Vol.2018-OS-144 No.16
2018/7/31

GPUミドルウェア”Victream”のシミュレーションによる性能評価

鈴木 順[1,2,a)]　林 佑樹[1]　荒木 拓也[1]　喜連川 優[2]

概要：Victream は著者らが [1] で提案した複数の GPU を用いた Out-of-Core 処理向けミドルウェアである．Out-of-Core 処理では GPU(Graphics Processing Unit) のメモリ容量より大きなデータを分割し，GPU とホストのメインメモリの間で入れ替えながら処理を行う．そのため GPU のデータ I/O が性能ボトルネックとなる．このボトルネックの緩和のため，Victream は GPU へのデータのプリフェッチと，GPU へのデータ I/O 量最小化を同時に実現する．そのため Victream スケジューラは動的でヒューリスティックなスケジューリング手法を採用している．しかし，ヒューリスティック手法は必ずしも最良のスケジュールを保証しない．そこで本稿では，与えられた計算に対して可能なスケジュールを Brute-Force シミュレーションで全探索し，得られた最良のスケジュールと Victream のスケジュールの計算時間を比較することで Victream の性能を評価する．結果，Victream は最良のスケジュールが実現する最短の計算時間ではないが，最短の計算時間に対する増加を 10%以内に抑えた計算性能が実現できることがわかった．本稿ではさらに，Victream をホストと GPU を分離して計算プラットフォームの高い運用効率を実現する I/O デバイス分離のアーキテクチャに適用し，Victream の有効性を検証した．I/O デバイス分離のアーキテクチャでも性能ボトルネックはホストと GPU の間のデータ I/O となる．評価の結果，Victream は I/O デバイス分離のアーキテクチャで従来手法より効率的に GPU の演算リソースを使用でき，用いる GPU 数を増加させた場合も高い計算性能の向上が得られることがわかった．評価では，Victream は従来手法に対し最大 296%の性能向上を実現した．

Simulation Evaluation of Performance of GPU Middleware ”Victream”

Jun Suzuki[1,2,a)]　Yuki Hayashi[1]　Takuya Araki[1]　Masaru Kitsuregawa[2]

1. はじめに

GPU(Graphics Processing Unit) は CPU(Central Processing Unit) の計算をオフロードし，高い計算性能を実現するアクセラレータとして注目されている．データ処理フレームワークの Spark [2] やディープラーニングフレームワークの Tensorflow [3] でも GPU が用いられている．

GPU を用いた計算で処理するデータの量はデータ解析や AI(Artificial Intelligence) の需要の高まりにより増大している．しかし，GPU のメモリ容量は 10GB 程度と今日データセンターで用いられているホストのメインメモリより 2 桁小さい．従って GPU を用いた大容量のデータ処理では大きなデータを分割し GPU とホストのメインメモリの間でデータを入れ替えながら計算を行う Out-of-Core 処理となる．しかし，データを GPU とホストのメインメモリで入れ替えるデータスワップで用いる I/O バスの帯域は 16GB/s 程度と GPU のローカルメモリの帯域より 1 桁小さい．このため GPU の Out-of-Core 処理では GPU のデータ I/O が性能ボトルネックとなる．

このデータ I/O のボトルネックの課題を解決するため，著者らは [1] で Victream と呼ぶミドルウェアを提案した．

[1] NEC データサイエンス研究所
1753, Shimonumabe, Nakahara-Ku, Kawasaki, Kanagawa 211–8666, Japan
[2] 東京大学生産技術研究所
Institute of Industrial Science, the University of Tokyo
4-6-1, Komaba, Meguro-Ku, Tokyo 153-8505, Japan
[a)] j-suzuki@ax.jp.nec.com

Victream は Dryad [4] や Spark [5] のようなデータパラレル型アプリケーションのための DAG(Directed Acyclic Graph) 型ミドルウェアである．DAG はアプリケーションプログラムの処理のデータフローを示す．Victream ではアプリケーションプログラムで DAG を作成しランタイムに渡すと，ランタイムが DAG の頂点のタスクを実行する．このとき複数の GPU を用いた Out-of-Core 処理においてデータ I/O 量が最小となるよう DAG が含むタスクの実行のスケジューリングを行う．

Victream のスケジューラの特徴は，Out-of-Core 処理におけるデータ I/O 量の最小化と，GPU へのデータプリフェッチを同時に実現することである．GPU へのデータプリフェッチでは，GPU が演算を行っている間に将来のタスクの入力データをホストのメインメモリから GPU のメモリにロードする．このデータプリフェッチは Out-of-Core 処理においてもボトルネックリソースである I/O バスを常に稼働させリソースを効率的に利用するために重要である．Victream のスケジューラでは，GPU のデータ I/O を最小化するためにローカリティアウェアスケジューリングを用いる．このスケジューリングでは，GPU に実行させるタスクとして GPU が保持するデータを可能な限り再利用し，発生させるデータスワップが最小のタスクを選択する．Victream ではさらに，GPU のデータプリフェッチを同時に行うため，上記のローカリティアウェアスケジューリングを拡張する．プリフェッチは将来のタスクの入力データをロードすることであるため，スケジュールも GPU が空いてからではなく，GPU が現在のタスクの演算を実行中に将来実行するタスクのスケジューリングを行う必要がある．そこで Victream のローカリティアウェアスケジューリングでは，将来実行するタスクをスケジューリングするように拡張を行っている．

