUDA ストリームに登録する．CUDA API の `cuMemcpyHtoDAsync` と `cuMemcpyDtoHAsync` を用いる．

(4) **CPU 計算**．CPU の関数を呼び出し，完了するのを待つ．

(5) **GPU 計算**．GPU カーネルの呼び出しを CUDA ストリームに登録する．CUDA API の `cuLaunchKernel` を用いる．

本実験では，各構成要素について，4 KiB のバッファに対する操作を 4 GiB 分行い，その実行時間と，時間あたりの操作回数を測定した．

2.2 実験環境

上述の実験を**表 1** に示す環境で行った．

2.3 CPU IO と GPU IO の性能

多重度を m としたとき，CPU IO と GPU IO を以下のようにして多重化し，その性能を測定した．

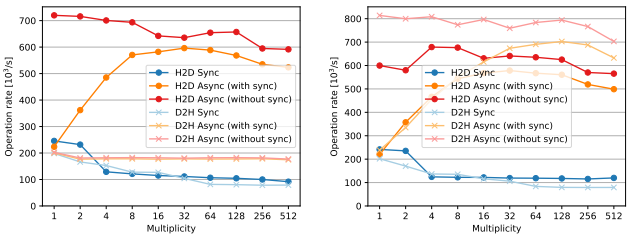
(1) CPU 同期 IO と GPU 同期 IO は，m 個のスレッドを立ち上げ，各スレッドで IO を発行し続ける．

(2) CPU 非同期 IO と GPU バッチ IO は，ひとつのスレッドから，常に m 個の IO が発行中になるように発行命令と待機命令を繰り返す．

(3) GPU 非同期 IO の同期あり方式は，m 個のストリームを作り，各ストリームに対して IO を発行し，`cuStreamSynchronize` を用いて結果を同期することを繰り返す．

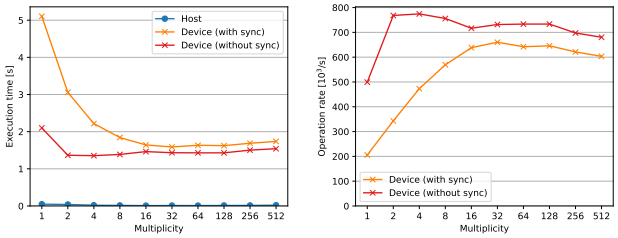
(4) GPU 非同期 IO のコールバック方式は，m 個のストリームを作り，IO が完了しているストリームに対して IO を発行し，直後に `cuLaunchHostFunc` を用いて IO の完了通知を送るようにすることを繰り返す．

結果を**図 2** に示す．インターフェイスとして対応するのは CPU 同期 IO と GPU 同期 IO，CPU 非同期 IO と GPU バッチ IO である．**図 2(a)** を見ると，第一に，これらの対応する IO 方式について，CPU と GPU でほとんど同じ性能を発揮していることがわかる．第二に，CPU 同期 IO と GPU 同期 IO が CPU 非同期 IO と GPU バッチ IO よりも高い性能を発揮している．これはスケジューリングの問題で，`libaio` や cuFile Batch API を用いて多重度を常に m にするのが難しいためである．第三に，GPU 非同期 IO については，同期方式でもコールバック方式でも同様に，他のケースと比べて著しく低いスループットを確認した（図示しない）．これについて Nsight Systems で実行パイプラインを可視化したところ，`cuFileReadAsync` は内部で `cuLaunchHostFunc`，`cuFileGpuWaitValue32<<<1, 1>>>`，`cuLaunchHostFunc`，`cuCopyGpu<<<1024,`



(a) ページロックなし　(b) ページロックあり

図 3: CPU–GPU 間メモリ転送のスループット



(a) 実行時間　(b) スループット（CPU 計算は大きいため省略）

図 4: CPU 計算と GPU 計算の性能

1024>>>, cuLaunchHostFunc といった一連の手順を呼び出していた．すなわち，GPU 非同期 IO の多重化性能は，IO の多重度よりもこれらの内部手順の多重度によって律速されているものと考えられる．今後，更に調査を進める予定である．

