複数アクセラレータ向けデータ処理ミドルウェアの検討

鈴木 順[1,2,a)]　菅 真樹[1,b)]　林 佑樹[1,c)]　荒木 拓也[1,d)]　宮川 伸也[1,e)]　喜連川 優[2,f)]

概要：GPU 等のアクセラレータは HPC 用途だけでなく機械学習やデータベース処理を高速化するデバイスとして注目されている．我々は CPU とアクセラレータ等の I/O デバイスをインターコネクションで接続し，アプリケーションに要求される性能を満たす I/O デバイスを必要なときに必要なだけ CPU に割り当てる Resource Disaggregated Architecture (RDA) を提案している．本稿では RDA で割り当てた複数のアクセラレータ間で単一タスクを分散処理することで，アクセラレータの数の調整でシステムの処理性能を設定でき，またアクセラレータの数を動的に変化させた場合に Out-of-Core となってもユーザプログラムに透過に対応するミドルウェアについて検討する．要件検討の結果，従来のミドルウェアではアクセラレータ上のデータ配置を最適に行っていないこと，及びアクセラレータが特徴とする SIMD(Single Instruction Multiple Data) 型計算器に適した抽象化となっていないことからミドルウェアの処理抽象化のオーバヘッドが大きく非効率であるという考察に至った．また検討要件を踏まえて予備試作を行ったミドルウェアでは，ユーザプログラムの処理を示す DAG(Directed Acyclic Graph) を用いてアクセラレータ上のデータ配置を最適化する手法を検討した．評価の結果，比較対象とした個別の関数をアクセラレータにオフロードするライブラリ形式での実装に対し性能優位を確認した．

1. はじめに

近年，Nvidia 社の GPU や Intel 社の Xeon Phi を始めとするアクセラレータは，HPC への応用だけでなく機械学習 [1] やデータベース処理 [2] を高速化するビッグデータ向けのデバイスとして注目されている．個別のアクセラレータを用いてそれぞれのアプリケーションを高速化する手法は従来から検討されてきた．その一方，近年は GPU クラスタにおいて異なるホストの GPU 間で低遅延に直接データを通信する技術の導入や [3]，最大 8 枚のアクセラレータを挿入可能なサーバプラットフォームが商用化され，複数のアクセラレータを用いてアプリケーションを高速化する環境が整った．

我々はその中で，Resource Disaggregated Architecture (RDA)[4] と RDA を可能とするデバイスインターコネクションである ExpEther[5] を提案した．RDA では CPU とアクセラレータ等の I/O デバイスをインターコネクションを用いて接続し，アプリケーションから要求される性能を満たす数の I/O デバイスを必要なときに必要なだけ割り当てる．そして I/O デバイスを多く割り当てることで処理性能を向上させ，少なく割り当てることで処理性能が低下するように調整を行いたい．

RDA をアクセラレータに適用する場合，システムに割り当てるアクセラレータの数でシステムが提供する処理性能を定義できるようにしたい．従ってアプリケーションが保持する単一のタスクを複数のアクセラレータに分散させることで高速化を実現することがより重要になる．従来のアクセラレータの検討では，アプリケーション処理を構成する各タスクは単一のアクセラレータに割り当てられ，タスク自体の高速化はアクセラレータ内のマルチコアを用いた並列化によるものが多かった [6], [7], [8]．今後はアクセラレータで処理する画像や行列データの大型化と，処理に用いるアクセラレータの数に依存して処理性能を決めたいという要求から，単一タスクを複数のアクセラレータで分散して処理する重要性が高まると考えている．

一方，RDA によりシステムが保持するアクセラレータ数を自由に変更する環境が一般化すると，これまでよりシステムのアクセラレータ数がアプリケーションの要求に基づいて頻繁に変更される可能性がある．例えば機械学習の学習フェーズにおいてデータアナリストが行列演算を一時的に多数のアクセラレータを確保して高速化させる場合や，

[1] NEC グリーンプラットフォーム研究所
Green Platform Research Laboratories, NEC
[2] 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
[b)] kan@bq.jp.nec.com
[c)] y-hayashi@kv.jp.nec.com
[d)] t-araki@dc.jp.nec.com
[e)] s-miyakawa@ce.jp.nec.com
[f)] kitsure@tkl.iis.u-tokyo.ac.jp

衛星画像解析において突発事象が発生し，詳細な画像解析をアクセラレータを確保することで臨時に高速で行う場合等である．

本稿では，複数アクセラレータを用いた単一タスクの分散実行による高速化と，アクセラレータ数の動的再構成に適応するデータ処理ミドルウェアの要件を検討し，予備試作により有効性を確認する．アクセラレータ向けシステムソフトウェアの検討は，これまでにもミドルウェア，OS, データベース等の分野で行われてきた．本稿で対象とするミドルウェアは，単一タスクを複数のアクセラレータで分散する形態を取ることでアクセラレータ数により処理性能を調整する．また，アクセラレータ数の変更により実行する処理が生成するデータがアクセラレータメモリに収容できない Out-of-Core となっても対応することとする．これらの機能をアプリケーションプログラムの変更なく実現することでユーザのプログラミングを容易化し，またミドルウェアでは低オーバヘッドの抽象化 API を提供することで高いシステム性能の実現を目指す．

