Vol.2016-OS-138 No.5
2016/8/8

リソース分離アーキテクチャのためのアクセラレータミドルウェアVictreamの提案

鈴木 順[1,3,a)]　菅 真樹[1]　林 佑樹[1]　荒木 拓也[1]　宮川 伸也[2]　喜連川 優[3]

概要：コンピュータに複数の GPU(Graphic Processing Unit) を用いて大規模データのデータパラレル処理を行う需要が増加している．GPU を用いた処理では画像と行列演算が同一処理内で混合する等，異なるデータ形式を扱う必要がある．従来の DAG(Directed Acyclic Graph) 型ミドルウェアは複数の計算リソースを用いたデータパラレル処理を容易化する一方，GPU による種々のデータ形式の混合処理には対応していなかった．本稿では，GPU 数を自由に増加できる環境において，データ形式が混合する大規模データのデータパラレル処理の性能を GPU 数に比例して向上させるミドルウェア，Victream を提案する．Victream では，DAG で扱うデータに属性を保持させることで，異なるデータ形式の処理を同一のフレームワークで対応する．また，大規模データ処理をメモリ容量が制限された GPU で行う場合，Out-of-Core 処理となり GPU へのデータ I/O が性能ボトルネックとなることが多かった．Victream では各属性のデータを共通に管理し，貪欲法を用いた処理とデータ I/O のスケジューリングにより GPU に対するデータ I/O を最小化する．これらの構成により，より多様なデータ形式の混合処理に対し，データ I/O のボトルネックを抑制し，用いる GPU 数にスケールする高い処理性能を提供する．試作による評価では，従来手法に対し最大で 145%の性能向上が得られた．

Victream: Accelerator Middleware for Resource Disaggregated Architecture

Jun Suzuki[1,3,a)]　Masaki Kan[1]　Yuki Hayashi[1]　Takuya Araki[1]　Shinya Miyakawa[2]
Masaru Kitsuregawa[3]

1. はじめに

リソース分離アーキテクチャでは，コンピュータの I/O デバイスを分離し，コンピュータと I/O デバイスをインターコネクションで接続する．我々はリソース分離アーキテクチャを実現するインターコネクションの一例として，PCI Express (PCIe) をイーサネットで伝送する ExpEther を提案した [1]．リソース分離アーキテクチャでは，コンピュータに I/O デバイスを柔軟に接続することができる．これにより GPU(Graphic Processing Unit) 等のアクセラレータを多数接続し，大規模なデータ処理を行うことが可能となる．

近年，GPU は大規模データ処理を高速化するデバイスとして注目されている．GPU は大規模データ処理のフレームワークである Spark[2] や，機械学習フレームワークの Tensorflow[3] にも用いられている．

本稿では，リソース分離アーキテクチャを想定し，GPU 数を自由に増加できる環境において，用いる GPU 数に比例してデータパラレル処理の処理性能を向上させるミドルウェアを提案する．特にデータパラレル処理では画像と行列演算が同一処理内で混合する等，異なるデータ形式を扱う必要がある．本稿では異なるデータ形式の大規模データ処理を同一フレームワークで扱うための方式に着目する．

[1] NEC　システムプラットフォーム研究所
System Platform Research Laboratories, NEC
[2] NEC　IoT デバイス研究所
IoT Devices Research Laboratories, NEC
[3] 東京大学生産技術研究所
Institute of Industrial Science, the University of Tokyo
[a)] j-suzuki@ax.jp.nec.com

分散データパラレル処理を複数の計算ノードを用いて分散して行う場合，DAG(Directed Acyclic Graph) 型ミドルウェアが多く用いられている．クラスタ向けでは Dryad[4] や Spark[5] が知られている．アプリケーションプログラマは，ミドルウェアが提供する API(Application Programming Interface) を用いて所望の処理フローを示す DAG を作成する．DAG の各ノードは入力データに適用するユーザ定義関数である．DAG の実行では，ミドルウェアが処理データのパーティションと分散を自動で行い，各ホストでユーザ定義関数を実行する．複数の GPU を用いるための DAG 型ミドルウェアとしては PTask[6] が提案されている．

ここで異なるデータ形式を混合して扱うデータパラレル処理を複数の GPU を用いて大規模データに対し実行する場合，従来の DAG 型ミドルウェアには 2 つの課題があった．1 つ目はミドルウェアが扱うデータ形式や処理の種類に制限があることである．また，これまでに開発された GPU ライブラリも再利用できない．複数の GPU を用いるデータパラレル処理では，処理データをパーティションに分割し，パーティション内の各データ要素の処理を GPU に多数のスレッドを生成して並列に行わせる．ここで，例えば画像処理に代表的なステンシル計算では，パーティションの境界ピクセルを担当するスレッドは隣接するパーティションに含まれるピクセルを参照する．しかし，従来のミドルウェアでは隣接するパーティションを参照する仕組みを提供していない．また，行列の演算の例では密行列と疎行列に依存して，処理を行う GPU スレッドに必要な担当行列要素のオフセットの情報が異なる．また，パーティション内のデータ構成も行列の形式により異なる．しかし，従来のミドルウェアは行列の種類に応じた対応を行わない．またデータパーティションに対し, 行列演算を提供する GPU ライブラリ関数の適用に必要なパラメータも与えない．