2.4 CPU–GPU 間メモリ転送の性能

CPU から GPU への転送を host to device（H2D），GPU から CPU への転送を device to host（D2H）と呼ぶ．実験を行った環境においては，cudaDeviceProps::asyncEngineCount の値を確認したところ，GPU にコピーエンジンが 2 つ搭載されており，コピーエンジンの個数を超える転送をハードウェア的に同時並列で実行することはできない．よって，転送を発行したり待機したりする際のレイテンシを小さくすることが実行性能の鍵となる．本実験では，多重度を m としたとき，H2D，D2H を以下のようにして多重化し，その性能を測定した．

(1) H2D 同期転送と D2H 同期転送は，m 個のスレッドを立ち上げ，各スレッドで転送を発行し続ける．
(2) H2D 非同期転送と D2H 非同期転送の同期あり方式は，m 個のストリームを作り，各ストリームに対して転送を発行した直後に，cuStreamSynchronize を用いて同期することを繰り返す．
(3) H2D 非同期転送と D2H 非同期転送の同期なし方式は，m 個のストリームを作り，各ストリームに対して転送を発行し続け，最後に cuStreamSynchronize を用いて同期する．

ただし，CPU 側のメモリバッファを cuMemHostRegister を用いてページロック（ピン止めとも呼ばれる）しておくかどうかによって性能が変わるため，ページロックしない場合とする場合の両方を測定した．

結果を**図 3** に示す．**図 3(a)** と**図 3(b)** を見ると，第一に，スレッドを用いた同期転送の多重化では，スレッド数が増えるほど性能が下がっていることが分かる．これは CUDA のリソースの扱いについてスレッド競合が生じているためと考えられ，機構を実装する上でスレッドよりもストリームによる多重化を行ったほうがよいことが伺える．第二に，H2D はページロックの有無の影響をあまり受けていないのに対して，D2H はページロックの有無の影響を大きく受けていることが分かる．Nsight Systems を用いて実行パイプラインを可視化したところ，D2H ではページロックをしないと転送の呼び出しが同期的になっているように見受けられた．第三に，cuStreamSynchronize による同期のレイテンシは大きく，なるべく同期を呼び出さない（図の凡例 without sync）か，複数のストリームを用いる（図の凡例 with sync で x 軸を大きくする）ことで性能を向上させることができることが分かる．

2.5 CPU 計算と GPU 計算の性能

どのような計算を行うかはワークロードに依存するため，本実験では特に GPU カーネルの呼び出しのレイテンシや並列度に注目する．本実験では，与えられたバッファを TPC-H の LINEITEM の配列とみなし，各 extendedprice の総和を求める演算を CPU と GPU それぞれで行った．CPU では単純なループを用いて総和を求め，GPU ではバッファに含まれる LINEITEM の個数分のスレッドでカーネルを呼び出す並列化を行った．実験を行った環境においては，cudaDeviceProps::concurrentKernels の値を確認したところ，複数のカーネルの並行実行が可能であることは確認できたが，同時にいくつ実行可能であるかの詳細な数値は得られなかった．多重度を m としたとき，CPU 計算と GPU 計算を以下のようにして多重化し，その性能を測定した．

(1) CPU 計算は，m 個のスレッドを立ち上げ，各スレッドで関数を呼び出し続ける．
(2) GPU 計算の同期あり方式は，m 個のストリームを作り，各ストリームに対してカーネル実行を発行した直後に，cuStreamSynchronize を用いて同期することを繰り返す．
(3) H2D 非同期転送と D2H 非同期転送の同期なし方式は，m 個のストリームを作り，各ストリームに対してカーネル実行を発行し続け，最後に cuStreamSynchronize を用いて同期する．

結果を**図 4** に示す．**図 4(b)** を見ると，転送と同様に，

なるべく同期を呼び出さない（図の凡例 without sync）か，複数のストリームを用いる（図の凡例 with sync で x 軸を大きくする）ことで性能を向上させることができることが分かる．