検討の結果，これらの要件を満たすシステムソフトウェアの従来例としてはアクセラレータを用いるためのライブラリと，具体的な処理をユーザカーネル関数として与えるミドルウェアに大別された．

アクセラレータのライブラリに関しては，ミドルウェアのような抽象化を行っていないためこれに関するオーバヘッドはないが，ユーザプログラムがライブラリの関数を呼び出す毎にアクセラレータにデータロードが発生し I/O コストが増大する課題がある．また，アクセラレータにデータを用意してからライブラリ関数を呼び出す別の形態では，アクセラレータでのデータのキャッシュや計算の実行順を最適化するためにユーザプログラムが複雑になるという課題があった．またライブラリは Out-of-Core にも対応しない．

一方ミドルウェアの代表例として Stuart らにより提案された複数 GPU 版の MapReduce がある [9]．MapReduce は計算を抽象化した Map および Reduce に対応する処理であれば汎用的にアプリケーションを実装可能である．また Out-of-Core にも対応できる．ただし MapReduce には 2 つの課題があると考えている．1 つめはアクセラレータに対する最適なデータ配置ができないということである．そのためにアクセラレータのメモリをデータキャッシュとして活用できなかったり，既にアクセラレータにロードされているデータを意識して計算順を最適化することができない．2 つめはアクセラレータが特徴とする SIMD(Single Instruction Multiple Data) 型計算に適した効率的な抽象化をミドルウェアが提供していないということである．MapReduce は中段の Shuffle 処理で計算コストが高いキーのソートを行い，Map 処理の結果を並べ直す．これに対し，アクセラレータ処理に関して主要な適用領域と考えられる行列や画像の計算では，各要素の位置はソートなしで決めることが原理的に可能である．特に画像の各ピクセルをソートする抽象化は現実的ではない．

また，本稿では検討したミドルウェアの要件に基づいて予備試作を行った．試作では Dryad [10] や Spark [11] で提案されたユーザの処理を示す DAG(Directed Acyclic Graph) を用いることで，アクセラレータ上の計算とデータ配置のスケジュールを最適化する手法を検討した．評価では，比較対象としたライブラリ形式でのアクセラレータへのオフロードの実装に対し，処理性能が向上することを確認した．

以下本稿では 2 節で背景としてアクセラレータによる処理の特徴や RDA を可能とするインターコネクション技術について述べる．3 節では複数のアクセラレータでタスクを分散実行するシステムについて処理性能の制約要因をハードウェアとシステムソフトウェアの観点からまとめる．続いて 4 節では今後の複数アクセラレータ向けミドルウェアに要求される要件について述べ，5 節でミドルウェアアーキテクチャの検討と予備試作による性能検証について述べ，最後に 6 節でまとめる．

表 1 Nvidia Tesla K40 緒元 [12]

最大倍精度浮動小数点性能	1.43TFLOPS
最大単精度浮動小数点性能	4.29TFLOPS
メモリバンド幅	288GB/s
メモリサイズ	12GB
CUDA コア	2880

2. アクセラレータと RDA インターコネクション

本節では今回検討するミドルウェアが対象とするアクセラレータの代表例である GPU についてその処理の特性を整理する．また，著者らが [5] で提案した，RDA を可能とする標準イーサネットを用いた I/O デバイスインターコネクション技術である ExpEther について述べる．

2.1 GPU

GPU は元々 PC やワークステーションの画像処理を行うプロセッサである．その高い計算資源をグラフィックレンダリングだけでなく汎用の計算処理に用いる技術を GPGPU(General-Purpose Computing on Graphics Processing Units) と呼ぶ．本稿の 5 節で述べるアクセラレータ向けミドルウェアの試作でも GPGPU を用いた．Nvidia 社の最新の GPU である Tesla K40 の性能を**表 1** にまとめる．2000 以上のコアを保持し，単精度浮動小数点性能が 4.29TFLOPS，倍精度浮動小数点性能が 1.43TFLOPS である．これは CPU の 10 倍のオーダーである．

GPU を利用するプログラマは，プログラム内で GPU

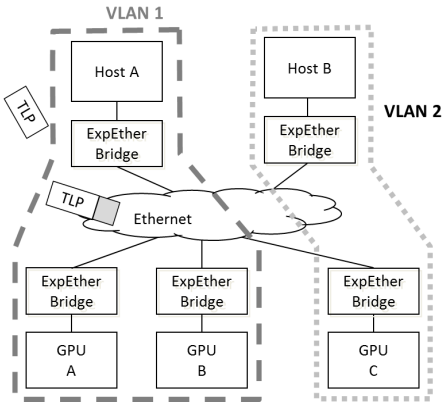

図 1 ExpEther の構成

に実行させるカーネル関数を定義し，GPU にスレッドを生成してカーネル関数を並列で実行させる．このとき生成されるスレッドは 32 スレッドを 1 単位とする Warp で管理される．各 Warp は GPU が保持する複数のマルチプロセッサのうちの 1 つで実行される．これら 32 個のスレッドは Single Instruction で駆動される SIMD 型の計算を行う．各 Warp はそれぞれを構成するスレッドが if 文等で分岐せず同じ命令を実行する場合，高速に処理を行うことが可能である．GPU に画像や行列を処理させる場合，各スレッドに別の要素の処理を割り当てることで，1 つの命令で分岐なく高速に並列処理を行う．