2 つ目の課題は GPU の大規模データ処理で多く発生する Out-of-Core 処理で性能が低下することである．GPU が保持するメモリ容量は 10GB 程度であり，データセンタで用いられる汎用サーバに対し 2 桁オーダーが異なる．また GPU を用いてアプリケーションを高速化する場合，アクセラレータ化によりサーバ台数が削減される．従って個々の GPU が担当するデータ量は増大し，サーバを用いた処理より Out-of-Core 処理となる場合が多い．しかし，GPU のローカルメモリ帯域に対し，GPU のデータ I/O に使用するインターコネクションの帯域は 1 桁以上小さい．このため GPU 数を増加しても，インターコネクションの帯域が処理全体の性能ボトルネックとなり性能向上が制限されることが多い．

本稿では GPU 数に比例してデータパラレル処理の高い性能向上が得られるミドルウェア “Victream” を提案する．Victream は異なるデータ形式を混合して扱うための DAG 型ミドルウェアである．また，データ形式に柔軟に対応すると同時に，複数の GPU を用いた Out-of-Core 処理において GPU に実行させる処理とデータ I/O のスケジューリングを最適化し，データ I/O を抑制することで GPU 数に応じた高い処理性能を実現する．

具体的には GPU で分散処理を行うデータにデータ形式の属性を付与し，属性に応じた処理やデータの管理をミドルウェアで提供する．一方同時に，処理やデータ I/O のスケジューリングはデータの属性に依存せずミドルウェアで共通に扱えるようにする．これにより，データパーティションの内部構成はデータの属性に応じて整形し，ユーザ定義関数の実行時にはデータ形式の属性に応じた引数を渡す．また，汎用の GPU ライブラリの適用も可能とする．一方データ I/O と処理のスケジューリングでは，貪欲法を用いて GPU が保持するデータの再利用性を高めることで，GPU に対するデータ I/O を最小化する．また，GPU のデータ I/O と計算リソースの利用効率を最大化するため，パイプラインによりそれぞれの管理を行う．

以下本稿では，2 節で研究の動機を議論し，3 節で Victream のシステムアーキテクチャ，4 節で Victream のスケジューラの詳細，5 節で試作による評価結果についてそれぞれ述べ，最後に第 6 節でまとめる．

2. 研究の動機

本節では Victream 研究の動機について述べる．

2.1 リソースの追加による処理性能の向上

大規模なデータ処理では計算時間が増大する．これらの処理では計算リソースを追加することで計算プラットフォームの処理性能を向上し，計算時間を短縮することが重要である．計算クラスタで用いられることが多い Hadoop や Spark は，クラスタにコンピュータを追加することでこれを実現する．アプリケーションプログラムは，ミドルウェアが用意する API を用いて作成されていれば，コンピュータの増減に応じてプログラムを書き換える必要はない．

リソース分離アーキテクチャは使用するリソースの増減を柔軟に行うためのアーキテクチャである．リソース分離アーキテクチャはコンピュータから I/O デバイスを分離し，コンピュータと I/O デバイスをインターコネクションで接続する．これにより，コンピュータに必要なデバイスを必要なときにスケーラブルに接続する．接続される I/O デバイスとしては，GPU 等のアクセラレータや NVMe(NVM Express) 等の高速ストレージデバイスが想定される．これにより提供する処理のボトルネックとなっているリソースを自由に増減することができる．また，リソース分離アーキテクチャはボトルネックとなる I/O デバイスのみを増減

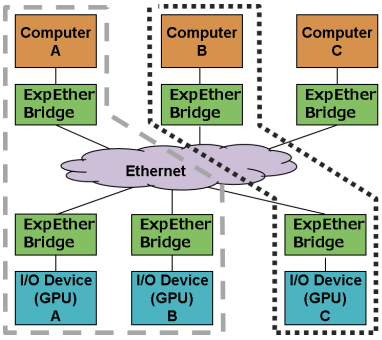


図 1　ExpEther の構成の一例．点線は I/O デバイスが割り当てられたコンピュータを示す．

するため，I/O デバイスの増設のために新たにコンピュータを増設する従来手法と比較して省スペース/低コストでデバイスの増減を行える．

著者らはリソース分離アーキテクチャを実現する手段として ExpEther を提案している [1]．ExpEther はコンピュータと I/O デバイスをイーサネットで接続する．図 1 に ExpEther の構成の一例を示す．コンピュータに対しては，I/O デバイスツリーが ExpEther で接続された I/O デバイスまで延長されるように仮想化される．このため I/O デバイスやデバイスドライバ，またコンピュータの OS に変更は不要である．イーサネットスイッチも標準品で良い．I/O デバイスのコンピュータへの割り当ては，自由に変更することができる．