しかし，Victream のスケジューラは動的でヒューリスティックなスケジューリングを用いており，必ずしも最短の計算時間を実現する最良のスケジューリングを保証しない．ただし，Victream が想定しているアプリケーションでは，DAG の頂点のユーザ定義関数の実行時間を予めミドルウェアが知ることができず，最良のスケジューリングを計算できない．またもし同じタスクを繰返し計算するような限定された応用例でタスクの実行時間がわかった場合でも，最適なタスクの実行順を全探索で探索すると計算量が組み合わせ爆発で増大し，応用を想定する規模のスケジューラを実現することができない．しかし，これらの課題を解決する Victream のスケジューラが，実際は最良のスケジュールに対しどの程度の性能を実現しているかを知ることは重要である．

そこで本稿では，計算シミュレーションを用いて与えられた DAG が取り得るスケジュールを Brute-Force で全探索し，計算時間が最も短いスケジュールを選択することで最良のスケジュールを計算する．そして，その結果と Victream が選択するスケジュールを比較することで Victream の性能を評価する．評価の結果，Victream は最良のスケジュールは実現しないが，最良のスケジュールの計算時間に対し，最大でも 10%以内に計算時間の増加を抑えた性能が実現できることがわかった．

さらに本稿では，著者らが [6] で提案した ExpEther を用いてホストと GPU を分離した計算プラットフォームに Victream を適用した場合の評価結果を述べる．ホストと GPU を分離したプラットフォームでは，ホストに必要に応じて所望の数の GPU を割り当てることができ，計算リソースの高い運用効率を実現できる．しかし，ホストと GPU をイーサネットで接続するため，Out-of-Core 処理時にはイーサネットを用いたデータスワップが性能ボトルネックとなる．ここに Victream を適用し Victream の有効性の評価を行う．その結果，Victream は従来手法より GPU の演算リソースを効率的に使用することができ，用いる GPU 数を増加させた場合も計算性能を効率的に向上させられることがわかった．

以下本論では第 2 節で Victream の概要，第 3 節でシミュレーションによる性能評価，第 4 節で I/O デバイス分離のアーキテクチャへの Victream の適用を述べ，最後に第 5 節でまとめる．

2. Victream

2.1 アーキテクチャ

始めに Victream のアーキテクチャについて述べる．アーキテクチャの構成を図 1 に示す．Victream はユーザライブラリとランタイムから構成される．ユーザライブラリはアプリケーションに組み込まれて使用され，ユーザプログラムの処理を示す DAG を生成する．ランタイムはユーザライブラリが生成した DAG を受け取り，ホストが保持する複数の GPU を用いて DAG の処理を実行する．このとき，Out-of-Core 処理で GPU のデータ I/O が最小となるようにタスクのスケジューリングを行う．

ユーザライブラリはユーザプログラムからの API (Appication Programming Interface) の呼び出しにより内部で DAG を生成する．Victream は Spark と同様，map や reduce のデータパラレル型の API を提供する．API が呼び出されユーザ定義関数が引数として渡されると，ライブラリの内部で生成する DAG に新たな頂点のタスクが追加される．そして DAG の実行を開始する API が呼ばれると，それまでに生成した DAG の実行が RPC(Remote Procedure Call) でランタイムに依頼される．

ランタイムでは DAG アナライザがライブラリから受信した DAG を解析し，DAG の頂点のタスクをサブタスクに分割する．サブタスクは GPU でのタスクの実行単位である．Victream ではデータオブジェクトを分割してデー

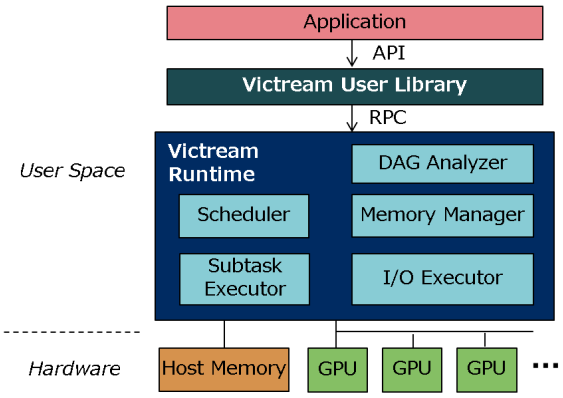


図 1　Victream のアーキテクチャ

タパーティションの単位で管理しており，データパーティションに対して実行するタスクの単位がサブタスクである．DAG アナライザは生成したサブタスクの実行をスケジューラに依頼する．スケジューラはサブタスクの実行順序をスケジューリングする．スケジューラはまず，サブタスクを実行させる GPU にサブタスクの入出力データのメモリ領域を確保するようメモリマネージャに依頼する．そして GPU に入力データパーティションに存在せず，GPU へのロードが必要であれば I/O エクセキュータに依頼する．また入力データパーティションの準備と出力データパーティションのメモリ領域の確保が完了した後は，サブタスクの実行をサブタスクエクセキュータに依頼する．