2.2 ExpEther

ExpEther は RDA を可能とするインターコネクションである．RDA を用いると，アプリケーションが要求する性能を満たすアクセラレータを必要なときに必要なだけ CPU に割り当てることができる．これにより例えば機械学習の学習フェーズにおいてデータアナリストが行列演算を一時的に多数のアクセラレータを確保して高速に行ったり，衛星画像解析において突発事象が発生し，詳細な画像解析をアクセラレータを確保することで臨時に高速で行ったりすることが可能となる．

ExpEther はホストとアクセラレータ等の I/O デバイスを標準イーサネットを用いて接続する．**図 1** に ExpEther の構成を示す．ホストと I/O デバイスをイーサネットに接続するデバイスが ExpEther ブリッジであり，ホスト側は I/O スロットに挿入される PCIe カード，I/O デバイス側は複数の I/O デバイスを収容する I/O Box の基盤上にチップとして実装される．ExpEther ブリッジはホストと I/O デバイスの間で PCIe(PCI Express) パケット (TLP: Transaction Layer Packet) のトンネリングを行う．

ExpEther ではホスト側と I/O デバイス側のブリッジの VLAN 番号を設定することで，同じ VLAN 番号がグループ化され，I/O デバイスを割り当てるホストを選択することができる．また VLAN 番号の変更により，I/O デバイスの割り当てを別ホストに変更することができる．各ホストでは I/O デバイスの割り当ては PCIe デバイスのホットプラグイベントとして処理される．

ExpEther ではこの他にもアーキテクチャ上のいくつかの特徴がある．まず ExpEther ブリッジは PCIe プロトコルを処理することでホストには PCI-PCI ブリッジとして認識される．これによりイーサネットはホストに透過となり，各ホストの PCIe ツリーがイーサネットに拡張されるよう疑似される．そして PCIe に準拠するアクセラレータを変更することなく ExpEther を用いてイーサネットで接続することが可能となる．また，ホスト側と I/O デバイス側の ExpEther ブリッジの間で行う TLP のトンネリングでは，ブリッジ間で通信遅延をモニタし，遅延が増大しないように TLP をカプセル化するイーサネットフレームの送信レートを制御する．これにより低遅延の PCIe コネクションをイーサネットで提供する．

3. 性能制約要因

本節では複数のアクセラレータを用いてタスクを分散実行し，また分散実行するアクセラレータの数を自由に変更できるシステムについて処理性能の制約要因をハードウェアとシステムソフトウェアの観点からまとめる．

3.1 ハードウェアに関する制約

3.1.1 I/O バス帯域

近年は PC や組み込み用途向けに CPU と一体化された GPU が製品化されている一方，高い性能を保持するアクセラレータは I/O デバイスとしての実装であり CPU と I/O バスを介して接続される．従ってアクセラレータで処理を行うためにはホストのメインメモリ上のデータを DMA(Direct Memory Access) でアクセラレータのメモリに送信し，アクセラレータで計算を行い，その計算結果を DMA でホストのメインメモリに再度送信する必要がある．このためアクセラレータを用いてアプリケーションの処理を高速化する場合，I/O バスによるデータ通信オーバヘッド以上に計算が高速化される必要がある．プログラムの良し悪しにも依存するが，計算の負荷はアプリケーション自体に依存するため，計算負荷が小さいアプリケーションではアクセラレータで処理を行うことでシステム性能が低下する．また複数のアクセラレータを用いる場合，I/O バスである PCIe バスのトポロジーがツリーであるため，配下のアクセラレータに一斉にデータを送信すると，ツリーの上流で共有しているバスの帯域がボトルネックとなる場合がある．このような場合，各アクセラレータにシリアライズしてデータを送信するコストと同等のオーバヘッドが発生する．