3. CPU–GPU 併用エンジンの設計と TPC-H 問い合わせ実験

3.1 多重化機構とその組み合わせによるエンジンの設計

独立な複数のタスクを並行して行えるように多重化することを，「horizontal な分割」と呼ぶことにする．horizontal な分割の方法として以下が考えられる．

- **スレッドによる多重化**．すべての構成要素，あるいはその組み合わせがこの方法を取ることができる．
- **ストリームによる多重化**．ストリームを用いる構成要素である GPU 非同期 IO，GPU 計算，あるいはその組み合わせがこの方法を取ることができる．

また，horizontal な分割ではないが，IO のみのタスクに関して以下の多重化も可能である．

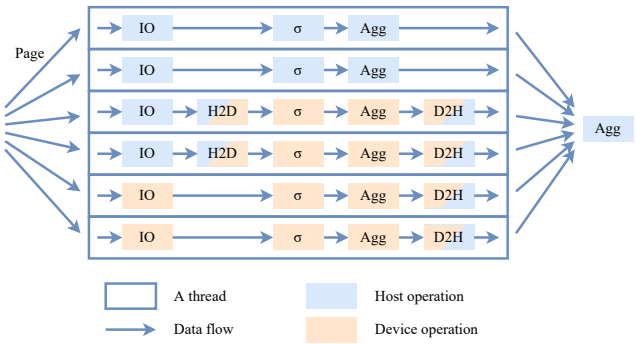
- **バッチによる多重化**．バッチを用いる構成要素である CPU 非同期 IO と GPU バッチ IO がこの方法を取ることができる．

さらに，依存関係にある前後のタスクを分割することを「vertical な分割」と呼ぶことにする．IO と計算のような異なる種類のタスクが依存関係にある場合，vertical な分割を行うことによって，分割後の各タスクに対してそれぞれの最適な多重度での多重化が可能となる．

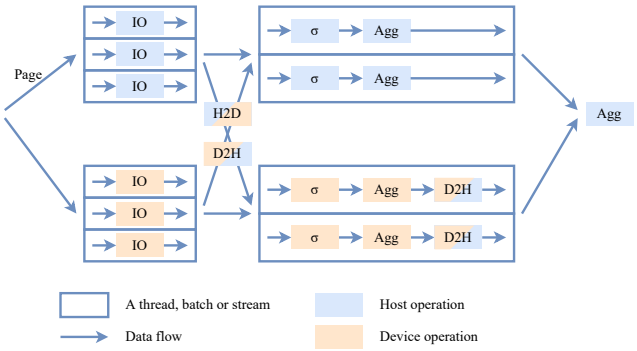
これらの多重化機構をもとに，第 2 章の構成要素を組み合わせて，以下の 2 つの CPU–GPU 併用エンジンを設計した．

(1) **horizontal エンジン**．**図 5(a)** に示す機構で，以下の 3 種類のタスクを horizontal に分割し，スレッドにより多重化する．
  (a) **host スレッド**．CPU で IO と計算を行う．
  (b) **device indirect スレッド**．CPU で IO を行い，GPU に転送し，GPU で計算を行う．
  (c) **device direct スレッド**．GPU で IO と計算を行う．

(2) **horizontal + vertical エンジン**．**図 5(b)** に示す機構で，CPU のタスクと GPU のタスクに horizontal に分割し，さらに IO のタスクと計算のタスクに vertical に分割する．本論文の執筆時点では，H2D による host read と device compute 間の受け渡し，D2H による device read と host compute 間の受け渡しは実装できていない．
  (a) **host read スレッド**．CPU で IO を行う．スレッドかバッチによる多重化が可能である．
  (b) **host compute スレッド**．CPU で計算を行う．スレッドによる多重化が可能である．



(a) **horizontal エンジンの構造**．上 2 段は host read + host compute (host)，中 2 段は host read + device compute (device indirect)，下 2 段は device read + device compute (device indirect) を表す．