2.2　スケジューラ

次に Victream のスケジューラについて述べる．スケジューラは複数の GPU を用いた Out-of-Core 処理で計算の性能ボトルネックとなる GPU のデータ I/O 量を最小化する．Victream が従来手法と異なる点は，ボトルネックリソースの I/O バスを常に稼働させるデータプリフェッチをデータ I/O 量の最小化と同時に実現することである．

Victream のスケジューラは Out-of-Core 処理で GPU のデータ I/O 量を最小化するためローカリティアウェアスケジューリングを用いる．すなわち，スケジュールするサブタスクを選択する際に，GPU メモリが保持しているデータを入力データとして可能な限り再利用し，発生させるデータスワップの I/O 量が最小のサブタスクを候補の中から選択する．

さらに，Victream のスケジューラでは GPU のデータプリフェッチを行うため，上記のローカリティアウェアスケジューリングを拡張する．GPU のデータプリフェッチは将来実行するサブタスクの入力データをホストのメインメモから GPU メモリにロードすることである．そのため，Victream のスケジューラは GPU の演算リソースが空いてからではなく，GPU が現在のサブタスクの演算を実行中に，将来実行するサブタスクのスケジューリングが行えるようローカリティアウェアスケジューリングを拡張している．

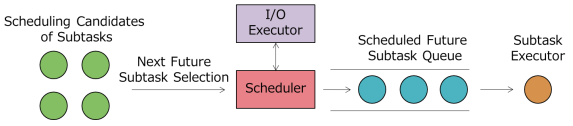


図 2　データプリフェッチを行うためのデータアウェアスケジューリングの拡張

スケジューラの構成を図 2 に示す．Victream のスケジューラはスケジュール候補のサブタスクの中から将来実行するサブタスクを選択する．選択するサブタスクの基準は発生させるデータ I/O 量が最も小さいサブタスクである．そして選択したサブタスクの入力データがホストのメインメモリにスワップアウトされていた場合，I/O エクセキュータにデータを GPU にロードするよう依頼する．GPU への入力データのロードが完了し，出力メモリ領域も確保したサブタスクはサブタスクキューにキューイングされる．キューイングされたサブタスクは FCFS(First-Come First-Serve) でサブタスクエクセキュータによって演算が実行される．一方スケジューラはサブタスクをキューイングすると直ちに次のサブタスクのスケジュールに移る．このような構成により，GPU ではサブタスクエクセキュータが実行している演算と非同期に将来実行するサブタスクの入力データのプリフェッチが継続される．

また Victream のスケジューラではスケジュール可能なサブタスクの定義に独自性がある．図 3 にスケジュール可能なサブタスクの例を示す．従来の DAG のフレームワークでは，スケジューラが選択可能なサブタスクは DAG 内の依存する親のサブタスクが全て完了し実行可能になったサブタスクだった．一方 Victream では，将来実行するサブタスクをスケジュールするため，スケジュール時に実行可能ではないサブタスクもスケジュール候補とできる．そのための拡張を行っている．新たにスケジュール候補となるサブタスクは，スケジュール時は実行可能ではなくとも，そのサブタスクが実行する時点では実行可能となることが保証されているサブタスクである．そのようなサブタスクは，未完了の全ての親のサブタスクが図 2 のサブタスクキューにすでにキューイングされ，そのサブタスクがサブタスクエクセキュータで実行されるまでに先にキューイングされた親のサブタスクが完了するサブタスクである．また，このとき未完了の全ての親のサブタスクは，サブタスクをスケジュールする GPU と同一の GPU にキューイングされている必要がある．図 3 はこのような方針でスケジュール可能なサブタスクを再定義したものである．

3.　シミュレーションによる性能評価

本節では動的でヒューリスティックな Victream のスケジューラの性能をシミュレーションにより評価する．シミュレーションでは DAG が与えられた際に取り得るスケ

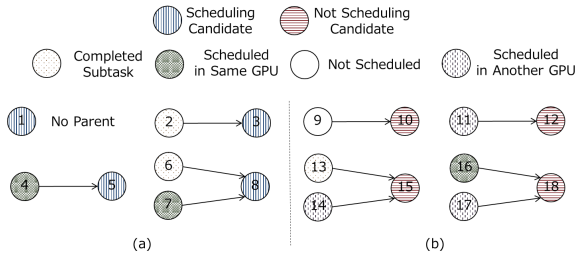


図 3　Victream スケジューラのスケジューリング候補と親のサブタスクとの関係．(a) スケジューリング候補．(b) スケジューリング候補ではない．

表 1　シミュレーションに用いたパラメータの値

GPU I/O Bandwidth	6377 MB/s
File Read Throughput	2482 MB/s
File Write Throughput	1680 MB/s

ジュールを Brute-Force で全探索し，最も計算時間が短いスケジュールを最良のスケジュールとする．そして得られた最良のスケジュールと Victream が選択したスケジュールの計算時間を比較することで Victream の性能を評価する．本節でシミュレーションに用いるアプリケーションは，[1] で使用したマイクロベンチマークの内，ロジスティック回帰，ブラーフィルタ，行列積とする．

