博士論文

I/O デバイスの分離アーキテクチャに関する研究

鈴 木　順

目次

図目次

表目次

第 1 章

序論

1.1 はじめに

ICT (Information and Communication Technology) は安全・安心・公平・効率的な社会を実現する基盤技術として必要不可欠な存在となっている．それに伴い ICT 産業への需要も継続的に増加している．2011 年〜2016 年における世界の ICT 投資規模は年間 5.4% で成長している [79].

ここで ICT 技術の投資の内訳を見ると，ICT サービスへの投資を増加する企業が 42% ある一方，ハードウェアへの投資を増やす企業は 30% と相対的に低い [80]．これは ICT への関心が，ハードウェアというインフラストラクチャーから ICT で実現する社会の価値に移っているためだと考えられる．その動きを裏付けるように，ICT 設備の保有者から利用者への移行を可能にするクラウドサービスの需要が益々増加している [30]．また，ICT 技術の発展により，ICT 技術を適用する新たな応用分野の開拓も継続している．例えばドイツが先行するインダストリー 4.0 では，ICT を製造業に適用することで，効率的な生産や流通，個別の需要に応じた大量生産を行うマスカスタマイゼーションが実現される．

ここで ICT の基盤である計算機に目を向けると，計算機は Microsoft 社の Windows OS (Operating System) と Intel 社の CPU (Central Processing Unit) の組み合わせにより長年コモディティ化が進んできた．Windows と Intel CPU を調達して製造された計算機は，計算機を提供するメーカーに関わらず価値が同質となった．これは計算機産業で水平分業が進み，CPU 層と OS 層の寡占化が進んだことで，計算機メーカーは単に標準品を組み立てる事業者となったことを示す．その結果計算機の利用者には計算機間で実現される価値が同質となり，近年の利用者が計算機の保有者からクラウドの利用者へ移行す

る動きを加速させた．

一方，計算機の利用を支えるクラウドデータセンターでは，計算機の高密度化が進行している [8]．これは限られた面積のデータセンターに効率良く計算機を収容し，サービスを低コストで提供するためである．高密度化された計算機のアーキテクチャは，CPU の他，搭載する I/O デバイスを必要最小限に抑えたコモディティサーバである．

しかし，近年 AI (Artificial Intelligence) やビッグデータ処理の需要の増大により，データを GPU (Graphics Processing Unit) や PCIe (PCI Express) SSD (Solid State Drive) 等の I/O (Input/Output) デバイス型のアクセラレータで処理する機会が増加した．このような需要にデータセンター向けの高密度サーバで応える場合，現在の計算機のアーキテクチャでは非効率になる．それは 1 台の計算機に搭載できる I/O デバイスの数が制限されるからである．そのため 1 台の計算機が保持する数以上のアクセラレータを用いる場合，アクセラレータを確保するために追加の計算機を占有する必要がある．特に高密度サーバの場合，I/O スロットの数が少なくアクセラレータ数も限定される．ここで N 個のアクセラレータを用いるために必要な計算機の数は，各計算機の I/O スロット数を M とすると

$$\lceil \frac{N}{M} \rceil \tag{1.1}$$

で表される．また逆に，各計算機の I/O スロットにアクセラレータを挿入し，アクセラレータを占有させたが，個別の計算機のアクセラレータの使用率が低い場合もハードウェアの運用が非効率となる．

上記はコモディティアーキテクチャの計算機をデータセンターに集約してサービスを提供する場合の議論である．ここで，従来の計算機は様々な用途に汎用に適用するためにアーキテクチャがコモディティ化した．しかし，データセンターの場合，サービスを提供する API (Application Programming Interface) より下は利用者には透過である．従って，データセンター内で運用する計算機のアーキテクチャは，より効率的な計算機の運用を実現できるなら従来のコモディティなアーキテクチャを踏襲する必要がなくなっている．むしろ提供するサービスの要求に合わせて構成を柔軟に目的に特化させ効率的な運用を実現する新しいアーキテクチャが必要である．

また，インダストリー 4.0 への ICT 技術の導入といった新しい応用を考える場合，従来のコモディティの計算機をインダストリー 4.0 に適用するのは耐環境性や製造業の既存機器との親和性の観点で望ましくない．製造業では従来から専用の計算機や工作機械として I/O デバイスを使用している．そのため，それらの環境からの移行が容易な新しいアーキテクチャを実現し，工場の工作機械同士の協調やセンサーデータの集約と AI による解析

を実現する必要がある．

本研究の目的は 2 つである．1 つ目は I/O デバイスを計算機のホストから分離し，ホストと I/O デバイスを必要に応じて柔軟に組み合わせることで目的に特化した効率的なハードウェアの構成を実現することである．また，製造業等における既存の機器にホストと I/O デバイスを分離するアーキテクチャを適用し，I/O デバイスをホストに柔軟に割り当てることで，既存の機器を変更せずにホストの I/O デバイスの拡張を実現することである．また 2 つ目の目的は，分離した I/O デバイスとしてアクセラレータを用いた場合，割り当てたアクセラレータを効率的に使用して高い計算性能を実現し，用いるアクセラレータ数に応じた計算性能の向上を実現することである．

1 つ目の I/O デバイスをホストから分離する研究では，ネットワークによる I/O デバイスの分離方式に取り組む．具体的にはホストの I/O バスをネットワークに拡張し，ホストとネットワークで接続した I/O デバイスを従来のホストの I/O スロットに挿入されたデバイスとして仮想化する技術の研究を行う．これにより，従来の計算機の I/O スロット数の制限なくネットワークによる接続で所望の I/O デバイスを仮想的に増設できる．また個別のホストの I/O デバイスの稼働率が低い場合，ネットワークで接続した I/O デバイスを複数のホストで共有することで，効率が良いハードウェアリソースの運用を実現できる．

本研究ではホストと I/O デバイスを接続するネットワークに標準のイーサネットを使用し，計算機の I/O バスである PCIe (PCI Express) バスを拡張する．PCIe は計算機内の I/O バスとして広く用いられている標準規格である．またイーサネットは計算機のホスト間を接続するネットワークとして多用されている．本研究では，I/O デバイスをイーサネットを用いて接続するが，計算機のソフトウェアにはイーサネットを介した PCIe バスの拡張を透過とし，イーサネットで接続した I/O デバイスを従来の OS やデバイスドライバから使用できるようにする．