3.1.2 アクセラレータ間のデータ通信遅延

複数のアクセラレータを用いて並列処理を行う場合，次

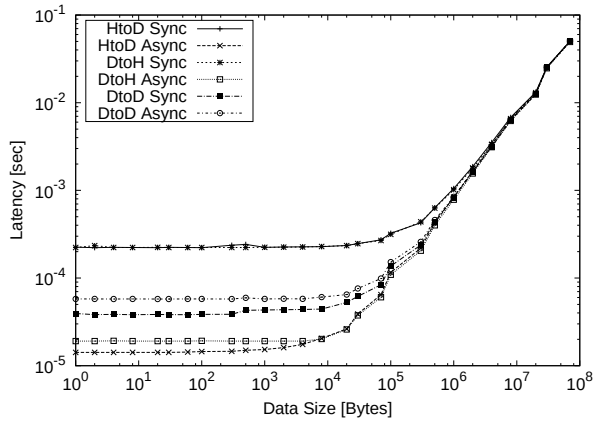

図 2　GPU 間のデータ通信遅延 [13]．H: ホスト，D: デバイス (GPU)．GPU 間の通信には GPU Direct を用いた．

の処理のステップに進むためのバリアとなるアクセラレータ間のデータ通信のオーバヘッドはアクセラレータ数に対する処理性能のスケーラビリティの制約要因となる．**図 2** は著者らが [13] において測定した 2 つの GPU 間でのデータ通信遅延である．データ通信遅延はプロトコルを処理する処理遅延と，通信するデータ量に依存する伝送遅延に大別される．図 2 は ExpEther によりイーサネットを用いて GPU を接続した場合だが，ホストの I/O スロットに GPU を直接挿入した場合でも処理遅延が支配的なグラフ左側の領域では相違が 2 倍以内だった．図 2 が示すように処理遅延が支配的な 1KB 以下のデータ通信では，通信遅延は同期モードで 40us，非同期モードで 60us 程度である．複数のアクセラレータを用いる場合，アクセラレータ間の通信を必要とする 2 つのバリアの間の処理量が通信オーバヘッドより大きくない場合アクセラレータ数に対する処理性能のスケーラビリティの制限要因となる．

3.1.3　アクセラレータメモリ容量

アクセラレータのメモリ容量は現行の Nvidia 社の Tesla GPU で 5GB-12GB である [12]．また複数のアクセラレータを使用する場合，全体のメモリ容量はこの整数倍となる．実行する処理の使用メモリ量がアクセラレータのメモリ容量を超えない場合，使用しないメモリは今後使用するデータのキャッシュとして用いることができる．キャッシュはアクセラレータの計算では I/O バス帯域がボトルネックとなるため有効である．一方，処理の使用メモリ量がアクセラレータのメモリ容量を超える場合，実行する処理内で複数回使用されるデータに関しては一度のアクセラレータへのロードでできるだけ多くの処理を行う必要がある．このスケジューリングがない場合，データは処理に必要とされる度に I/O バスを介してロードされ，I/O オーバヘッドが増大する．またアクセラレータをシステムに柔軟に参加，離脱させる場合，全体のメモリ容量が変化するためその点にも対応する必要がある．

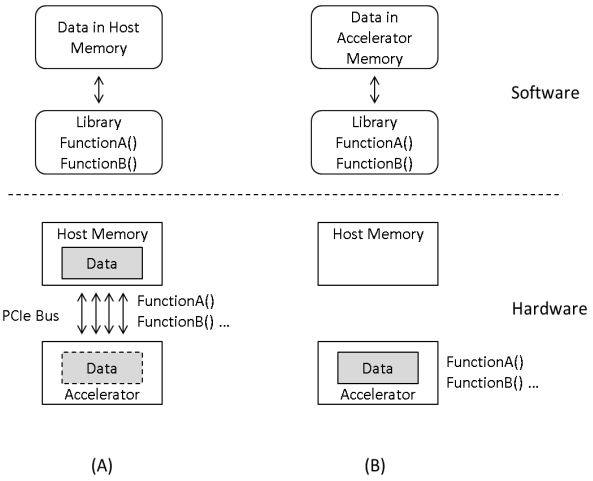

図 3　アクセラレータを利用するためのライブラリアーキテクチャ．(A) ライブラリにホストメモリ上のデータを渡す形式．(B) ライブラリにアクセラレータ上のデータを渡す形式．

3.2　システムソフトウェアに関する制約

アクセラレータを使用するためのシステムソフトウェアはライブラリによる実装とユーザがカーネル関数を与えるミドルウェアによる実装に大別される．以下では 2 つについて性能制約要因をまとめる．

3.2.1　ライブラリによる実装

アクセラレータを利用するプログラムを作成する場合，行列計算や画像処理等，所望の処理を実装したライブラリを用いる方法がある．ライブラリの実装形態には**図 3** に示す 2 つの種類がある．形式 (A) はホストのメモリ上のデータをライブラリに渡すと，ライブラリがデータをアクセラレータに移動し，計算を行い，結果をホストのメモリ上に格納する方式である．この場合，複数アクセラレータへのデータの分散も含めてライブラリが実施するが，ユーザプログラムがライブラリの複数の機能を利用する場合，ライブラリの呼び出し毎にメインメモリとアクセラレータ間でデータ移動が発生する．この場合アクセラレータを用いることによる高速化効果が低下するだけでなく，処理全体に占める I/O コストの割合が増大し，複数のアクセラレータを用いて計算コストを低下させた場合でも全体の処理コストの削減が制限される．

一方形式 (B) はアクセラレータのメモリ上にライブラリではなくユーザプログラムがデータを用意し，そのデータをライブラリに渡す手法である．この場合，アクセラレータのメモリ上のデータ管理をプログラマが行う必要があり，データのキャッシュ，アクセラレータ数の変動に対応する機能を実装するためユーザプログラムの処理が複雑になる．