以下本節では 3.1 節で作成したシミュレータの有効性を検証する．次に 3.2 節で作成したシミュレータを用いて Victream の性能の評価を行う．

3.1　シミュレータの有効性検証

本節では作成したシミュレータの有効性を検証するため，[1] で使用したマイクロベンチマークにおける Victream の実機の計算時間と，シミュレータが計算した計算時間を比較した．用いた実機はスタンドアロンのサーバであり，I/O スロットに NVIDIA Tesla K20 を 4 個挿入した．シミュレーションに用いたパラメータを表 1 に示す．これらは用いた実機の実測で得られた値である．

実機の計算時間と，シミュレーションで得られた計算時間の比較を図 4 に示す．これらの計算時間の乖離は最大 9%であり，有効なシミュレーションが行えていることを確認した．

3.2　Victream の性能検証

次に作成したシミュレータを用いて Victream のスケジューラの性能を評価した．表 2 にシミュレーションに用いたマイクロベンチマークの条件を示す．ここで今回シミュレーションに用いた DAG は極めて小規模だが，この規模でも各マイクロベンチマークの計算に半日程度を要する．これは Brute-Force で全探索を行うスケジュールの数が DAG の頂点の数や用いる GPU 数に対して組み合わせ爆発で増大するためである．表 2 から DAG の頂点の数等

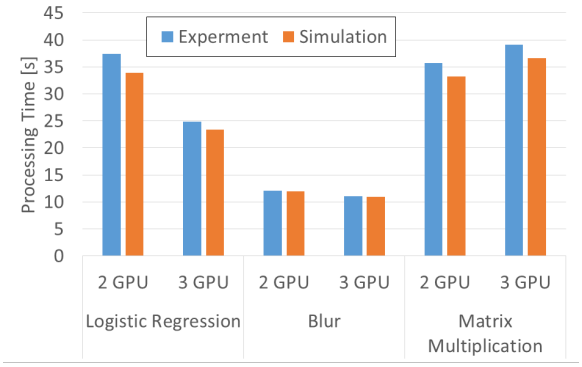


図 4　Victream の実験とシミュレーションによる計算時間の比較

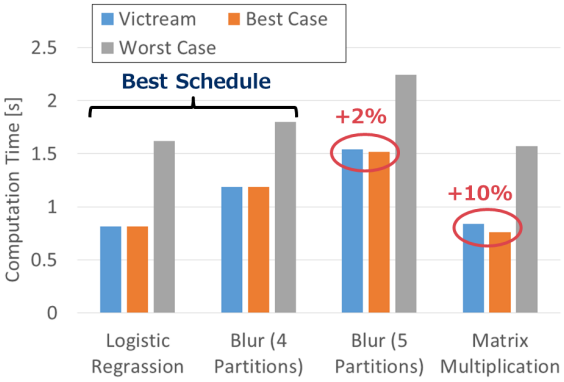


図 5　シミュレーションによる最良のスケジュールと Victream の計算時間の比較

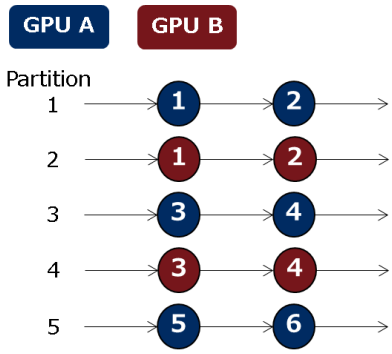
を少しでも増加させると，計算時間が大きく増大する．

図 5 にシミュレーションで得られた最良のスケジュール，最悪のスケジュール，Victream の計算時間をそれぞれ示す．Victream の計算時間もシミュレータで計算した結果である．図 5 を参照すると，ロジスティック回帰と入力が 4 パーティションのブラーフィルタでは Victream は最良のスケジュールと同じ性能が得られる一方，5 パーティションのブラーフィルタでは 2%，行列積では 10%最良のスケジュールより計算時間が増加することがわかった．

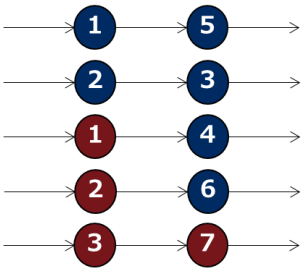
この計算時間の増加を考察するため，Victream と最良のスケジュールのサブタスクの実行順を調べた．その結果を図 6 と図 7 に示す．図 6 のブラーフィルタでは，Victream では 2 つの GPU に対するサブタスクの割り当てが入力データのローカリティを考慮するため 6 個と 4 個となり，負荷が不均一となっている．一方最良のスケジュールも割り当てが 6 個と 4 個だが，ブラーフィルタでは図の左側のサブタスクの方が入力データパーティションをファイルから読みだすため処理負荷が高い．よって最良のスケジュールでは 4 個のサブタスクしか処理しない GPU がより負荷の高いサブタスクを担当することで，Victream より処理負荷の割り当てが均一となっている．また図 7 の行列積では，Victream と最良のスケジュールのサブタスクの実行順の形が類似しているため違いがわかりにくい．そこでシ