(b) **horizontal + vertical エンジンの構造**．左上は host read，右上は host compute，左下は device read，右下は device compute を表す．

図 5: 各エンジンの構造

  (c) **device read スレッド**．GPU で IO を行う．スレッド，バッチ，ストリームによる多重化が可能である．
  (d) **device compute スレッド**．GPU で計算を行う．スレッドかストリームによる多重化が可能である．

代表として，スレッドによる多重化の詳細な手順を**アルゴリズム 1** に示す．スレッド間通信には Rust 言語の `std::sync::mpsc::sync_channel` を用いた．

これらのエンジンを用いて，TPC-H ベンチマーク [3] の Query 6 の実行性能を測定した．Query 6 の内容は**図 6** の通りである．データの生成には TPC-H の `dbgen` を用い，scale factor = 10 のサイズで生成した．Query 6 の実行には `LINEITEM` 表が必要であり，このとき生成された `LINEITEM` 表の行数は 59,986,052 行であった．ページサイズを 4 KiB としてストレージに格納し，ページ数は 2,399,443 ページとなった．また本実験は，表 1 に示したものと同じ環境下で行った．

3.2 horizontal エンジンの実行性能

horizontal エンジンで実行したときの結果を**図 7** に示

Vol.2023-DBS-177 No.24
Vol.2023-IFAT-152 No.24
2023/9/22

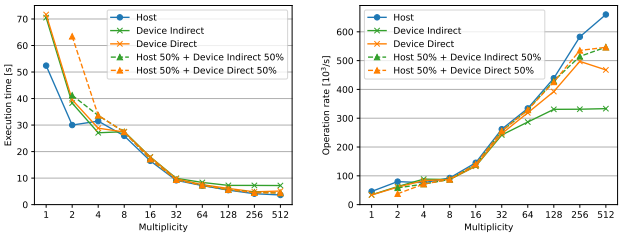
アルゴリズム 1: スレッドによる多重化

function `Main`(n, C_{in} (if vertical), C_{out} (if vertical))
- $C_{\text{done}} \leftarrow$ Create a channel;
- **for** $i = 0$ **to** $n - 1$ **do**
  - $C_i \leftarrow$ Create a channel;
  - $T_i \leftarrow$ Spawn a thread with `ThreadFn`(i, C_i, C_{done}, C_{out});
  - Send the index i to C_{done};
- **while** $I \leftarrow$ Receive an input from C_{in} (if vertical; otherwise generate a task) **do**
  - $i \leftarrow$ Receive an index from C_{done};
  - Send the input I to C_i;
- **for** $i = 0$ **to** $n - 1$ **do**
  - Join the thread T_i;

function `ThreadFn`(i, C_i, C_{done}, C_{out} (if vertical))
- Initialization;
- **while** $I \leftarrow$ Receive an input from C_i **do**
  - Process with the input I;
  - Send the index i to C_{done};
  - Send the output to C_{out} (if vertical);
- **return** gathered output

```
select
    sum(l_extendedprice * l_discount) as revenue
from
    lineitem
where
    l_shipdate >= date '1994-01-01'
    and l_shipdate < date '1994-01-01' + interval
        '1' year
    and l_discount between .06 - 0.01 and .06 +
        0.01
    and l_quantity < 24;
```