3.2.2　ミドルウェアによる実装

アクセラレータ向けプログラムを作成するための別の方法として，複数のアクセラレータを仮想化するミドルウェアを利用し，所望の処理を行うユーザカーネル関数を作成

してミドルウェアに渡すことで処理を実行する手法がある．本稿の検討もミドルウェアを対象としている．ミドルウェアの利点として，ライブラリに実装される程度に一般的ではない特殊な処理も実装できる点や，異なる処理を組み合わせることができる点がある．

単一タスクを複数のアクセラレータに分散して並列実行し，アクセラレータの数を調整することで処理性能を調整し，Out-of-Core にも対応する代表的なミドルウェアとして MapReduce がある．GPU 向け MapReduce は J. A. Stuart らによって提案された [9]．

アクセラレータ向け MapReduce の課題として，MapReduce はアクセラレータが特徴とする SIMD 型計算に適した抽象化ではないということがある．MapReduce は元々データの分割を単一プロセスが処理することを想定しており，データの分割内の処理は順序のない要素指向であり，各分割の処理の関係も Out-of-Order で良い．そのため MapReduce の中段の Shuffle 処理では，Out-of-Order で行った処理結果を後段の Reduce 処理で結合するために計算コストが高いキーのソートを行う．これに対し，アクセラレータ処理に関して主要な適用領域と考えられる行列や画像の計算では，各要素の位置はソートなしで決めることが原理的に可能である．特に画像の各ピクセルをソートする抽象化は現実的ではない．また，MapReduce のようにアプリケーションに依存せず共通のレイヤでミドルウェアを作成する手法がある一方で，計算の典型例毎にミドルウェアの対応を分離する手法もある．一例は UC Berkeley の Aspire Lab であり，実行する処理の形式を行列計算やグラフ処理，MapReduce 等の計算のタイプにマッピングする [14]．ただしこの場合，ミドルウェアの構成が複雑化するという課題がある．

4. 複数アクセラレータ向けミドルウェアへの要求事項

本節では単一タスクを複数のアクセラレータに分散して実行するミドルウェアに要求される機能を議論する．

4.1 アクセラレータ数の柔軟な対応

本稿で対象とするシステムは RDA を用いてアクセラレータ数を柔軟に変更することを想定している．このためミドルウェアにはアクセラレータ数に関わらず同一プログラムを動作させることが求められる．アクセラレータは，システムの動作中でも必要に応じて追加，削減を行うことができ，システム動作に影響を与えずに性能をアクセラレータ数に応じて柔軟に変更できることが望ましい．このためミドルウェアがユーザに提供する API は，処理を実行するアクセラレータ数を仮想化する必要がある．

4.2 Out-of-Core

近年のアクセラレータのメモリ容量は 5GB-10GB 程度である．今後，ビッグデータを背景にアクセラレータのメモリ容量を超えるメディア処理や行列/ベクトル処理が必要になるだろう．また，複数のアクセラレータを用いて実行した処理において，アクセラレータ数を削減した結果，処理中に生成されるデータがアクセラレータのメモリに収容できなくなる場合も考えられる．以上からミドルウェアは処理の過程で生成されるデータがアクセラレータメモリの容量を超える場合でも，ユーザプログラムを変更せず透過的に対応できる必要がある．

4.3 処理を高速化するデータ管理

アクセラレータを用いる処理において高い性能を実現するためには，それぞれのアクセラレータに I/O に関するブロッキングなく常に計算を行わせることが重要である．そのためにはデータ管理の工夫により，次に行う計算の入力データが，計算の開始時までにアクセラレータにロードされていることが必要である．

アクセラレータメモリ上のデータ管理では，メモリリソースに余裕がある場合，複数回使用されるデータをキャッシュすることでアクセラレータへの I/O オーバヘッドを削減することができる．また，メモリリソースに余裕がない場合でも，処理の過程で同じデータを用いて複数回の計算が行われる場合，アクセラレータへの一度のデータロードで多くの計算をまとめて行うようスケジュールすることで I/O オーバヘッドを削減できる．また次の計算の入力データのプリフェッチも有効である．

一方アクセラレータ間のデータ通信もアクセラレータ間でのデータ分割を工夫する方法や，アクセラレータ間でのデータ交換と計算をオーバラップさせる方法が I/O オーバヘッドを削減する手法として有効である．

4.4 SIMD に適した低オーバヘッドの処理抽象化

アクセラレータの適用領域と考えられる行列やメディア処理では，アクセラレータの各スレッドが計算を担当する要素 (行列の要素や画像のピクセル) の位置が予めわかっている場合が多い．ミドルウェアはこの制約を利用し，キーのソートが必要ないときはソートを行わない抽象化を提供する必要がある．また，SIMD を用いた処理ではスレッドを階層的に用いる場合が多い．例えば各要素の和を取る場合である．ミドルウェアはこれらの SIMD 特性に適したオーバヘッドが低い抽象化 API を提供する必要がある．

4.5 ヘテロリソース対応

アクセラレータを保持するプラットフォームでは，計算手段として少なくとも CPU とアクセラレータという 2 種のデバイスを保持する．またストレージデバイスの観点で