表 2　各マイクロベンチマークのシミュレーション条件

	Logistic Regression	Blur (4 Partitions)	Blur (5 Partitions)	Matrix Multiplication
No. of GPU	1	2	2	2
GPU Memory Capacity	1650 MB/s			
GPU Memory Use Threshold (Data Load)	0.5	0.66		
GPU Memory Use Threshold (Data Swap)	0.33			
No. of Input Data Partitions	3	4	5	4
No. of Iterations	2	2	2	2

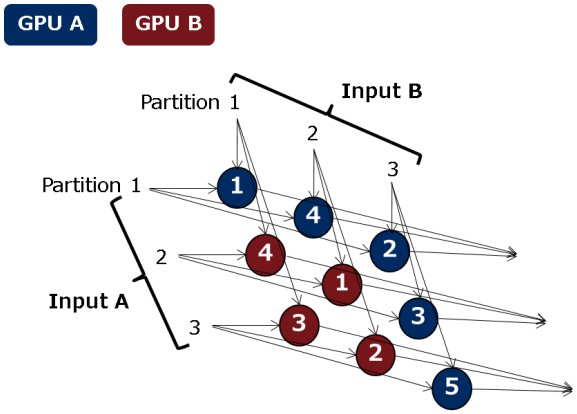


(a) Victream

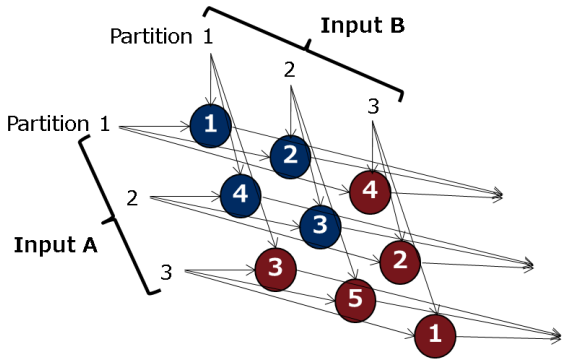


(b) 最良のスケジュール

図 6　入力データが 5 パーティションのブラーフィルタのサブタスクの実行順のシミュレーション結果．数字はサブタスクの実行順．



(a) Victream

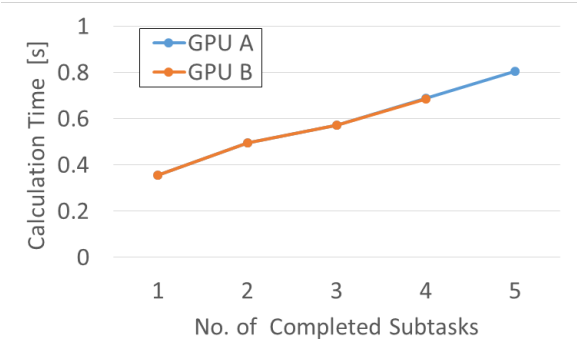


(b) 最良のスケジュール

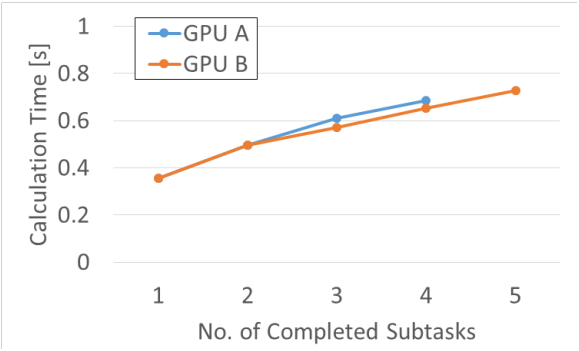
図 7　行列積のサブタスクの実行順のシミュレーション結果．数字はサブタスクの実行順．

ミュレータのログを確認したところ，Victeam では計算の後半で，未実行のサブタスクの入力データがスワップアウトされているためサブタスクをスケジュールできず，GPU がアイドル状態となる時間が長かった．これをさらに確認するため，図 8 に示すようにサブタスクの完了数に対する計算の経過時間を調べた．2 つの手法で差が確認できるのは各 GPU が 3 つめ以降のサブタスクを処理する場合である．最良のスケジュールでは 5 つのサブタスクを計算する GPU の完了時間が早く，GPU へのデータロードやサブタスクの実行が効率的に行われていることが分かる．

本節では動的でヒューリスティックなスケジュール手法を採用している Victream の性能をシミュレーションで求めた最良のスケジュールと比較することで評価した．その結果，Victream のスケジューラは最良のスケジュールが実現する最短の計算時間ではないものの，最短の計算時間からの増加を 10%以内に抑えた性能が実現できることがわかった．Victream が想定する応用では，ユーザアプリケーションからユーザ定義関数を頂点とした DAG を受け取り，その DAG の GPU での実行を最適化する．従ってユーザ定義関数の中身をミドルウェアから知ることができず，ユーザ定義関数の計算コストはわからない．そのため最良のスケジュールを計算することは不可能である．またもし，特定の応用でユーザ定義関数の計算コストがわかった場合でも，最良のスケジュールを求める場合，今回シミュレーションを行った表 2 のように非常に小規模の DAG でもスケジュールの全探索に半日を要する．しかも Victeam