本稿ではミドルウェアとリソース分離アーキテクチャを組み合わせ，処理データの規模やシステムの要求性能に応じて柔軟に処理能力を調整できるシステムを想定する．

2.2　異なるデータ形式が混合する処理への対応

複数の GPU を用いたデータパラレル処理では，処理データをパーティションに分割し，各パーティションを各々の GPU に処理させる．GPU が処理対象とするデータは画像，密/疎行列，Key-Value(K-V) ペア等様々なデータ形式が考えられる．また, これらを組み合わせたり，変換して異なるデータ形式が混合する処理も多い．従って異なるデータ形式に柔軟に対応できるミドルウェアが必要である．

しかし，従来の DAG 型ミドルウェアでは対応できる処理に制限がある．例えば画像等のステンシル計算では，データパーティションの境界領域のピクセルを計算するために隣接するパーティションを参照する必要がある．しかし従来のミドルウェアはこのような処理に対応していない．

また，従来のミドルウェアは対応できるデータ形式にも制限がある．さらに，これまで開発された GPU ライブラリを用いることができない．GPU にデータパーティションの処理を行わせる場合，GPU スレッドを多数生成し，並列計算を行う．各スレッドが担当するデータパーティション内のデータ要素を知るには，データ形式に応じたオフセットの情報が必要である．画像の例では担当するピクセルのパーティション内のオフセットと共に，分割前の画像全体に対するオフセットも画像の端を判別するために必要である．また，データパーティションに対して GPU ライブラリが提供する関数を適用する場合，関数に与えるパラメータが必要である．これらの情報は処理データのデータ形式により異なる．しかし従来のミドルウェアはこれらの異なるデータ形式に対応していない．

このように複数の GPU を用いたデータパラレル処理用のミドルウェアでは，異なるデータ形式に柔軟に対応することで，より多様なデータ形式の混合処理に対応する必要がある．

2.3　性能の最大化

本稿では処理性能を GPU 数に比例して向上させることを目的としている．この特性を実現するには，実行する処理が計算ボトルネックであり，GPU 数の増加により各 GPU の処理負荷が減少し，結果として全体の計算時間が短縮される必要がある．

しかし，GPU を用いた計算では GPU へのデータ I/O が性能ボトルネックとなることが多い．特に複数の GPU を用いて大規模なデータ処理を行う場合，Out-of-Core 処理となることが多く，GPU へのデータ I/O が頻繁に発生する．なぜなら，GPU が保持するメモリは 10GB 程度であり，今日データセンタで用いられているサーバよりオーダーが 2 桁小さい．また，GPU を用いることにより，サーバ台数の削減効果があり，結果として個別の GPU が担当するデータ量はサーバより増加する．しかし GPU を接続するインターコネクションの帯域は PCIe で 16GB/s, ExpEther では 10GB/s であり，GPU ローカルメモリの帯域である 200GB/s 以上に 1 桁及ばない．従って，GPU による大規模データの処理性能はデータ I/O がボトルネックとなる可能性が高い．

ここで，各 GPU に実行させる処理とデータ I/O のスケジューリングを工夫することにより，データ I/O のボトルネックを抑制することが可能である．それには GPU メモリが保持するデータをできるだけ再利用するように GPU に実行させる処理のスケジューリングを行えば良い．これにより GPU のデータ I/O の総量を削減できる．また，GPU は処理とデータ I/O の制御をホストのプログラムからそれぞれ独立に行う．従って GPU が処理を実行中でも，できるだけ GPU へのデータ I/O を並列に行い，GPU のデータ I/O 資源を稼働させることが有効である．

しかし，従来の DAG 型ミドルウェアである Vispark[7] や PTask では，データ I/O の最小化に着目していない．このため，GPU の演算毎に処理データを GPU にロードする不要やデータ I/O を行っていたり，Out-of-Core 処理において GPU メモリのデータの再利用を考慮せずにスケ

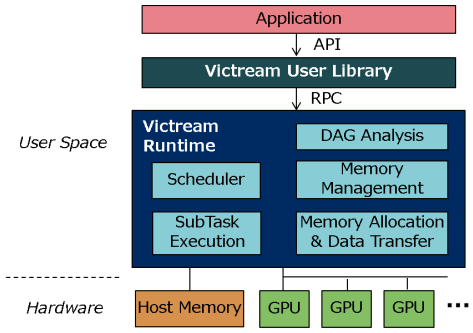


図 2　Victream のアーキテクチャ.

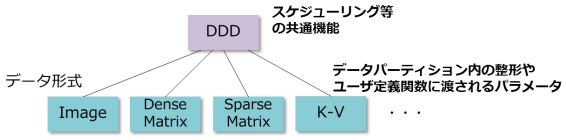


図 3　DDD のデータ形式.