2 つ目のアクセラレータを効率的に用いる研究では，計算アクセラレータとして GPU を用いる．そして分離した複数の計算アクセラレータを用いた計算の最適化方式に取り組む．具体的には GPU の演算リソースを効率的に利用し，高い計算性能を実現するミドルウェアの研究を行う．GPU を用いる多くの計算ではそのような特性を得ることが難しい．なぜなら処理するデータをホストのメインメモリと GPU の間で移動させる通信オーバヘッドが性能ボトルネックとなるからである．本研究のミドルウェアではこのような通信オーバヘッドを最小化し，効率が良い計算を自動で実現する技術に取り組む．

近年 GPU は計算アクセラレータとして最も多用されている．GPU は CPU と比較して演算性能が 5 倍以上，ローカルメモリの帯域が 8 倍以上であり，CPU の計算を GPU

にオフロードすることで大きな計算性能の向上が期待できる．しかし，GPU のメモリ容量はホストのメインメモリの 10 分の 1 以下であるため，GPU が多用される大容量のデータ処理では Out-of-Core 処理となる．Out-of-Core 処理では GPU のメモリ容量以上のデータを分割し，GPU メモリ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．このとき，データスワップの通信で用いる I/O バスの帯域が GPU メモリの帯域より 1 桁小さく計算の性能ボトルネックになる．従って Out-of-Core 処理では，データスワップを最小化するようにタスクの実行順を制御する必要がある．しかし，現状では GPU で実行する演算や GPU メモリが保持するデータの管理は GPU の利用者であるアプリケーションプログラマに任されている．従ってプログラマは GPU メモリのデータの入れ替えを最小化するタスクの実行順を考える必要があり，大きな負担である．本研究で取り組むミドルウェアは，これらの管理からプログラマを解放し，システムで自動で実現する．

1.2 本論文の構成

本論文の構成は以下の通りである．

第二章では本研究の関連研究をまとめる．まず本研究のネットワークによる I/O デバイスの分離方式と関連する研究を述べる．関連研究には I/O デバイスのインターコネクションに関する研究を始め，I/O デバイスのリソースを複数のホストから共有する研究がある．また本研究で採用しているイーサネットで高い I/O インターコネクションの帯域を実現するためのハードウェアのパケット処理技術や，複数のパケットを集約して同一パケットにカプセル化する技術の研究がある．また本研究の分離した複数の計算アクセラレータを用いた計算の最適化方式に関連する研究も述べる．関連研究には，分散ヘテロ計算環境をアプリケーションに利用しやすくするミドルウェアの研究がある．その中でも本研究が対象とするデータパラレル処理や GPU に関する研究に着目して述べる．また，複数の計算リソースを用いて効率的に計算を行うためのミドルウェアの研究について述べる．

第三章ではネットワークによる I/O デバイスの分離方式について述べる．本技術を ExpEther と呼んでいる．ExpEther は PCIe over Ethernet の通信をハードウェアブリッジで実現する．ExpEther により，PCIe に準拠する I/O デバイスをホストから分離し，イーサネットの接続で必要に応じて I/O デバイスを所望のホストに柔軟に割り当てることができる．またハードウェアブリッジによる仮想化機構により，ホストに対してイーサネットが透過となり，従来のソフトウェアスタックやイーサネット，I/O デバイス

への変更を行わずにホスト内部の PCIe バスをイーサネットに拡張することができる．

第四章では本研究で作成した ExpEther のプロトタイプと，プロトタイプを用いた評価結果について述べる．評価ではプロトタイプを用いて汎用のホストと I/O デバイスをイーサネットで接続し，それらの I/O デバイスに対する性能評価を行う．これにより，ExpEther は小さいオーバヘッドでホストの I/O デバイスを拡張可能であることを示す．また，ExpEther が一般のデータセンターで用いられている汎用 I/O デバイスに広く適用可能であることも示す．さらに，複数の PCIe パケットを集約してイーサネットフレームにカプセル化することで，カプセル化オーバヘッドを低減し，イーサネットが実現する PCIe の帯域が向上することも示す．

第五章では分離した複数の計算アクセラレータを用いた計算の最適化方式について述べる．本方式により複数の GPU を用いた Out-of-Core 処理を最適化するミドルウェアを Victream と呼んでいる．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 を自動で最小化する．Victream はデータパラレル計算向けの DAG (Directed Acyclic Graph) 型ミドルウェアであり，API の呼び出しによりミドルウェア内部でアプリケーションの処理を示す DAG を生成し，生成した DAG を遅延実行する．このとき DAG の頂点のタスクを GPU で実行する順序を最適化することで GPU に対するデータ I/O を最小化する．ここで GPU の計算が CPU の計算と異なる点は将来実行するタスクの入力データを予め GPU にロードするデータプリフェッチを行う点である．データプリフェッチはボトルネックリソースである I/O バスを常に稼働させるために重要である．本研究では GPU のデータプリフェッチとデータ I/O の最小化を同時に実現する新しいスケジューリング方式を提案する．

第六章では本研究で作成した Victream のプロトタイプと，プロトタイプを用いて行った評価結果について述べる．評価ではデータパラレル処理のベンチマークアプリケーションを用いて Victream が従来方式より優れた性能を実現することを示す．また，本研究で提案する新たなスケジューリング方式であるローカリティアウェアスケジューリングの拡張手法の有効性について検証する．また Victream のスケジューリングはヒューリスティック手法を用いているが，本手法が最良のスケジュールに対してどの程度の性能を実現するか Brute Force の計算機シミュレーションで検証する．さらに，ExpEther を用いてホストと GPU を分離したプラットフォームに Victream を適用し，Victream の有効性を示す．

最後に第七章でまとめと今後の研究課題について述べる．

第 2 章

関連研究

本章では本論の関連研究について述べる．また，関連研究と本研究との関係についても述べる．まず，I/O (Input/Output) バスの拡張と，それにより I/O デバイスのリソースを複数のホストから共有する研究について述べる．次に，パケットを効率的に処理するためのハードウェアオフロード技術とパケット集約技術の研究について述べる．次に分散，あるいはヘテロ計算リソースを使いやすくするためのミドルウェアの研究について述べる．最後に計算リソースを効率的に用いるためのミドルウェアの研究について述べる．

2.1 I/O バスの拡張と I/O デバイスのリソース共有に関する研究

I/O デバイスをホストから分離し，必要に応じてホストに柔軟に割り当てることにより I/O デバイスをホスト間で共有する技術は Krishnan らが [39] において提案した．Krishnan らの提案は専用のネットワークを用いて PCIe (PCI Express) をトンネリングするものだった．Thunderbolt [5] も PCIe を Thunderbolt ネットワークでトンネリングする機能を持つが，ネットワークに接続するホストは 1 台に限定されている．一方 Hou らはサーバホストとクライアントホストを PCIe のネットワークで低遅延に接続し，サーバがクライアントから I/O デバイスへの要求を受け付けるアーキテクチャを提案した [29].