も，近年の SSD や PCM(Phase Change Random Access Memory) の発展により，RAM や HDD を含めた 3 種以上のデバイスを保持するのも一般である．これらのデバイスには個別の特性があり，例えば計算デバイスでは並列処理はアクセラレータが強いが，並列性が効かない処理は CPU の方が向いている．また，両者の性能に大きな差がない場合，計算開始時点の入力データのローカリティが重要である．今後のミドルウェアは，これらの複数の計算デバイスやストレージデバイスをユーザに抽象化し，ミドルウェアで処理を適切なデバイスにマッピングする必要がある．

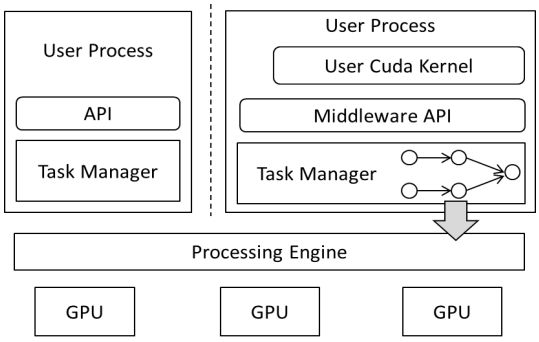

図 4 検討したミドルウェアアーキテクチャ

4.6 適切な言語のレベル

ミドルウェアを利用するユーザプログラマに対し，どの階層の言語の記述を要求するかが重要な選択である．例えば Nvidia 社が提供する GPU 開発環境である Cuda を用いる場合，プログラマが処理の最適化を行う自由度は向上するが，ミドルウェアは GPU 専用となる．これに対し同じ低レベルの抽象度でも OpenCL 等のアクセラレータ間で標準化された言語を用いると，異種のアクセラレータに対しミドルウェアを共通化できる可能性がある．Cuda や OpenCL を用いるその他の利点は，ライブラリ関数や，既に開発されたアクセラレータ向けコードを再利用したり，一部を改造してミドルウェアに与えるたりすることが可能であることである．

また，アクセラレータに詳しくないアプリケーションに近いプログラマ向けには，Brown らが提案しているアプリケーション領域の処理に特化した抽象度の高い DSL(Domain Spacific Language) を定義し，プログラミング負荷を減らす手法も考えられる．DSL では処理を行う CPU やアクセラレータのコードをコンパイラで生成する [8]．

以上の議論から，階層が低い Cuda や OpenCL でも，階層が高い DSL でもミドルウェアが想定するターゲットに合致する言語を選択する必要がある．

5. ミドルウェアのアーキテクチャ検討と予備試作

本節では 4 節で述べた複数アクセラレータ向けミドルウェアの要件を満たすアーキテクチャの検討とその検討に基づいて行った試作について述べる．

5.1 アーキテクチャ検討

今回は 4 節に整理した要件の内，特に (1) アクセラレータ数の柔軟な対応，(2)Out-of-Core，(3) 処理を高速化するデータ管理に対応できるアーキテクチャという視点で検討を行った．(4)SIMD に適した低オーバヘッドでの抽象化に関しては，今回は API は仮決めとし，今後継続して検討する予定である．また，ミドルウェアが対象とするアクセラレータとして現在一般に多用されている Nvidia 社の GPU を選択した．

検討したミドルウェアのアーキテクチャを図 **4** に示す．図 4 に示す全体のソフトウェアスタックは，1 つ以上の GPU を保持する単一のホストで動作する．なお，処理を行う GPU の数は ExpEther を例とする RD インターコネクションで接続される想定であり，アプリケーションの要求により柔軟に参加や離脱が行われる．

プログラムはユーザプロセスと処理エンジン (Processing Engine) の 2 つのプロセスから構成される．ユーザプロセスはユーザプログラマが作成する Cuda カーネル，ミドルウェアがプログラマに提供する API，ユーザプログラムの処理を示す DAG を作成するタスクマネージャから成る．

Cuda カーネルはユーザプログラマが作成し，GPU に対し行わせる処理を示す．Cuda カーネルをミドルウェアの API の引数として与えることで，ミドルウェアが指定された処理を行う．

ミドルウェアの API は MapReduce の例では Map 関数及び Reduce 関数に対応するが，アクセラレータの特徴である SIMD 型の計算に適し，アプリケーションとして想定される行列計算やメディア処理に対してオーバヘッドが低く性能が高い抽象化を行う定義を今後検討する予定である．次節で述べる今回の試作では，API を [11] を参考に Transformation と Action の 2 種類に分類した．Transformation の呼び出しではタスクマネージャにより DAG の作成が行われるが，実際の処理は実行されない．Action が呼ばれるとそれまで作成した DAG の処理が開始される．

タスクマネージャはユーザプログラムの処理を示す DAG を作成する．今回の試作では DAG のノードをデータ，エッジをミドルウェアの API とそれに与えられた Cuda カーネルが示す処理とした．ノードが示すデータは，それを構成するデータの分割を複数の GPU に分散して格納する．それにより DAG のエッジに対応する処理を GPU 間で並列に行う．また DAG の起点となるデータはファイルシステムに格納されたデータである．ミドルウェア API が提供するデータクラスのコンストラクタにファイルのパスを与えることで，ミドルウェアによる DAG 処理の実行時にデータがファイルシステムからリードされ，GPU メモリ上に