ジューリングを行い，データ I/O の総量が増大したりする．よって本稿では，従来手法が注力していなかったデータ I/O の最小化に着目し，大規模データの Out-of-Core 処理でも，高い性能が得られるようにする．

3. システムアーキテクチャ

Victream は DAG 型のミドルウェアであり，所望の処理を DAG で記述し，ミドルウェアに DAG の処理を実行させる．図 2 に Victream のアーキテクチャを示す．Victream はライブラリとランタイムから構成される．アプリケーションプログラムは Victream ライブラリが提供する API を用いてプログラムを記述する．これによりプログラム内で DAG が生成され，RPC(Remote Procedure Call) により DAG の実行がランタイムに依頼される．ランタイムでは，受信した DAG をホストが保持する複数の GPU を用いて処理する．

Victream では GPU に分散して処理するデータオブジェクトを DDD(Distributed Data among Devices) とし，DDD をデータパーティションに分割し，データパーティションの単位で各 GPU に処理を割り振る．Victream は更に，図 3 に例を示すように DDD に属性を持たせ，DDD の属性に応じた制御をミドルウェアに行わせる．同時に，処理とデータ I/O のスケジュール最適化等の共通制御は，DDD の属性とは無関係に行う．

DDD のデータパーティションは Victream の管理下の GPU のいずれかがローカルメモリに保持するか，ホストのメインメモリに退避される．これらのデータパーティションの管理や，パーティションに対する処理の実行は Victream が行い，アプリケーションプログラムには透過である．将来的には Victream を活用し，データパーティションを NVMe 等の記憶デバイスに保持することも可能である．

表 1　DDD のオペレータ.

Transformations	DDD DDD::map(f, MM)
	DDD DDD::zip(DDD)
	DDD DDD::mapPartitions(f, MM)
	DDD DDD::groupByKey(MM)
	DDD DDD::join(f, DDD, MM)
Actions	DDD::reduce(f_{gpu}, f_{cpu}, MM)
	void DDD::outputFile($filePath$)
	void DDD::storeDDD($DDDPath$)

以下では Victream ライブラリとランタイムについて説明する．

3.1 Victream ライブラリ

Victream ライブラリはアプリケーションプログラム内で使用され，プログラムの処理を示す DAG を作成し，DAG の実行をランタイムに依頼する．Victream ライブラリは Spark の RDD(Resilient Distributed Datasets) と同様に，DDD のメソッドをオペレータとして定義し，オペレータが呼ばれることで DAG を生成する．表 1 に DDD のオペレータの一例を示す．

オペレータには Transformations と Actions の 2 種類が定義されている．Transformations は返り値として新たな DDD を返す．この返り値は，DDD にアプリケーションプログラムで指定された処理を行うことで得られる出力 DDD である．出力される DDD にさらにオペレータを適用することで，DAG のノードが増加していく．一方 Actions が呼ばれると，それまでライブラリ内で生成した DAG の実行がランタイムに依頼される．

表 1 に示すオペレータにおいて，f はアプリケーションプログラマが与えるユーザ定義関数である．f は DDD のデータパーティションが保持する各データ要素に適用される CUDA で記述された関数を含む．この関数は，ランタイムにより自動で生成される GPU スレッドにより実行される．一方，引数 MM(Method Meta-Information) は f を実行する際に必要な情報を含む．MM が含む情報の一例は実行するオペレータが出力する DDD の属性や，出力 DDD のデータ要素の型 (int や float 等)，出力 DDD のために GPU に確保するメモリの容量である．メモリ容量は出力データ要素数から間接的にランタイムが計算する場合もある．また，f は実行時にランタイムからデータパーティションの処理に必要な情報を引数として受け取る．この情報は処理を実行するデータパーティションの DDD の属性に依存する．この情報から GPU の各スレッドは担当するデータ要素のオフセット等がわかり処理を実行できる．あるいは，この情報を用いて汎用の GPU ライブラリを呼び出すことができる．

f が実行時にランタイムから渡される引数と，f が処理

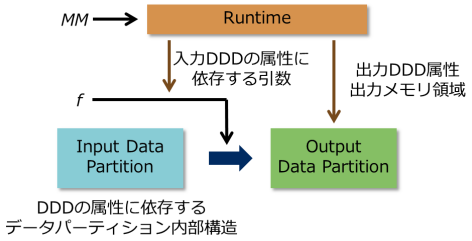


図 4　入出力データパーティション，処理，及びパラメータの関係．

を行うデータパーティションの内部構造は DDD のデータの属性に依存して異なる．f や MM と処理の関係を図 4 に示す．これにより，処理データの属性に応じて柔軟な処理が実現できる．例えば処理データの属性が画像等でステンシル計算を行う場合，MM は各 GPU スレッドが参照する隣接する要素数を含む．すると，Victream は f を実行する際に，データパーティションの境界領域に参照される以上の要素数を冗長に用意する．MM が出力 DDD の属性として画像を指定していれば，このステンシル計算により出力される DDD の属性は画像である．また，ステンシル計算を行う f の引数として，データパーティションが画像全体の内部に占める位置を渡す．これにより，本来の画像の端を処理する GPU スレッドは，画像の端を意識して計算を行うことが可能となる．また別の例として，Victream はこれまでに開発された GPU ライブラリを用いることを可能としている．例えば疎行列計算では，疎行列を DDD とし，データパーティションには元の疎行列を分割した部分行列が含まれる．このとき f として，ホストから cuSPARSE が提供する関数を呼ぶユーザ定義関数がアプリケーションプログラムから渡される．f の実行時にはデータパーティションに対し cuSPARSE の関数を実行するために必要な非ゼロ要素数や行/列数の情報がランタイムから渡される．