図 6: 実験に用いた Query 6

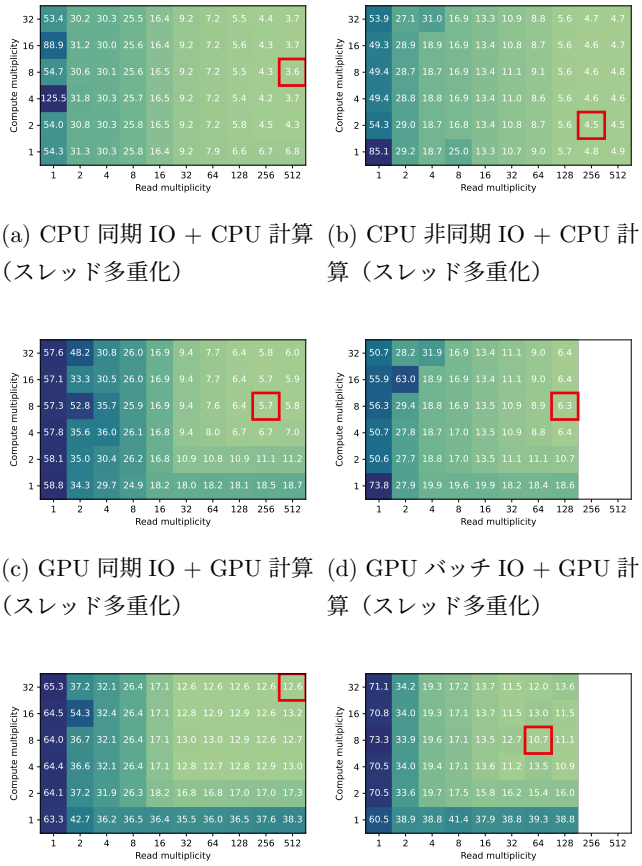


(a) 実行時間　(b) スループット

図 7: horizontal エンジンによる Query 6 の実行性能

す．**図 7(b)** を見ると，device indirect 実行は H2D 転送を伴う分，host 実行に比べて半分程度の性能となっている．一方で，device direct 実行は host 実行に多少劣るものの近い性能を発揮している．

また，host と device direct の混合実行は，host 実行と device direct 実行の間に位置する実行速度となっている．



(a) CPU 同期 IO + CPU 計算（スレッド多重化）　(b) CPU 非同期 IO + CPU 計算（スレッド多重化）

(c) GPU 同期 IO + GPU 計算（スレッド多重化）　(d) GPU バッチ IO + GPU 計算（スレッド多重化）

(e) GPU 同期 IO + GPU 計算（ストリーム多重化）　(f) GPU バッチ IO + GPU 計算（ストリーム多重化）

図 8: **horizontal + vertical エンジンによる Query 6 の実行性能**．横軸は IO の多重度，縦軸は計算の多重度を表す．色が濃いほど実行時間が大きい．最も実行時間が小さい多重度の組み合わせを赤い四角で囲んでいる．

host と device indirect の混合実行も同様である．本実験のワークロードでは計算に比べて IO が支配的であるため GPU を用いることによる恩恵はないものの，この結果は horizontal エンジンが CPU と GPU に適切にタスクを分配できていることを示している．

3.3 horizontal + vertical エンジンの実行性能

horizontal + vertical エンジンで実行したときの結果を**図 8** に示す．horizontal エンジンでは，IO と計算の多重度が一致する，図の対角線部分での実行しかできなかったのに対し，horizontal + vertical エンジンではそれぞれの多重度を自由に設定できるようになっている．全体を通して見ると，IO の多重度に関して，256 か 512 程度で飽和している．**図 8(a)** と**図 8(b)** を見ると，CPU による実行では，CPU IO の多重度が小さいときは IO が支配的であるため CPU 計算の多重度は無視できるが，CPU IO の多重度を大きくすると，CPU 計算を多重化するとよいことが分かる．CPU 計算の最適な多重度は，実験環境の CPU の

8 コア 16 スレッドという数値に依存していると言える．一方で，**図 8(c)**，**図 8(d)**，**図 8(e)**，**図 8(f)** を見ると，ストリームによる多重度は 8 程度で最適となっていることが分かる．また，ストリーム多重化は最適な場合でもスレッド多重化の 2 倍ほどの実行時間がかかっている．これについて Nsight Systems を用いて実行パイプラインを確認したところ，ストリーム多重化機構の `cuLaunchHostFunc` のレイテンシが `cuLaunchKernel` と同程度に影響しているように見受けられた．CUDA のこうした API は呼び出しのコストが大きく，なるべく呼び出し回数を減らすような設計が求まれる．