(a) Victream



(b) 最良のスケジュール

図 8 シミュレーションによる行列積のサブタスク完了数に対する計算経過時間

が想定する応用の DAG の規模は [1] で評価したように今回のシミュレーションより 2-3 桁が大きく，計算を行うことは不可能である．これに対し，GPU の計算オーバヘッドとならない高速なスケジューリングを実現する Victream は，最良のスケジュールに対して計算時間の増加を 10% に抑えた高い計算性能を実現できることが本評価で確認できた．

4. I/O デバイスの分離アーキテクチャへの適用

本節では Victream を I/O デバイスの分離アーキテクチャに適用し，Victream の有効性を検証する．I/O 分離アーキテクチャはホストと I/O デバイスを分離し，I/O デバイスを必要に応じてホストに柔軟に割り当てることで計算リソースの効率的な運用を実現する技術である．従来の計算機システムでは I/O デバイスはホストの I/O スロットに挿入して用いられ，各ホストに占有されていた．このような構成では，I/O デバイスが常に利用されない場合，I/O デバイスの稼働率が低下しリソース利用効率が下がる．また，ホストの I/O スロットの数は有限であるため，例えば GPU を複数用いたい場合，1 つのホストに搭載可能な GPU の数は I/O スロットの数に制限される．このような課題を解決するアーキテクチャが I/O デバイスの分離アーキテクチャである．

(a) HBA

(b) I/O リソースボックス

図 9 I/O デバイスの分離アーキテクチャを実現する ExpEther

しかし I/O デバイスの分離アーキテクチャでは，一般にホストと分離した I/O デバイスを接続するネットワークの帯域が，ホスト内部の I/O バスの帯域より小さい．従って I/O デバイスの分離アーキテクチャを用いて GPU のリソースプールからホストに GPU を割り当てて Out-of-Core 処理を行う場合，スタンドアロンサーバと同様にデータスワップの通信を行うネットワークの帯域が性能ボトルネックとなる．従って本評価ではこの性能ボトルネックを緩和する Victream の有効性を検証する．

著者らはこれまで，イーサネットを用いて I/O デバイスの分離アーキテクチャを実現する ExpEther を提案した [6]．ExpEther ではホストと I/O デバイスのリソースプールをイーサネットで接続し，リソースプールの I/O デバイスを必要に応じて所望のホストに割り当てる柔軟な運用を実現する．図 9 に今回の評価で用いる ExpEther の写真を示す．HBA(Host Bus Adaptor) はホストの I/O スロットに挿入して用いられ，ホストをイーサネットに接続する．また I/O リソースボックスは内部に PCI Express(PCIe) に準拠した I/O デバイスを挿入する．この I/O リソースボックスをイーサネットに接続することで内部の I/O デバイスがリソースプール化される．このような構成によりホストの内部の PCIe バスが HBA と I/O リソースボックスの間でトンネリングされ，ホストの PCIe ネットワークがイーサネットに拡張されてリソースボックスと接続される．PCIe バスのトンネリングは，HBA と I/O リソースボックスに実装した ExpEther ブリッジと呼ぶハードウェアで行っている．

図 10 に評価に用いた実験系を示す．ホストと各 GPU

は 2 本の 40GbE で接続した．ExpEther にはリンクアグリゲーション機能があり，この構成によりホストと GPU は 80Gb/s のイーサネットのコネクションで接続される．HBA と I/O リソースボックスに実装した ExpEther ブリッジは x8 PCIe 3.0 と 2 つの 40GbE のインターフェースを保持している．

図 11 と図 12 にマイクロベンチマークを用いて行った性能評価の結果を示す．用いたマイクロベンチマークは [1] で用いたベンチマークと同じである．図の縦軸はデータ処理量と単位時間あたりのデータ処理回数の積である．つまり単位時間あたりのデータ処理量を示す．今回の測定では用いる GPU 数に関わらず Out-of-Core 処理を行うため，1GPU あたりのデータ処理量を一定とした．つまり N 個の GPU を用いた測定では，1 個の GPU を用いた測定と比較して N 倍のデータ処理を行っている．

図 11 と図 12 の凡例は，Victream は提案手法，Victream(w/o local list) は 2.2 節で述べたスケジュール候補の再定義を考慮しないローカリティアウェアスケジューリング，PTask [7] と FIFO(First In First Out) は従来手法のスケジューリングである．ここで FIFO では親のサブタスクが完了し実行可能となったサブタスクから順に実行待ちキューにキューイングし，キューの先頭のサブタスクから貪欲法で GPU の演算リソースを割り当てる手法である．FIFO では Out-of-Core 処理のデータ I/O 量を考慮しない．また PTask は state-of-the-art の手法であり，FIFO と類似しているが，キューの先頭のサブタスクに GPU の演算リソースを割り当てる際に入力データのキャッシュを考慮する．すなわち，もし演算リソースが空いた GPU が実行待ちキューの先頭のサブタスクの入力データの過半を保持する GPU と異なる場合，スケジューラはデータを保持する GPU がしばらく待つと使用可能になることを期待し GPU の割り当てを一定時間待つ．しかし，このような PTask でも，Out-of-Core 処理時にキューの先頭のサブタスクの入力データがホストのメインメモリにスワップアウトされていた場合，FIFO と同じ動作になる．