これらの仕組みを用いてアプリケーションプログラムは DAG を生成する．始めに，Victream ライブラリを include し，処理データを DDD としてインスタンス化する．インスタンス化する DDD はホストのファイルシステムやアプリケーションプログラムのヒープ領域，また Victream がストアする DDD を指定することができる．Victream がストアする DDD については後述する．DDD のインスタンス化時には，DDD のデータの属性や属性に応じたメタ情報を引数として与える．インスタンス化した DDD に対し，先に述べたオペレータを適用していくことにより，DAG を生成し，またその DAG の実行を Victream ランタイムに依頼する．

以下では表 1 に示した DDD のオペレータについて概要を説明する．

map: f を入力 DDD が保持する各データ要素に適用する．入出力 DDD の間で DDD に含まれるデータ要素の数やパーティションは変化しない．ステンシル計算時には MM 内に GPU スレッドが参照する隣接要素数を指定する．

zip: DDD を異なる DDD と関連付ける．zip の出力 DDD に map を適用することで 2 入力 map とできる．3 つ以上の DDD を関連付けることも可能である．

mapPartitions: map と類似するが，パーティションに含まれるデータ要素数が変化する．パーティションの数とその対応は変化しない．

groupByKey: K-V ペアの属性を保持する DDD に対して適用される．等しい Key 値を持つ K-V ペアの要素を出力 DDD の同一パーティションに出力する．

join: 引数として与えられた DDD と行列積の演算を行う．

reduce: 入力 DDD に対し reduction 演算を行う．まず各データパーティションに対し f_{gpu} を適用し，続いて各データパーティションの出力に対し f_{cpu} が適用される．reduce の返り値は RPC の返り値としてアプリケーションプログラムに渡される．f_{gpu} として Thrust で定義された関数を用いることが可能である．

outputFile: DDD をホストのファイルシステムに出力する．ファイルパスを引数に与える．

storeDDD: DDD を Victream がストアする DDD として出力する．DDD パスを引数に与える．

3.2　Victream ランタイム

図 2 に Victream ランタイムの構成を示す．DAG 解析部はアプリケーションプログラムから RPC で DAG を受信し，DAG のノードに対応するタスクについて個々にサブタスクを生成する．サブタスクの数はタスクの入力 DDD が保持するデータパーティションの数と等しい．タスクはアプリケーションプログラムから呼び出された DDD のオペレータに対応する．各 GPU での処理の実行はサブタスクを単位として行う．DAG 解析部はスケジューラに生成したサブタスクの実行を依頼する．ここでサブタスクの数が DAG の受信時に決定できないタスクは，タスクが依存する DAG 内の上流のタスクの実行が完了するまで待ち，サブタスクの数が決定できるようになった後にサブタスクの生成を行う．このようなタスクの例として groupByKey がある．groupByKey のサブタスク数は，入力 DDD が含むユニークな Key の数に依存するためである．

スケジューラは GPU に実行させるサブタスクとデータ I/O のスケジューリングを管理する．スケジューラの詳細は第 4 節で述べる．

メモリ確保及びデータ移動部はスケジューラから呼ばれ，指定された GPU のローカルメモリ上に入力データパーティションを用意すると共に，出力データパーティションを出力するためのメモリ領域を確保する．出力メモリ領域は出力 DDD の属性に依存して求められる．入力データパーティションが既に GPU メモリに存在する場合，何も

行わない．これにより，DAG の上流のサブタスクが生成し GPU メモリが保持するデータパーティションを，下流のサブタスクの実行に再利用することができ，ホストのメインメモリと GPU メモリの間の不要なデータ移動を削減できる．

サブタスク実行部は，スケジューラからサブタスクの実行を依頼される．サブタスク実行部はランタイムが管理する DDD のメタ情報を参照し，DDD の属性に応じた引数を作成し，ユーザ定義関数の実行時に引数を渡す．

メモリ管理部は GPU とホストのメモリリソースを管理する．メモリ管理部はスケジューラと，メモリ確保及びデータ移動部の双方から呼び出される．スケジューラは GPU のメモリ使用率を確認し，GPU へのデータパーティションのロード及びホストメモリへの退避を行う．メモリ確保及びデータ移動部は，GPU 及びホストメモリの確保や解放のためにメモリ管理部を呼び出す．