また同一の I/O デバイスを複数のホストから同時に共有する研究も多く行われている．産業界からは独自規格による提案が行われたが [4]，技術の詳細は公開されなかった．その後 I/O デバイスを複数のホストから同時に共有する技術が PCI-SIG によって MR-IOV (Multi-Root I/O Virtualization) として標準化された [1]．しかし，筆者が知

る限り MR-IOV に準拠した I/O デバイスは販売されなかった．一方，筆者は本論の研究の発展として SR-IOV (Single-Root I/O Virtualization) [3] に準拠する I/O デバイスを複数のホストから同時に共有する技術を [62] で提案した．SR-IOV も PCI-SIG で標準化された規格だが，SR-IOV は単一ホスト内の仮想マシン環境で使用されることを想定している．SR-IOV に準拠した I/O デバイスは，仮想マシンから I/O デバイスへのアクセスのオーバヘッドを低減するため仮想マシン毎に専用のインターフェースを設けている．このような SR-IOV のアーキテクチャが実現する効率性は [51] や [71] で議論された．MR-IOV と異なり，SR-IOV に準拠した I/O デバイスはこれまでにも多数販売された．筆者らが [62] で SR-IOV のデバイスを共有する技術を提案した後にも様々な研究が行われている．これらの手法ではホストと I/O デバイスのアドレス空間の間でアドレスのリマップを行う．Tu らの手法ではリマッピングのために PCIe スイッチの Non-Transparent ブリッジ機能を用いる [65]．また Cao らの手法では独自の FPGA (Field Programmable Gate Aray) を用いる [16].

本論で提案した ExpEther では，単一ホスト内に制限されている PCIe ネットワークをイーサネットで拡張し，I/O デバイスを複数のホストから共有できるようにした．一方 PCIe ネットワークを拡張する他の目的として，複数の CPU (Central Processing Unit) 間の通信がある．Bo らと [13]Tu らは [66]PCIe スイッチの Non-Transparent Bridge の機能を用いて CPU 間の通信を実現した．また Hanawa らはインターコネクションのために独自のチップを開発した [25].

2.1.1 本研究との関係

本研究で提案する ExpEther は，イーサネットを用いて複数のホストによる I/O デバイスの共有を実現する．インターコネクションはハードウェアで実現しているため，従来提案されているソフトウェア処理が介在する手法より性能オーバヘッドが小さい．またハードウェアのブリッジに仮想化機構を導入し，イーサネットをホストに対して透過としているため，従来の OS (Operating System) やデバイスドライバの変更なくイーサネットで接続した従来の I/O デバイスを利用できる．従って I/O デバイスに SR-IOV や MR-IOV などの拡張の必要がない．また本論の提案ではホスト間通信用に ExpEther のブリッジを拡張しなかったが，そのような拡張によりブリッジが I/O デバイスの共有とホスト間通信の 2 つの機能を提供することも可能である．

2.2 パケット処理オフロードとパケット集約に関する研究

パケットの再送輻輳制御の処理をハードウェアにオフロードする技術は長年研究され実用化されてきた．例えば RDMA (Remote Direct Memory Access) は TCP や CEE (Converged Enhanced Ethernet) のプロトコル処理をオフロードする技術としてハイパフォーマンスコンピューティングで一般的に用いられている [2]．TCP は標準のイーサネットにおいて送信側でパケットロスに基づく輻輳制御を行う．一方 CEE はイーサネットスイッチの輻輳通知機能を用いる．従って CEE ではイーサネットスイッチの対応が必要である．

一方パケット集約にも多くの研究がある [17, 32, 57, 33, 34, 26, 38, 41, 48, 42, 73, 37]．Castro らは無線ネットワークの各ノードで待機時間とタイムアウト時間を組み合わせた集約手法を提案した [17]．また Jain らは通信遅延を増加させない hop-by-hop の適応的な集約を提案した [32]．Hasegawa らはパケット集約とネットワークコーディングを組み合わせた手法を提案した [26]．また無線 LAN では IEEE 802.11n でフレームの集約が標準化されている [57]．

2.2.1 本研究との関係

本研究で提案する ExpEther のハードウェアへのオフロードでは，遅延に基づく輻輳制御をイーサネットの End-End 間で行っている．従って ExpEther では標準のイーサネットを用いることができる．また遅延に基づく輻輳制御は，TCP のようなパケット欠落に基づく制御より輻輳に早く適応することができるため実現される通信遅延が短い．

またパケット集約は，著者らの提案手法は従来手法と異なり，通信遅延を増加させない適応的な集約を End-End 間で行う．これにより，従来のイーサネットを変更せず，また I/O パケットに追加で通信遅延を付与せずに I/O パケットをイーサネットフレームに集約し，通信帯域を増大する．

2.3 分散ヘテロ計算リソースを利用しやすくするミドルウェアに関する研究

分散する複数の計算リソースや，CPU と GPU (Graphics Processing Unit) 等とのヘテロ環境を用いた計算を容易にするミドルウェアの研究が多数行われた．

GPU を用いた計算に関しては，NVIDIA 社は GPU を汎用計算に用いるための開発環境である CUDA [46] を提供している．また GDM はアプリケーションプログラマが管理する必要がある GPU メモリを仮想化し管理を自動化する手法である [67]．Gdev は GPU の演算とメモリリソースを複数のアプリケーションの間で仮想化する技術である [35]．また TimeGraph は GPU を利用する複数の処理の間のスケジューリングを実現する [36]．Mars は MapReduce の形式で GPU のアプリケーションプログラムを作成可能にする [28]．また GPUStore はストレージの処理に GPU を利用できるようにする [60]．

複数の計算リソースを利用しやすくするミドルウェアでは，Dryad [31] や Spark [78] がアプリケーションプログラムの DAG(Directed Acyclic Graph) を作成することにより複数の計算リソースを用いた分散処理を行い易くしている．これらの研究はデータパラレルアプリケーションが対象である．Zaharia らはさらに，Spark をストリームデータに対応させる手法も提案した [77]．DryadLINQ は Dryad で LINQ が用いられるように Dryad を拡張した手法である [22]．

一方複数の GPU を用いたデータパラレルの分散処理では Dandelion が GPU の DAG 型ミドルウェアである PTask を拡張し，より高級な言語を用いてアプリケーションプログラムを作成可能にしている [53]．Spark-GPU は Spark の GPU 拡張である [75]．SWAT もまた Spark の GPU 拡張であり，JVM の言語で書かれたカーネル関数から GPU で実行する OpenCL で記述されたカーネルを生成する [23]．