図 11 と図 12 の結果を参照すると，Victream を I/O デバイスの分離アーキテクチャに適用した場合，従来手法の PTask や FIFO より優れた計算性能が実現できることがわかる．Victream は 1 つの GPU を用いた場合に GPU の演算リソースを従来手法より有効に活用し，また使用する GPU 数を増加させた場合も大きな計算性能の向上を実現できる．従来手法との差はブラーフィルタで最も大きく，4 つの GPU を用いた場合，Victream の計算性能は PTask より 296%向上する．また，ブラーフィルタやソートで用いる GPU 数に対し計算性能が飽和する原因は，ベンチマークの入出力データの格納に RAMDisk を用いているためである．今回の評価は GPU 数を増加させても，GPU あたりのデータ処理量が一定となる条件で行ったが，ベンチマークの入出力データのファイル I/O は増加する．特に行列積以外のベンチマークではファイル I/O は GPU 数に比例す

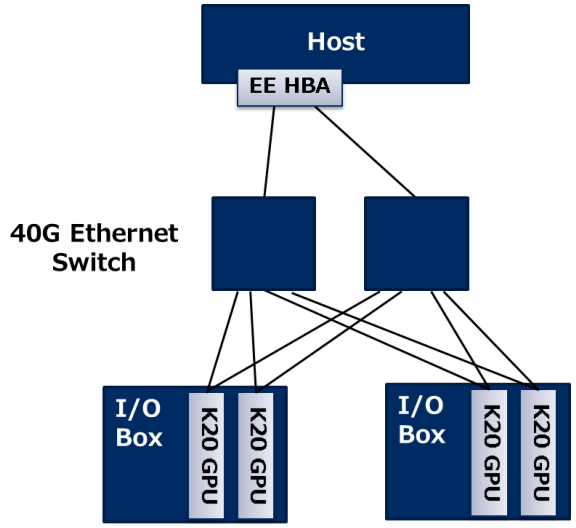
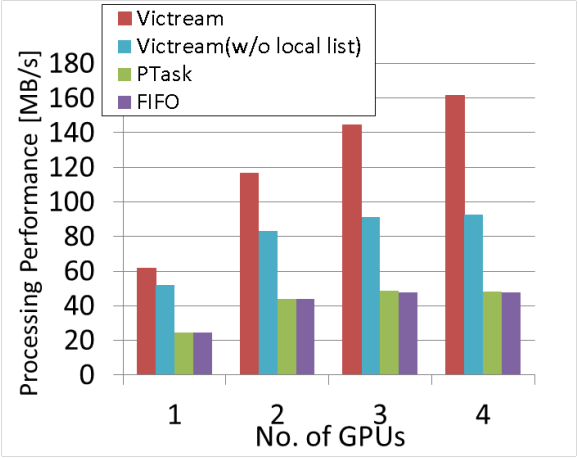
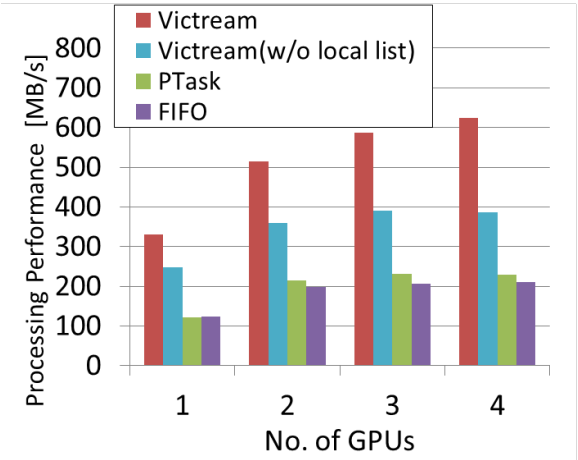


図 10 ExpEther を用いてホストから分離した GPU の性能評価の実験系

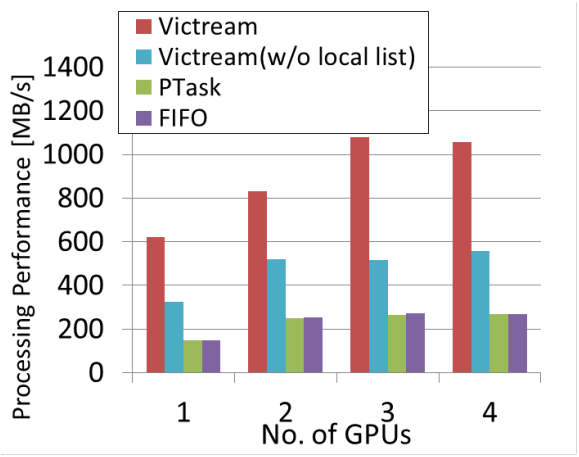


(a) ロジスティック回帰

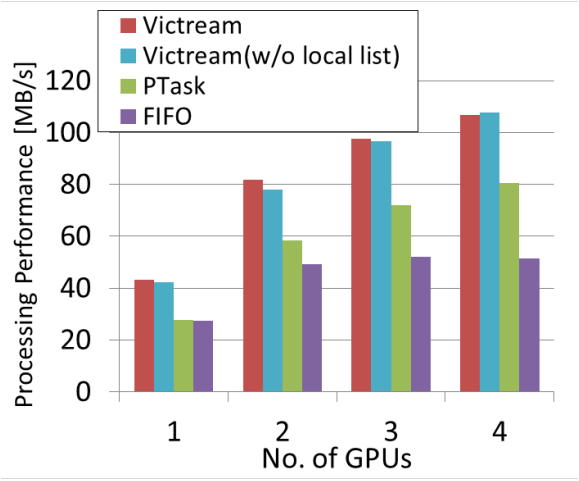


(b) ソート

図 11 ホストから分離した GPU の計算性能 (ロジスティック回帰とソート)