次にランタイムの DDD ストア機能と冗長計算機能について述べる．これらの機能は [8] において既に提案しているため，概要を述べるに留める．

DDD ストア機能は DAG の出力である DDD をアプリケーションプログラムの終了後もランタイム内で保持する機能である．ストアされる DDD はアプリケーションプログラムから与えられた名前が付与される．ストアされる DDD のパーティションは GPU メモリ，あるいはホストメモリで保持される．別の DAG はストアされた DDD を DAG の入力とすることができ，実行するサブタスクの入力データパーティションが GPU メモリに保持されている場合，ホストメモリからのロードが不要となる．これにより，従来は異なるアプリケーションプログラムの間でデータを受け渡す場合には GPU のデータ I/O を伴うファイルへの格納が必要だったが，この I/O コストが削減できる．

一方冗長計算機能はステンシル計算に関する機能である．各データパーティションに境界領域を冗長に保持させ，冗長に保持した領域も含めて計算を行う．これによりステンシル計算を連続して行う場合，計算の繰返し毎に行う境界領域の更新に関するデータ I/O を削減できる．

4. スケジューラ

Victream スケジューラは GPU における ouf-of-core 処理の処理性能を向上するため，GPU の計算とデータ I/O のリソース利用効率を高め，GPU へのデータ I/O を最小化する 2 つの機能を提供する．このため，Victream スケジューラは GPU のデータ I/O と計算の管理をパイプラインで行い，さらにデータ I/O を最小化するためにサブタスクの実行順を貪欲法を用いて決定する．スケジューラは Victream が扱うデータ形式に依存せず，データ形式が混合しても共通の機能として提供される．本節ではスケジューラの詳細について述べる．また図 5 にスケジューラの構成を示す．

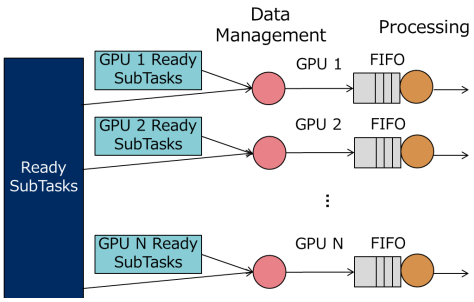


図 5　Victream スケジューラの構成．

4.1 データ I/O と計算のパイプライン管理

GPU を用いた計算では，GPU での計算の実行と GPU に対するデータ I/O を独立に行える．そこで Victream ではこれらのリソース利用効率を最大化することを目指す．特にデータ I/O のリソースは，データ I/O がボトルネックとなる場合，リソースを常に稼働することが重要である．Victream スケジューラは図 5 に示すようにサブタスクに関する 2 ステージパイプラインを用いてデータ I/O と計算の管理を行う．各 GPU には個別にパイプラインが割り当てられる．最初のステージがデータ I/O の制御であり，次がサブタスクの実行管理である．これらのパイプラインにサブタスクの投入を継続することにより，GPU の計算とデータ I/O のリソースを最大限利用することができる．

データ I/O のステージでは，投入されたサブタスクの入力データパーティションが GPU メモリ上に用意され，出力用の領域が確保される．またこれらのメモリ領域はサブタスクの実行が完了するまでロックされる．ここで入力データパーティションが既に GPU メモリ上に存在する場合，ここでの処理は入力データパーティションの領域をロックするだけで完了する．逆に，入力パーティションがホストメモリに退避されている場合，GPU メモリにロードが行われる．パイプラインへのサブタスクの投入は，GPU メモリの使用率が設定したしきい値を超えない間は常に継続される．

サブタスクの実行管理ステージでは，サブタスクの実行指示が GPU に対して行われる．このステージに入力されるサブタスクは，既に入出力の準備が完了している．またこのステージは FIFO(First In, First Out) キューを有しており，GPU へのデータ I/O とサブタスクの実行管理が非同期に行われる構造になっている．これにより，GPU の計算とデータ I/O に関するリソース使用率を独立に向上させることができる．サブタスクの実行完了後，入出力メモリのロックは解除される．

また，各 GPU のメモリ使用率が設定したしきい値を超えると，GPU からホストメモリへデータパーティションの退避が行われる．このアルゴリズムには LRU(Least

Recently Used) を用いている．

4.2 貪欲法によるデータ I/O 最小化

DAG が保持するサブタスクの実行順が，GPU メモリが保持するデータパーティションをより多く再利用する場合，ホストメモリから GPU メモリにデータパーティションをロードするデータ I/O が減少する．これにより，DAG 処理に伴う GPU へのデータ I/O を減らすことができる．そこで Victream スケジューラは，貪欲法を用いて GPU へのデータ I/O の総量が最小となるよう個々の GPU のパイプラインに投入するサブタスクの選択を行う．選択の候補となるサブタスクは，グローバルな準備完了サブタスクリストか，あるいは GPU 個別の準備完了リストに格納されている．ある GPU のパイプラインに投入するサブタスクの選択では，グローバルなリストと，その GPU に対応する個別のリストの中から候補を選択する．