一方，タスクの並列性を利用しタスクをヘテロ環境下の計算リソースに行わせるミドルウェアも多数研究されている．HYDRA はホスト内の複数の計算リソースを用いた連携処理を実現するミドルウェアである [69]．StarPU はヘテロ計算環境での計算を容易化するためのフレームワークである [12]．また OmpSs もヘテロ計算用のミドルウェアとして提案されている [21, 49, 50]

また近年，TensorFlow がディープラーニング向けの DAG 型フレームワークとして提案された [9] TensorFlow はディープラーニングのアプリケーションプログラムで使用する抽象度が高い関数を用意している．

2.3.1 本研究との関係

本研究で提案する Victream の目的は Spark-GPU の目的と同じであり，DAG 型フレームワークによって複数の GPU を用いた計算を行いやすくする．Victream はさらに，Out-of-Core 処理に着目し，Out-of-Core 処理の性能ボトルネックとなる GPU のデータ I/O を最小化するスケジューリングを行う．それにより，GPU を用いた計算を行いやす

くするだけでなく，計算ボトルネックを緩和し，GPU の計算リソースを効率的に用いることで高い計算性能を実現する．

2.4 計算リソースを効率的に用いるためのミドルウェアに関する研究

計算リソースを効率的に用いるためのミドルウェアでは多くの研究が行われている．

メニコアや共有メモリの計算機において，データアクセスを効率的に行うためにタスクのローカリティを考慮してスケジュールを行う手法が多く提案されている [10, 24, 54, 68]．Yoo らはそれまでのタスクパラレルではなくデータパラレルの処理に対してローカリティを考慮したスケジュール手法を提案した [74]．また Cho らは OS (Operating System) でキャッシュを考慮したメモリページの割り当てを行う手法を提案した [18]．Buttari らは線型代数の計算をタイル化し，タイル単位のスケジュールを行う事で効率的な計算を実現している [15]．

複数の計算リソースを協調させて効率的な計算を行うためのミドルウェアも多数研究が行われている．データパラレル処理の代表的なフレームワークは MapReduce である．MapReduce は計算を map と reduce の組み合わせとして抽象化する [20]．MapReduce をさらに発展させるスケジュール手法も多数提案されている [76, 11, 19]．またグラフ計算に特化したミドルウェアには GraphLab がある [43]．GraphChi は単一ホスト向けのグラフ計算ミドルウェアだが，SSD を用いることで複数の計算機並みの計算性能を実現する手法である [40]．

GPU 向けのミドルウェアでは Stuart と Owens が提案した GPU クラスタの MapReduce がある [59]．Shirahata らは GPU MapReduce を拡張しグラフ処理に特化したミドルウェアを提案した [56]．また複数の GPU を対象とする DAG 型のフレームワークには PTask がある [52]．

また多くのアプリケーションでは GPU を用いた Out-of-Core 処理で性能ボトルネックとなるのは演算ではなくデータスワップのデータ I/O である．Sundaram らは単一の GPU を用いた Out-of-Core 処理のデータ I/O を計算の実行前の静的な最適化で最小化する手法を提案した [61]．この手法では GPU の演算とデータ I/O はオーバラップせずシリアルに行っている．そのため全てのタスクは同一の GPU で処理され，GPU はデータ I/O かタスクの演算のどちらか一方しか行っていない構成となる．このような構成における GPU のデータ I/O の最小化は pseudo-Boolean 最適化問題に定式化できる．

また CPU と GPU のヘテロ計算環境を対象としたミドルウェアも提案されている．Maestro はヘテロ計算環境でデバイス間のロードバランスやデータ転送の最適化を行う [58]．タスクパラレル処理を対象とした分散ヘテロ計算環境用のミドルウェアには DAGuE などがある [14]．また Wu らは分散ヘテロ環境下で階層 DAG を用いてタスクの割り当ての最適化を行っている [72]．Wen らは複数のアプリケーションの間で CPU/GPU のヘテロ計算環境を最適に共有する手法を提案している [70]．

2.4.1 本研究との関係

本研究で提案する Victream は，関連研究である複数の GPU を対象とする DAG 型ミドルウェアの PTask と同様に，複数の GPU を用いて効率的な計算を行う事を目的とする．その中で Victream が着目するのは，従来着目されてこなかった GPU で大容量のデータを処理する Out-of-Core 処理である．Victream では Out-of-Core 処理のボトルネックである GPU へのデータ I/O を最小化するために新しいスケジュール手法を導入する．この領域では Sundaram らが単一 GPU で演算とデータ I/O が並列化しない環境でデータ I/O を最小化する手法を提案し，限定された問題を解いた．本研究では複数の GPU を用い，かつ GPU のタスクの演算とデータ I/O を並列に行う相対的に複雑な問題を解く．

第 3 章

ネットワークによる I/O デバイスの分離方式

クラウドサービスを提供するデータセンターでは，GPU (Graphics Processing Unit) や SSD (Solid-State Drive) を始めとする I/O (Input/Output) デバイスをホストの I/O スロットに挿入して用いている．このような構成では，I/O デバイスが各ホストに占有されるためハードウェアが稼働しない時間が増加しリソースの利用効率が低下する．それに加え，各ホストに搭載できる I/O デバイスの数は I/O スロット数に制限される．本章では，これらの非効率なリソースの利用と搭載の制限を，ホストから I/O デバイスを分離することで解決する手法を提案する．提案手法では，I/O デバイスをデバイスプールにプール化し，デバイスプールと各ホストを標準のイーサネットで接続することで，必要に応じて各ホストに I/O デバイスを割り当てる．本技術を ExpEther(エクスプレスイーサ) と読んでいる．本研究では，I/O デバイスのホストに対する柔軟な割り当てという新たな自由度を実現することにより，I/O デバイスのリソース利用効率を向上させ，ホストから所望の I/O デバイスを数の制限なく使用することを可能にする．さらに，ホストと I/O デバイスを接続するイーサネットのフレームに I/O パケットを集約してカプセル化することにより，I/O デバイス分離に関するオーバヘッドを低減する手法も提案する．

以下本章では，3.1 節で I/O デバイスのホストからの分離によるリソース利用率の向上，3.2 節で I/O デバイスを分離するシステムに要求されるシステム要求，3.3 節でイーサネットによるデバイス分離を実現する ExpEther のアーキテクチャ，3.4 節で複数の I/O パケットをイーサネットフレームに集約してカプセル化することによる I/O 帯域の向上手法について述べる．