(a) ブラーフィルタ



(b) 行列積

図 12 ホストから分離した GPU の計算性能 (ブラーフィルタと行列積)

る．このため GPU 数の増加に伴い，計算スループットが飽和する原因となった．

次に，今回得られた Victream の結果を [1] でスタンドアロンサーバを用いて評価した結果と比較する．結果を表 3 に示す．表ではスタンドアロンサーバの性能を 1 とし，今回得られた性能を相対値で示している．表を参照すると，I/O デバイス分離アーキテクチャでは，スタンドアロンサーバと比較して，ベンチマークにより異なるが概ね 50%-100%の性能が得られることがわかった．このことから，I/O デバイスの分離アーキテクチャの実現コストによるが，GPU の稼働率が概ね 50%以下の計算プラットフォームでは GPU が必要なときだけ I/O デバイスの分離アーキテクチャで GPU を柔軟に割り当てた方が効率的なプラットフォームを実現できる可能性がある．またそのとき，I/O デバイスの分離の性能オーバヘッドを補うため，スタンドアロンのときより多い GPU を割り当てる必要がある．今回の評価結果から，サーバの I/O スロットに直接 GPU を挿入する場合に対し，I/O デバイスの分離アーキテクチャ

表 3 ExpEther を用いてホストから分離した GPU の性能 (I/O スロットに GPU を挿入したときの性能を 1 とした場合)

	Logistic Regression	Sort	Blur	Matrix Multiplication
1 GPU	0.58	0.67	0.92	0.87
2 GPU	0.57	0.68	0.92	0.81
3 GPU	0.49	0.71	1.07	0.67
4 GPU	0.42	0.69	0.97	0.60

で 2 倍の GPU を割り当てると同等の性能を期待できる．

5. まとめ

本論では，複数の GPU を用いた Out-of-Core 処理向けのミドルウェアである Victream の性能をシミュレーションにより評価した．Victream は Out-of-Core 処理で性能ボトルネックとなる GPU へのデータ I/O 量を最小化する．このため Victream は動的でヒューリスティックなスケジューラを採用しているが，実現されるスケジュールは最良のスケジュールを保証しない．本論ではシミュレーションを用いて取り得るスケジュールを Brute-Force で全探索し，得られた最良のスケジュールと Victream の計算時間を比較することで Victream の性能を評価した．その結果，Victream は最良のスケジュールではないが，最良のスケジュールの計算時間から 10%以内に計算時間を抑えたスケジュールを実現できることがわかった．本論ではさらに，GPU の柔軟なリソースの運用を実現する I/O デバイスの分離アーキテクチャに Victream を適用した場合の有効性を評価した．その結果 Victream はホストと GPU の間のデータ I/O を抑制し，従来手法より最大で 296%性能を向上する高い計算性能が実現できることがわかった．Victream と I/O デバイス分離のアーキテクチャを組み合わせて用いる場合，GPU の稼働率が概ね 50%以下のプラットフォームではそれぞれのサーバに GPU を専用で割り当てるより，効率的なプラットフォームを実現できる可能性があることがわかった．

参考文献

[1] Suzuki, J., Hayashi, Y., Kan, M., Miyakawa, S., Takenaka, T., Araki, T. and Kitsuregawa, M.: Victream: Computing Framework for Out-of-Core Processing on Multiple GPUs, *Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, ACM, pp. 179–188 (2017).

[2] Yuan, Y., Salmi, M. F., Huai, Y., Wang, K., Lee, R. and Zhang, X.: Spark-GPU: An accelerated in-memory data processing engine on clusters, *Big Data (Big Data), 2016 IEEE International Conference on*, IEEE, pp. 273–283 (2016).

[3] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. et al.: Tensorflow: a system for large-scale machine learning., *OSDI*, Vol. 16, pp. 265–283 (2016).

[4] Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: distributed data-parallel programs from se-

quential building blocks, *ACM SIGOPS operating systems review*, Vol. 41, No. 3, ACM, pp. 59–72 (2007).

[5] 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 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, pp. 2–2 (2012).

[6] Suzuki, J., Hidaka, Y., Higuchi, J., Hayashi, Y., Kan, M. and Yoshikawa, T.: Disaggregation and Sharing of I/O Devices in Cloud Data Centers., *IEEE Trans. Computers*, Vol. 65, No. 10, pp. 3013–3026 (2016).

[7] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B. and Witchel, E.: PTask: operating system abstractions to manage GPUs as compute devices, *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ACM, pp. 233–248 (2011).