グローバルな準備完了サブタスクリストに保持されるサブタスクは，DAG 内で依存する上流のサブタスクの実行が全て完了し，入力データパーティションが Victream が管理するメモリのいずれかに出力されている．従ってグローバルなリストに保持されるサブタスクは全ての GPU で実行可能である．一方ローカルな準備完了サブタスクリストに格納されているサブタスクは，全ての未完了の上流のサブタスクが，対応する GPU のサブタスク実行管理ステージで実行待ちの状態である．ローカルなリストに格納されているサブタスクは，今後パイプラインに投入された場合でも，依存するサブタスクはパイプラインの先で処理されるため，実行時には依存するサブタスクの実行が完了していることが保証できる．

このように構成により，Victream スケジューラはリストに含まれるサブタスクの候補から，サブタスクの入力データパーティションの GPU メモリへのロードと，メモリ空間を確保するために必要なデータパーティションのスワップアウトの合計量が最小のサブタスクを選択しパイプラインに投入する．これにより，各 GPU メモリが保持するデータパーティションの再利用を高め，GPU へのデータ I/O を最小化する．これにより複数の GPU を用いたデータパラレル処理において I/O ボトルネックを抑制し，用いる GPU 数に応じた高い処理性能を提供する．

5. 評価

Victream のプロトタイプを実装し評価を行った．提案手法により，単一のフレームワークで異なるデータ形式に対応し，Out-of-Core 処理に高い性能が提供できることを確認した．今回は方式の原理確認のため，ExpEther は用いず単一ホストで評価を行った．ホストは GPU を 4 個まで挿入できるものを用いた．用いた GPU は Tesla K20 である．GPU のローカルメモリは 5GB であり，単精度浮動小数点の演算スループットは 3.52Tflops である。 ホストの OS は Ubuntu 14.04, CUDA のバージョンは 7.5 を用いた．

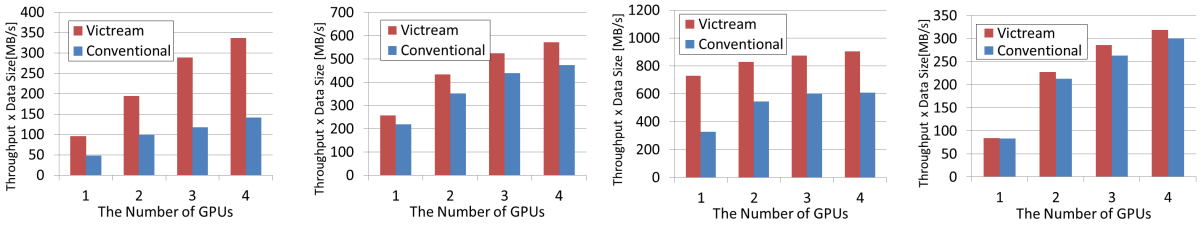
Victream は C++及び CUDA を用いて実装した．現在のコードライン数はライブラリとランタイムを合わせておよそ 10KL である．

評価には異なるデータ形式に対する処理である 4 つのマイクロベンチマークを用いた．それぞれロジスティック回帰，ソート，複数のイメージフィルタ，行列積である．ロジスティック回帰と行列積では行列を扱う．ソートではデータ形式を K-V ペアに変換し，Key に対してグルーピングを行う．イメージフィルタは画像に対する処理である．イメージフィルタでは Blur フィルタを画像に対し 10 回繰り返した．評価ではデータは ramdisk から入力した．

また提案手法のスケジューラと比較するため，PTask[6] と類似するスケジューラを従来スケジューラとして作成した．従来スケジューラでは DAG のサブタスクを，実行可能となった順に FIFO で実行する．結果，下流のサブタスクは上流の依存するサブタスクが完了した後に FIFO キューに積まれる．従来スケジューラはまた，入力データパーティションのローカリティを考慮する．GPU のスケジュール時に，探索されたサブタスクの入力データパーティションを最も多く保持する GPU が他の GPU だった場合，サブタスクの選択をスキップし，入力データパーティションを保持する GPU が空くのを待つ．