3.1 I/O デバイスの分離によるリソース利用率の向上

近年のデータセンターでは，クラウドで様々なサービスを一括して提供するために数万台規模のホストを運用している．従って，データセンターではホストの導入コストや運用コストの削減が益々重要な課題となっている．そのための解決策の一つは，導入した計算リソースの利用率を向上し，データセンターの限られた空間を有効に活用することである．このため近年の計算機のホストは，計算機ラックに可能な限り計算機を搭載するため，高密度な実装で体積を抑えた仕様となっている．ところが I/O デバイスに関しては，各ホストの I/O スロットに挿入してホスト間で排他的に使用するため，リソースの利用効率が下がり，有効利用の観点で改善の余地がある．

ところで最近のデータセンターでは，ビッグデータ処理や AI(Artificial Intelligence) の用途で PCI Express (PCIe) SSD や GPU を計算アクセラレータとして多用するようになった．これらのアクセラレータは I/O デバイスとして実装されているため，データセンターにおける I/O デバイスの重要性は高まっている．これらの I/O デバイスは，ホストの内部では CPU (Central Processing Unit) と I/O バスで接続される．ここで現在最も利用されている I/O バスの規格は PCIe である．PCIe は CPU をルート，I/O デバイスをリーフとするツリートロポジーを構成する．従って PCIe のトポロジーは I/O デバイスを単独のホストに排他的に使用させることを前提としている．そのため PCIe では I/O デバイスを複数のホストに接続してリソースを融通させ合うことができない．

このような PCIe による I/O デバイスの接続制限のため，クラウドサービスを提供するデータセンターのホストには様々な利用者のサービスを収容することを想定し最大利用時の I/O デバイスを搭載する必要がある．しかし，ほとんどの利用者は最大限搭載された I/O デバイスを利用せず，リソースの利用効率が低下する．それに加え，I/O デバイスをホストに最大限搭載した場合でも，ホストの I/O スロット数以上のリソースを要求する利用者は，要求するリソースが I/O デバイスのみであっても，I/O デバイス確保のため別ホストを占有する必要がある．

本研究ではこれらの課題を解決するため，I/O デバイスをホストから分離し，各ホストに所望の I/O デバイスを柔軟に割り当てる ExpEther を提案する．ExpEther はホストと I/O デバイスを標準のイーサネットを用いて接続する．そして，イーサネットで接続した I/O デバイスをイーサネット上のリソースプールに集約し，それぞれをイーサネットで接続した任意のホストに割り当てることを実現する．提案手法は PCIe に準拠する I/O デバイスであれば，あらゆるデバイスを分離できる．

ExpEther では複数のホストと I/O デバイスをイーサネットで接続する．しかし，汎用の I/O デバイスは CPU で動作するデバイスドライバで制御される存在であり，ホスト間通信のように CPU との間で TCP-IP 通信を行えない．このため PCIe over Ethernet を実現する ExpEther ブリッジを導入する．この PCIe-to-Ethernet ブリッジは PCIe のパケットをイーサネットフレームにカプセル化することで CPU と I/O デバイスの間で I/O パケットである PCIe パケットを伝送する．ここで ExpEther ブリッジの処理はハードウェアで実現されるため，I/O デバイスの分離を小さいオーバヘッドで実現することができる．さらに ExpEther ブリッジはホストに対しホストと I/O デバイスを接続するイーサネットを，PCIe ネットワークとして仮想化する．これにより，従来の I/O デバイスやデバイスドライバ，OS (Operating System) を変更せずにイーサネットで接続した I/O デバイスを利用することができる．なお，ここで議論する共有とは，I/O デバイスは 1 つのホストに排他的に使用されるが，I/O デバイスを割り当てるホストは柔軟に変更できるという意味である．これに対し単一の I/O デバイスを複数のホストから同時に使用する意味での共有がある．このような手法は著者らが本研究の発展として [62] で提案した．また，ExpEther ではイーサネットスイッチを用いることにより，ホストに接続する I/O デバイスの数を自由に増大させることができる．

また，提案手法では PCIe over Ethernet を行うため，PCIe パケットをイーサネットフレームにカプセル化する．しかし一般的に I/O パケットのサイズは小さく，それらを個別にイーサネットフレームにカプセル化すると性能オーバヘッドが増大する．そこで本研究では複数の I/O パケットを集約し単一のイーサネットフレームにカプセル化する手法を提案する．それによりオーバヘッド削減し，イーサネットの PCIe のスループットを向上する．ここで PCIe パケットの伝送では，伝送遅延が接続する I/O デバイスの性能に大きく影響する．そのため，遅延の蓄積を最小化する必要がある．提案する集約手法は，集約する PCIe パケットを適応的に決定することで，集約のための伝送遅延の増加抑制と PCIe のスループットの向上を同時に実現する．

3.2 I/O デバイスの分離に関するシステム要求

本節ではホストと I/O デバイスを分離したシステムを実現する上で満たすべきシステム要求を議論する．後の 3.3 節で提案する ExpEther による I/O デバイス分離手法はこれらの要求を最大限満たすようにアーキテクチャが実現されている．

効率的なハードウェアリソースの利用:

リソースプールの I/O デバイスは任意のホストに割り当てられる必要がある．またそ

れらの割り当てを必要に応じて柔軟に変更できる必要がある．そして割り当ての変更はホストが提供するサービスを停止することなく行えることが必要である．

接続のスケーラビリティ:

今日の大型のデータセンターでは数万台規模のホストが運用されている．従って，そのようなシステムに対応するためには提案手法によって十分な数のホストと I/O デバイスを相互に接続できる必要がある．また，ホストと I/O デバイスの間をネットワークで接続する際は，ネットワークの実装はスケーラビリティを実現できるものである必要がある．太い導線を用いたネットワークの実装はスケーラビリティの観点で適さない．

互換性:

従来のシステムから I/O デバイスの分離システムへの移行では，従来用いていた汎用のホストや I/O デバイスを変更なく継続して使用できる必要がある．さもなくば，I/O デバイス分離システムの導入はコストが大きすぎるため困難である．ここで変更が望ましくないホストの構成要素には CPU, OS, デバイスドライバがある．それに加え，ホストと I/O デバイスを相互に接続するネットワークスイッチも汎用な製品を使用できることが望ましい．

管理性:

ホストから分離した I/O デバイスの管理は容易に行える必要がある．またシステムに接続する I/O デバイスの数が増大する場合，大規模な数の I/O デバイスの管理を容易に行うスケーラビリティも重要である．また，人のデータセンター管理者による管理の観点では，分離してプール化された I/O デバイスはリソースプールに高密度に収容し，その高密度な実装を容易に管理できる必要がある．そのような実装の一案は，19 インチラックにマウント可能なボックスの中に I/O デバイスを集積することである．またネットワークで接続したホストと I/O デバイスのそれぞれの状態は，それらを接続するネットワーク上のリソース管理用ホストから監視できることが望ましい．さらに，ホストと I/O デバイスを接続するネットワークのルーティング設定や，I/O デバイスのホストへの割り当ては，単純な管理で実現できる必要がある．

システムの継続性:

近年のコンピュータシステムはソフトウェア及びハードウェア共，技術の進展の速度が著しい．I/O デバイスの帯域は継続的に向上し，利用可能なデバイスの種類も増加している．I/O デバイスを分離するシステムは，接続するホストや I/O デバイスが進化しても継続的に相互の接続を実現するアーキテクチャを採用する必要がある．それには，実現するシステムがコンピュータシステムの実装の特定の一世代に捕らわれないことが必要である．

性能:

I/O デバイスの分離システムに対する最も重要な要求の一つは，ホストから I/O デバイスを分離するための性能オーバヘッドが小さい事である．この性能オーバヘッドはホストと I/O デバイスの間の通信遅延と通信帯域の両方に依存する．

遅延は CPU と I/O デバイスの間でデータを伝達するために必要な時間である．CPU が I/O デバイスへの命令を発行するために複数回の往復を伴う小さいデータの通信を行う場合，通信遅延が性能ボトルネックとなりやすい．このようなアクセスの一例は，ブロックデバイスへ小さいデータのアクセスを行う場合である．ホストから I/O デバイスを分離する場合，CPU と I/O デバイスの間の距離が増加するため通信遅延が増大する．そのためそれに起因する性能低下が I/O デバイス分離の通信遅延に関するオーバヘッドとなる．

一方通信帯域は，使用する通信パスのボトルネック帯域で決定される．この通信帯域は，CPU と I/O デバイスの間で大容量のデータ通信を行う場合に重要である．このようなアクセスの例としてブロックデバイスに大容量のデータアクセスを行う場合がある．ホストから I/O デバイスを分離する場合，ホスト内部の PCIe I/O バスより，ホスト外部のネットワークの方が帯域が小さいため，より通信帯域が低下する．よってこの帯域低下に起因する I/O デバイスの性能低下が通信帯域に関する I/O デバイス分離のオーバヘッドとなる．

以上から I/O デバイスの分離システムでは，これら 2 種類の性能オーバヘッドを抑制し，高性能な I/O インターコネクトを提供する必要がある．

3.3 イーサネットによる I/O デバイス分離のアーキテクチャ

本節では本研究で提案する ExpEther のアーキテクチャについて述べる．ExpEther はホストから I/O デバイスを分離し，デバイスプールに収容する．デバイスプールに収容した I/O デバイスは必要に応じて任意のホストに柔軟に割り当てることができる．提案する ExpEther は 3.2 節で述べたシステム要求を満たすように設計されている．

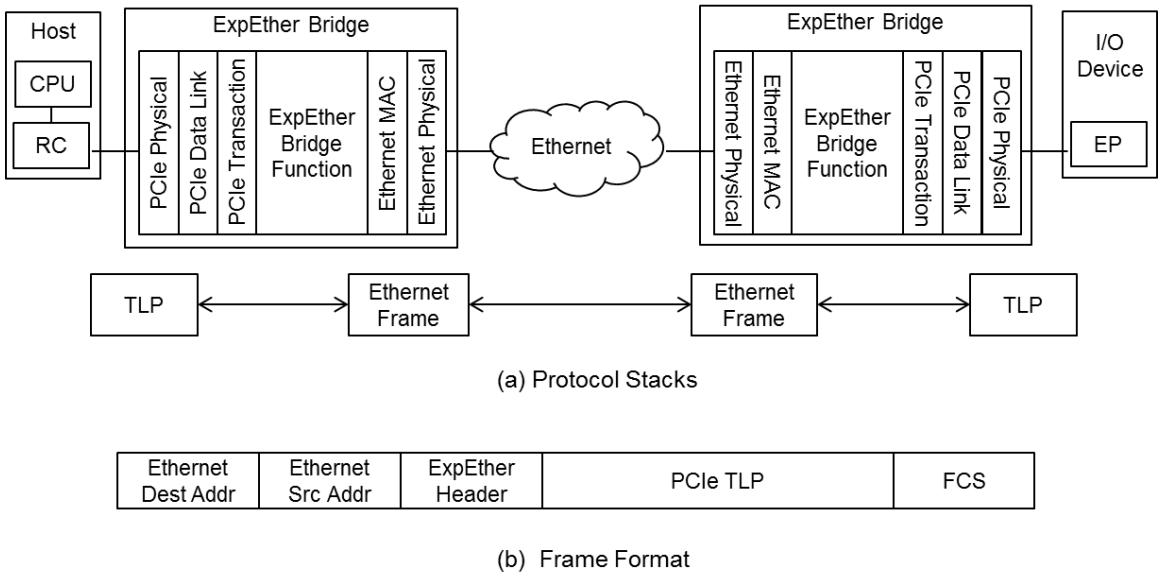


図 3.1 ExpEther のプロトコルスタックとフレームフォーマット．RC (Root Complex) と EP (Endpoint) は CPU ブリッジと I/O デバイスの PCIe インタフェースである．

ExpEther のプロトコルスタックを図 3.1 に示す．ExpEther はホストの PCIe バスをイーサネットに拡張し，I/O デバイスをイーサネットによって接続する．PCIe の I/O パケットはイーサネットフレームにカプセル化することで CPU と I/O デバイスの間を伝送する．PCIe の仕様では I/O パケットを Transaction Layer Packet (TLP) と呼ぶ [6]．PCIe に準拠した CPU や I/O デバイスは TLP を用いて通信を行う．PCIe バスをイーサネットに拡張する場合，CPU と I/O デバイスの間の遅延の増大が接続する I/O デバイスの性能に大きく影響する．そこで ExpEther ではイーサネットフレームへのカプセル化処理を PCIe バスとイーサネットをブリッジするハードウェア回路で行い，オーバヘッドを最小化する．さらに，ExpEther はイーサネットのフレーム伝送遅延を最小化するため，イーサネットでの輻輳を抑制する送信レート制御を行う．また ExpEther は，PCIe のイーサネットでのトンネリングに加え，イーサネットを PCIe ネットワークとして仮想化し，従来の OS やデバイスドライバをイーサネットで接続した I/O デバイスに使用可能にする．これらの低遅延通信と仮想化機能の詳細は 3.3.1 節で述べる．