ロードされる．

実行エンジンはソケットを経由してタスクマネージャが生成した DAG を受信し，ホストに割り当てられている GPU を用いて DAG で指定された処理を行う．実行エンジンはバックグラウンドで動作するプロセスであり，全ての GPU のメモリリソースとメモリ上のデータを管理する．実行エンジンはプロセスの開始時に GPU のメモリリソースを確保し，DAG の実行ではエンジンが管理するメモリを DAG のノード (データ) に割り当てることで処理を実行する．GPU メモリに DAG の各エッジの処理の入力データと出力データの総和が収容できる場合，GPU メモリからデータのスワップアウトを行うことなく DAG の一連の処理を行うことが可能である．また GPU メモリに DAG の各エッジの処理で使用する容量以外に余裕がある場合，それまで行われた各ノードのデータをキャッシュとして保持する．GPU メモリがキャッシュを保持する場合，データを GPU にロードする I/O コストが削減できる．これは複数の DAG に跨って使用されるデータに対しても有効である．

このようなアーキテクチャを用いることにより，DAG の各エッジの処理をホストに割り当てられた複数の GPU に分割することで GPU 数に対し処理性能を向上できるシステムを目指す．GPU はミドルウェアに対し柔軟に追加と削減が可能であり，ユーザプログラムには影響を与えない設計とする．また実行エンジンは GPU のメモリ容量の変化に適応してデータとメモリリソースの管理を行う．メモリ容量に余裕がある場合，DAG 上のデータをキャッシュすることで，キャッシュされたデータが再び使用される場合の I/O コストを削減する．また，GPU 数が少ない場合や容量の大きいデータを扱う場合に全ての処理データがメモリに収容できない場合，ユーザプログラムに対し透過に Out-of-Core 処理を行う．具体的には DAG の各データは分割されて GPU で処理が行われるため，そのデータ分割を順次 GPU にスワップし処理を行う．この Out-of-Core 処理では，DAG 内で単一のデータのみに依存する処理が直列に連続する場合，データ分割を一度ロードすると処理をまとめて行う等の高速化を実施する．また，データの分割に対する処理を GPU で行っている間に，別のデータの分割を GPU へロードし，計算と I/O をオーバラップさせることで I/O コストを削減する．

5.2 予備試作と評価

前節で述べたアーキテクチャの実現性を検討するため，ミドルウェアの予備試作を行った．用いたアプリケーションは衛星画像解析であり，衛星画像に対しダウンサンプリングを行い，処理した画像をタイルに分割し，特徴量を求めた．特徴量は単純例としてヒストグラムを用いた．このような画像処理と特徴量計算の組み合わせは画像解析の分野で頻繁に用いられる [15]．今回のアプリケーションで生成される DAG は 3 ノードの単純な構造であり，衛星の画像ノードに対し，ダウンサンプリング後の画像ノードが生成され，続いて特徴量のノードが生成される．仮設定したミドルウェアの API は，ユーザプログラマが与えたカーネル関数を実行する各スレッドが，そのインデックスにより処理を担当する画像のピクセル位置が判定できるものとした．用いた衛星画像のサイズは約 800MB だった．この処理を，今回試作したミドルウェアで実行した場合と，図 3 (A) に示したライブラリの構成で実行した場合とで性能の比較を行った．ライブラリの構成ではダウンサンプリング関数と特徴量計算関数が個別に GPU にデータを送信するため I/O コストが増大する．性能比較は GPU 数に柔軟に対応できることを確認するため，ユーザプログラムを変更せずに GPU 数が 1 個と 2 個の場合とで行った．またホストと GPU は ExpEther を用いて接続した．**図 5** に実験結果を示す．

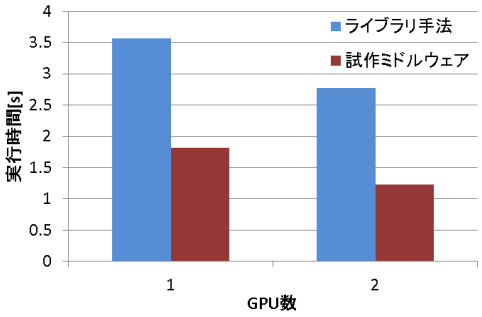

図 5 予備試作による評価

試作したミドルウェアではライブラリによる手法より 1GPU で 96%，2GPU で 126%性能が改善した．また GPU を 1 個から 2 個に変化させた場合の性能の伸び率は試作ミドルウェアで 48%，ライブラリ手法で 29%だった．これらの性能差はライブラリ手法では I/O コストが増大する一方，試作ミドルウェアではユーザプログラムの DAG に関するデータが GPU メモリに収容されるため，一度データがロードされると DAG の完了まで全ての処理が GPU 上で行われるためである．