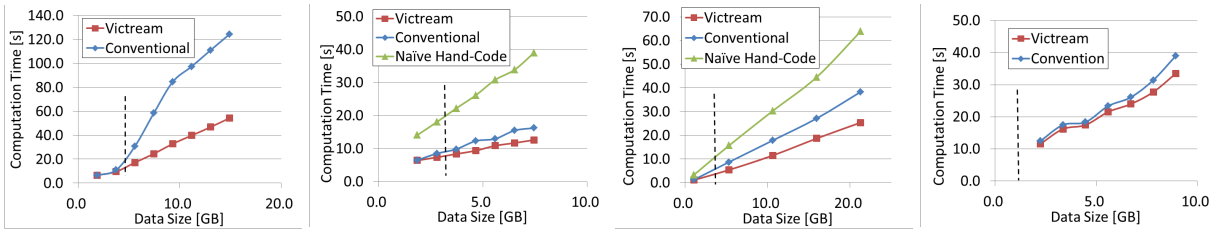
1 つ目の評価では GPU 数に対する処理性能のスケーラビリティを調べた．図 6 に得られた評価結果を示す．処理データは GPU 数に比例するよう設定した．つまり，1GPU あたりの処理データ量が等しくなるようにした．4GPU の性能が 1GPU の 4 倍の性能となれば処理性能が GPU 数に比例していることを示す．また，評価は Out-of-Core となるようデータサイズを十分大きく取った．Victream スケジューラは従来スケジューラより全てのベンチマークで優位な性能を示している．特にロジスティック回帰では 97%-145% の性能向上が得られた．また GPU 数に応じて処理性能を向上させられることを確認できた．ロジスティック回帰は Out-of-Core となる大規模なデータに対し繰返し計算である勾配法を行う．Victream スケジューラは毎回の繰返しにおいて，前回の繰返しから GPU メモリ上に残っているデータパーティションを入力とするサブタスクを優先してスケジュールする．これにより GPU へのデータ I/O を最小化できる．一方従来スケジューラでは，入力パーティションの場所は考慮せず，毎回の繰返しにおいて DAG の上流タスクに属するサブタスクから順に計算を行うため，データ I/O が処理ボトルネックとなる．

2 つ目の評価では，GPU 数を 3 つに固定し，処理データサイズを変化させた場合の性能を調べた．図 7 に評価結果を示す．図中の点線は処理が Out-of-Core となる境界を示し



(a) ロジスティック回帰　(b) ソート　(c)Blur フィルタ (10 回)　(d) 行列積

図 6　GPU 数を変更した場合の処理性能のスケーラビリティ.



(a) ロジスティック回帰　(b) ソート　(c)Blur フィルタ (10 回)　(d) 行列積

図 7　データサイズを変更した場合の処理時間.

ている．Victream と従来スケジューラでは，Out-of-Core ではない領域では性能が等しい．一方 Out-of-Core の場合，Victream スケジューラがデータ I/O 最小化を行うため性能が優れている．図 7 のソートと Blur フィルタはナイーブハンドコードの結果も示している．ナイーブハンドコードの結果は GPU における計算とデータ I/O の並列化を行わない．Victream が提供する処理とデータ I/O の最適化により，計算性能が向上することがわかる．

以上の評価結果は，Victream が画像や K-V ペア，行列等の異なるデータ形式の処理を単一の DAG 型ミドルウェアで扱えること示している．また，大規模データの処理時に生じる Out-of-Core 処理において，処理とデータ I/O のスケジュールを共通に最適化し，GPU に対するデータ I/O を最小化することで従来手法より高い性能が提供できることを示している．

6. 結論

本稿では，リソース分離アーキテクチャにおいてコンピュータに接続する GPU 数をスケーラブルに増加できる環境を想定し，GPU 数に応じて処理性能が向上できるミドルウェア Victream を提案した．Victream では，DAG で処理するデータに属性を付与し，データの属性に応じてユーザ定義関数に与えるパラメータや，処理データのデータパーティションの内部構造をミドルウェアで対応するようにした．これにより，画像や密/疎行列，K-V ペア等，異なるデータ形式を単一の DAG 型ミドルウェアで扱うことを可能とした．また，既存の GPU ライブラリの適用も可能とした．さらに，Victream ではデータの属性に非依存に共通の処理とデータ I/O のスケジューリング機能を提供した．これにより，大規模なデータ処理では一般的な Out-of-Core 処理において，貪欲法を用いて GPU に対するデータ I/O を最小化し，データ I/O ボトルネックを抑制して GPU 数に応じた高い処理性能を得ることを可能とした．Victream を試作し評価を行ったところ，従来手法と比較して最大で 145%の性能優位が得られた．

参考文献

[1] Suzuki, J. et al.: Expressether-ethernet-based virtualization technology for reconfigurable hardware platform, *14th IEEE Symposium on High-Performance Interconnects (HOTI'06)*, IEEE, pp. 45–51 (2006).

[2] Liu, J. F. and Hu, Y.: GPU Support in Spark and GPU/CPU Mixed Resource Scheduling at Production Scale, https://spark-summit.org/2016/events/gpu-support-in-spark-and-gpu-cpu-mixed-resource-scheduling-at-production-scale/.

[3] Abadi, M. et al.: TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, *Software available from tensorflow. org.*

[4] Isard, M. et al.: Dryad: distributed data-parallel programs from sequential building blocks, *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 3, ACM, pp. 59–72 (2007).

[5] Zaharia, M. et al.: 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] Rossbach, C. J. et al.: 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).

[7] Choi, W. and Jeong, W.-K.: Vispark: GPU-accelerated distributed visual computing using spark, *Large Data Analysis and Visualization (LDAV), 2015 IEEE 5th Symposium on*, IEEE, pp. 125–126 (2015).

[8] 鈴木順ほか：複数 GPU 向けミドルウェアにおけるデータ管理手法の検討，研究報告計算機アーキテクチャ (ARC)，Vol. 2015, No. 29, pp. 1–6 (2015).