4. 関連研究

4.1 GPU を用いたデータベースシステム

Yogatama らが提案する Mordred [4] は，CPU と GPU 間のデータ配置と問合せ実行を工夫することにより，全体のトラフィックを最適化し，CPU と GPU の並列性を高めた．データ配置ポリシーとしては，GPU へのキャッシュを細かい粒度に分け，コストモデルに基づいた計算を行い，GPU にキャッシュされていてほしいデータを高い精度で GPU に配置した．異種問合せ実行としては，細かく分けたデータがどう配置されているかによって問合せ実行を分割し，CPU と GPU の両方を活用した．

他にも近年の研究では，処理配置 [5], [6]，問合せコンパイル [7], [8]，問合せ実行 [9], [10], [11] などに注目した手法がある．共通する傾向としては，CPU と GPU が協力するヘテロジニアスなデータベースシステムを意識した研究が多い．

4.2 GPU 直接 IO に関する研究

Li ら [12] は，NVMe SSD と GPU 間の直接 IO を独自に実装し，圧縮とデータ転送の最適化を用いる OLAP システムを構築した．Ravi ら [13] は，GPUDirect Storage を階層的ファイル形式 HDF5 のライブラリに適用した．Inupakutika ら [14] は，GPUDirect Storage の性能の初期段階的な定量化を行った．

5. おわりに

本論文では，CPU と GPU を併用してストレージ上のデータを処理するための最適な機構を検討した．そのために，GPU 直接 IO をはじめとする機構の各構成要素をまとめ，定量的な性能を測定した．また，構成要素を組み合わせて機構を試験的に作成し，TPC-H 問合せの実行性能を評価した．

今後の展望としては，horizontal+vertical エンジンに H2D と D2H を組み込み，CPU と GPU 間で自由にデータをやり取りできるように実装していきたい．また，Query 6 は問合せのごく簡単なケースであり，より複雑な問合せを動かせるように機構の設計を検討していきたい．

謝辞

本研究の一部は，日本学術振興会科学研究費補助金基盤研究 (B) JP20H04191 の助成を受けたものである．

参考文献

[1] NVIDIA GPUDirect Storage. `https://docs.nvidia.com/gpudirect-storage/index.html`. Accessed: 2022-12-03.

[2] CUDA C++ Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Accessed: 2022-10-25.

[3] TPC-H Homepage. `https://www.tpc.org/tpch/`. Accessed: 2022-12-04.

[4] Bobbi W Yogatama, Weiwei Gong, and Xiangyao Yu. Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS. *PVLDB*, Vol. 15, No. 11, pp. 2491–2503, 2022.

[5] Sebastian Breß, Henning Funke, and Jens Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *Proc. SIGMOD*, pp. 1891–1906, 2016.

[6] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *PVLDB*, Vol. 10, No. 7, pp. 733–744, 2017.

[7] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined Query Processing in Coprocessor Environments. In *Proc. SIGMOD*, pp. 1603–1618, 2018.

[8] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *PVLDB*, Vol. 12, No. 5, pp. 544–556, 2019.

[9] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proc. SIGMOD*, pp. 1413–1425, 2021.

[10] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *Proc. SIGMOD*, pp. 1017–1032, 2022.

[11] Bala Gurumurthy, David Broneske, Gabriel Campero Durand, Thilo Pionteck, and Gunter Saake. ADAMANT: A Query Executor with Plug-In Interfaces for Easy Co-processor Integration. In *ICDE 2023*.

[12] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *PVLDB*, Vol. 9, No. 14, pp. 1647–1658, 2016.

[13] John Ravi, Suren Byna, and Quincey Koziol. GPU Direct I/O with HDF5. In *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, pp. 28–33, 2020.

[14] Devasena Inupakutika, Bridget Davis, Qirui Yang, Daniel Kim, and David Akopian. Quantifying Performance Gains of GPUDirect Storage. In *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–9, 2022.