今回の試作により DAG を用いた GPU 上のデータ配置の最適化を行い，DAG のノードに対応するデータを複数の GPU で分散実行することで (1) アクセラレータ数の柔軟な対応，(3) 処理を高速化するデータ管理について実現できることを示し，ライブラリ形式の実装より性能が優位であることも示した．(2)Out-of-Core については今後対応する予定である．また (4)SIMD に適した抽象化に関しては，今回は仮の設計として各スレッドが処理する画素の位置がわかる仕様を想定したが，GPU の適応領域と考えられる行列やメディア処理に適した定義を今後検討する．なお試作したミドルウェアの 1GPU から 2GPU への性能向

上が 48%であるが，今回は GPU での計算と I/O をオーバラップする等の最適化を省略しており，今回の結果はさらに向上可能である．

6. まとめ

本稿では単一タスクを複数のアクセラレータで分散処理することでアクセラレータの数の調整でシステムの処理性能を設定でき，またアクセラレータの数を動的に変化させた場合に Out-of-Core となってもユーザプログラムに透過に対応するミドルウェアを検討した．検討では従来のアクセラレータを利用するライブラリやミドルウェアではアクセラレータ上のデータ配置を最適に行っていないことを示した．また，ミドルウェアでは SIMD 型計算器に効率的な低オーバヘッドの抽象化を行っていないことも示した．これに対し検討要件を踏まえて予備試作を行ったミドルウェアでは，ユーザプログラムの処理を示す DAG を用いてアクセラレータ上のデータ配置を最適化する手法を検討した．評価では，比較対象とした個別の関数をアクセラレータにオフロードするライブラリ形態での実装に対し最大 126% 性能が改善することを確認した．今後の予定として SIMD に適した低オーバヘッドの処理を抽象化するミドルウェアの API を検討する．

参考文献

[1] Canny, J. and Zhao, H.: Big Data Analytics with Small Footprint: Squaring the Cloud, *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pp. 95–103 (2013).

[2] Wang, K., Zhang, K., Yuan, Y., Ma, S., Lee, R., Ding, X. and Zhang, X.: Concurrent Analytical Query Processing with GPUs, *Proc. 40th Int. Conf. on Very Large Data Bases (VLDB)* (2014).

[3] Mellanox: Mellanox OFED GPUDirect RDMA, Mellanox Technologies (online), available from ⟨http://www.mellanox.com/page/products_dyn?product_family=116&mtag=gpudirect⟩ (accessed 2014-10-24).

[4] 菅真樹， 鈴木順， 林佑樹，吉川隆士，宮川伸也：CPU/IO 分離アーキテクチャの特性を活かしたミドルウェアの検討，SWoPP 新潟 2014, CPSY-5 (16) (2014).

[5] Suzuki, J., Hidaka, Y., Higuchi, J., Yoshikawa, T. and Iwata, A.: ExpressEther - Ethernet-Based Virtualization Technology for Reconfigurable Hahardware Platform, *Proc. IEEE Symp. on High-Performance Interconnects*, pp. 45–51 (2006).

[6] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B. and Witchel, E.: PTask: Operating System Abstractions To Manage GPUs as Compute Devices, *Proc. Twenty-Third ACM Symp. on Operating Systems Principles (SOSP)*, pp. 233–248 (2011).

[7] Pienaar, J. A., Raghunathan, A. and Chakradhar, S.: MDR: Performance Model Driven Runtime for Heterogeneous Parallel Platforms, *Proc. Int. Conf. on Supercomputing (ICS'11)*, pp. 225–234 (2011).

[8] Brown, K. J., Sujeeth, A. K., Lee, H. J., Rompf, T., Chafi, H., Odersky, M. and Olukotun, K.: A Heterogeneous Parallel Framework for Domain-Specific Languages, *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pp. 89–100 (2011).

[9] Stuart, J. A. and Owens, J. D.: Multi-GPU MapReduce on GPU Clusters, *Proc. IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1068–1079 (2011).

[10] Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks, *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pp. 59–72 (2007).

[11] 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, *Proc. 9th USENIX conf. on Networked Systems Design and Implementation (NSDI)* (2012).

[12] Nvidia: サーバ用 TESLA GPU アクセラレータ，Nvidia（オンライン），入手先 ⟨http://www.nvidia.co.jp/object/tesla-servers-jp.html⟩（参照 2014-10-22）.

[13] Nomura, S., Mitsuishi, T., Suzuki, J., Hayashi, Y., Kan, M. and Amano, H.: Performance analysis of the multi-GPU system with ExpEther, *Proc. Fifth Int. Symp. on Highly-Efficient Accelerators and Reconfigurable Technologies* (2014).

[14] Alon, E., Asanovic, K., Bachrach, J., Demmel, J., Fox, A., Keutzer, K., Nikolic, B., Patterson, D., Sen, K. and Wawrzynek, J.: ASPIRE, Aspire Lab., UC Berkeley (online), available from ⟨https://aspire.eecs.berkeley.edu/wp-content/uploads/2013/03/ASPIRE-20130214-BEARS1.pdf⟩ (accessed 2014-10-23).

[15] Teodoro, G., Hartley, T. D. R., Catalyurek, U. and Ferreira, R.: Run-time optimizations for replicated dataflows on heterogeneous environments, *Proc. ACM Int. Symp. on High Performance Distributed Computing*, pp. 13–24 (2010).