ExpEther ブリッジは PCIe バスとイーサネットをブリッジするハードウェアであり，ASIC (Application Specific Integrated Circuit)，または FPGA (Field-Programmable Gate Array) として実装する．ホスト側では ExpEther ブリッジは HBA (Host Bus dapter) として実装され，ホストの I/O スロットに挿入して用いられる．一方 I/O デバ

イス側では I/O デバイスを多数収容してリソースプール化を実現する I/O リソースボックスの内部に実装される．

PCIe では各々のリンクをバスと呼び，バスをスイッチで接続したスイッチドネットワークを構成する．スイッチは PCIe スイッチと呼び PCIe バスの 1:N のファンアウト機能を提供する．PCIe ネットワークでは CPU をルート，I/O デバイスをリーフとするツリートポロジーを構成する．PCIe のスタックは物理層，データリンク層，トランザクション層の 3 つの機能レイヤから構成される．物理層とデータリンク層はそれぞれのバス毎に終端される．一方トランザクション層はルーティングと TLP の転送機能を担い，送信端と受信端の End-End で終端される．

ここで I/O デバイスをホストから分離することを考える．このとき単に PCIe バスの接続をイーサネットでトンネリングするのは PCIe の仕様の制限から困難である．PCIe のデータリンクレイヤでは TLP を再送するタイマの値が数十マイクロ秒に設定されている．そのためイーサネットのネットワーク遅延では PCIe のデータリンクレイヤの制限を満たすことができないからである．そこで ExpEther ではタイムアウトの発生を避けるため，PCIe バスを ExpEther ブリッジで一度終端した後にトンネリングを行っている．このアーキテクチャの詳細は 3.3.1 節で述べる．

図 3.1 に ExpEther のフレームフォーマットを示す．ExpEther では標準のイーサネットフレームを用いることで汎用のイーサネットスイッチでホストと I/O デバイスを接続する．ExpEther はフレームヘッダに ExpEther ヘッダを定義し，分離した I/O デバイスとの通信を制御する管理情報を保持させている．また，イーサネットの VLAN 番号を用いて I/O デバイスが割り当てられているホストの識別を行っている．

ホストと I/O デバイスの接続にイーサネットを用いる事によるメリットの 1 つは，イーサネットの自動ルーティング機能が利用できることである．これにより ExpEther で独自にネットワークのルーティングを行う必要がなくなる．もしホストと I/O デバイスの接続に独自のネットワークを開発する場合，ルーティングプロトコル自体を設計する必要がある．

3.3.1 ExpEther アーキテクチャの詳細

本節では ExpEther のアーキテクチャの詳細について，柔軟なデバイスの割り当て，PCIe ネットワークの仮想化，イーサネットフレームの高信頼低遅延通信，イーサネットリンクの集約による帯域向上，システム管理，の観点で述べる．

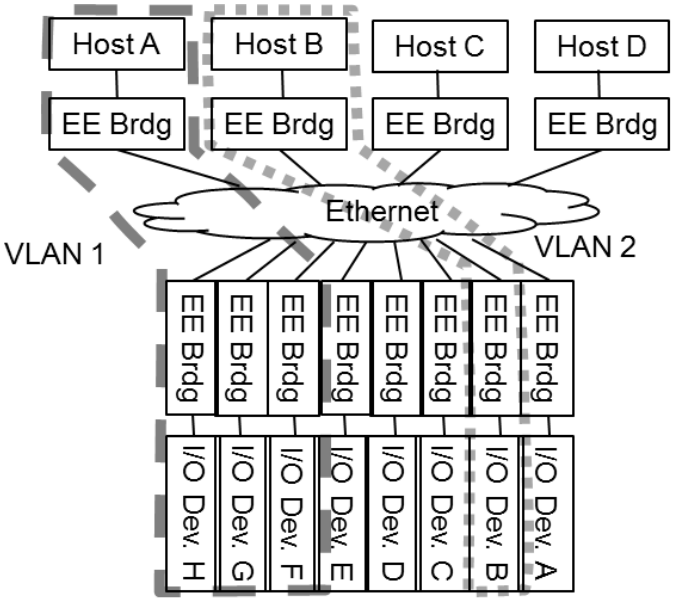


図 3.2　VLAN 番号を用いた I/O デバイスのホストへの割り当て

ホストへの柔軟なデバイスの割り当て:

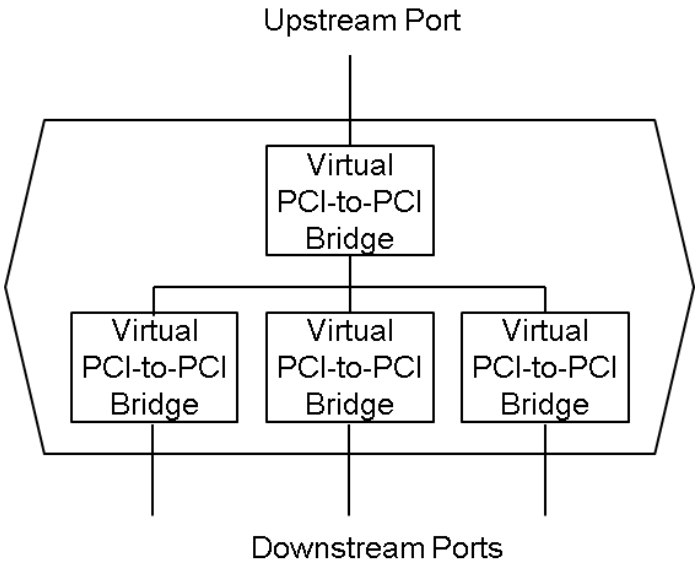
ExpEther ではデバイスプールの I/O デバイスを任意のホストに柔軟に割り当てることができる．そのためにイーサネットの VLAN 番号を用いる．ExpEther ブリッジは単一の VLAN 番号に属している．そして I/O デバイス側の VLAN 番号を所望のホストの ExpEther ブリッジの VLAN 番号に変更することで I/O デバイスをホストに割り当てることができる．図 3.2 に示すように，同じ VLAN 番号を持つホストと I/O デバイスはグルーピングされ 1 つの仮想的な計算機を構成する．また，I/O デバイスが割り当てられているホストから I/O デバイスを削除し，別のホストに割り当てる操作も可能である．これらの操作はホストでは汎用 OS でサポートされている I/O デバイスのホットプラグ及びホットリムーブのイベントとして処理される．

仮想 PCIe ネットワーク:

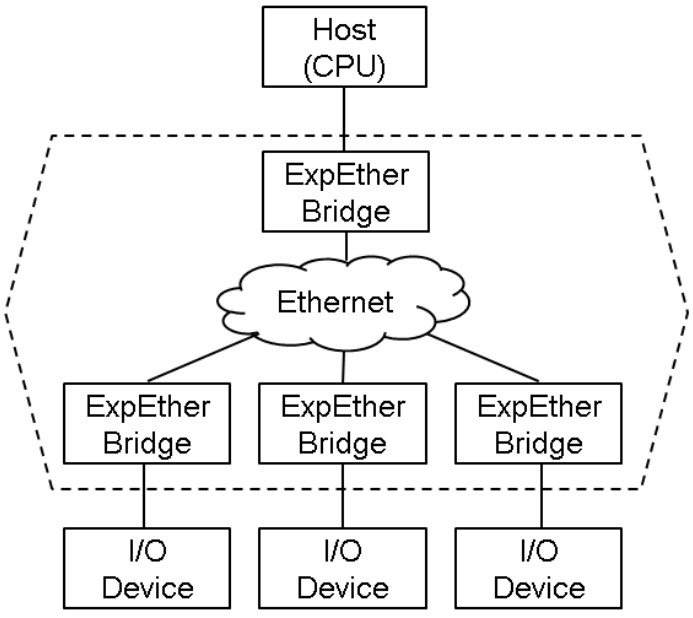
3.2 節で述べたように PCIe バスの途中で TLP をトンネリングすることは困難である．このため，ExpEther では PCIe バスを ExpEther ブリッジで終端する．この構成ではイーサネットの領域をホストにどのように管理させるかが課題となる．そこで ExpEther ではイーサネットの領域を PCIe ネットワークとして仮想化する．具体的には，イーサネット上で分散仮想 PCIe スイッチを形成する．ここで PCIe スイッチは PCIe の標準仕様で定義されたコンポーネントの 1 つであり，PCIe バスのファンアウト機能を提供する．図 3.3(a) に示すように，標準の PCIe スイッチは上流と下流ポートのそれぞれに仮想

PCI-PCI ブリッジを持つ．それに対し ExpEther の場合，図 3.3(b) に示すように，ホスト側の ExpEther ブリッジが PCIe スイッチの仮想上流 PCI-PCI ブリッジに，I/O デバイス側のブリッジが下流 PCI-PCI ブリッジに相当するよう仮想化する．この仮想化により，ホストの BIOS(Basic Input/Output System) や OS は ExpEther ブリッジとイーサネットを合わせた領域を PCIe スイッチであると認識する．また上流と下流の PCI-PCI ブリッジの間のバスは従来は PCIe スイッチの内部にあった論理的なバスで PCIe のデータリンクレイヤのタイムアウトの制限を受けないためイーサネットで実現することが可能である．その結果イーサネットの領域はホストに対し透過となり，ホストの PCIe ツリーはイーサネットに拡張される．このような構成により，イーサネットを介してホストに I/O デバイスを接続すると，接続したデバイスがホストの PCIe ツリーの配下に現れるようになる．以下ではホスト側の ExpEther ブリッジを上流 ExpEther ブリッジ，I/O デバイス側の ExpEther ブリッジを下流 ExpEther ブリッジと呼ぶ．

ExpEther ではデバイスプールの I/O デバイスを任意のホストに割り当てる際，下流 ExpEther ブリッジも同時に割り当てる．そのため分散仮想 PCIe スイッチの下流 PCI-PCI ブリッジはホストに割り当てられた I/O デバイスの数に比例する．OS によってはこの下流 PCI-PCI ブリッジの変動に対応しない場合がある．そこで ExpEther ではホストに常に同じ数の下流 PCI-PCI ブリッジを示し，PCIe 空間上の一定のリソースを確保する．そのため上流 ExpEther ブリッジは接続する I/O デバイスの数に関わらず仮想的に下流ブリッジをホストに示す．これにより BIOS や OS は常に同じ分散仮想 PCIe スイッチの構成を認識する．



(a) PCIe Switch



(b) Distributed Virtual PCIe Switch

図 3.3　PCIe スイッチと ExpEther の分散仮想 PCIe スイッチ

イーサネットフレームの高信頼低遅延通信:

今日用いられている汎用計算機のコンピュータアーキテクチャでは，CPU と I/O デバイスの間で送受信されるデータは通信の間に欠落しないことが前提である．その実現のため，PCIe ではデータリンク層で到達確認ができなかった TLP の再送を行っている．従って I/O デバイスをイーサネットを用いてホストから分離する場合，それらの間で高信頼な通信機能を提供する必要がある．しかし，イーサネット自体は信頼性がない通信機能であり，従来のホスト間通信でその機能を実現しているのは TCP(Transmission Control Protocol) などの上位機能である．そこで ExpEther では ExpEther ブリッジによる End-End 間でイーサネットの高信頼通信機能を提供する．また，イーサネットで接続される I/O デバイスの性能は TLP の通信遅延に大きく影響を受ける．この TLP の遅延はホストと I/O デバイスを接続するイーサネットスイッチでキューイングが発生すると増大する．そこで ExpEther では，イーサネットにおける遅延を最小化するため，輻輳を抑制する手法をさらに採用する．この輻輳制御手法の詳細は [55] で述べたため，ここでは概要を述べるに留める．

まず高信頼通信を実現する再送手法では，ExpEther ブリッジは対向ブリッジから到達確認を受信しなかったイーサネットフレームの再送を行う．再送の契機は同一フレームに対する Acknowledgement 通知フレームを受信した場合か，タイムアウトが発生した場合である．ExpEther の再送では，到達確認を受信しなかったイーサネットフレーム以降の全てのフレームを再送する．この手法は一般に”go-back-N automatic repeat request” と呼ばれている．

一方輻輳制御手法には遅延モニタに基づく送信レート制御手法を用いる．TCP Reno と同様に，ExpEther ブリッジは TLP をカプセル化したイーサネットフレームの End-End 間の遅延を監視し，イーサネットスイッチにおけるキューイング遅延が最小となるよう送信レートの調整を行う．ここで ExpEther の手法の特徴は，送信レートの調整のフィードバックに Acknowledgement フレームの受信レートを用いる事である．従来手法ではイーサネットフレームの送信と Acknowledgement 受信までの Round-Trip Time (RTT) を用いていた．しかし，イーサネットスイッチにおけるキューイングは，実際の RTT の増大より Acknowledgement フレームの受信レートに早く現れる．そのため ExpEther ではより早い輻輳の抑制が実現できる．

ここで ExpEther の輻輳制御は遅延に基づく手法を採用しているため，ホストと I/O デバイスを接続するイーサネットには専用のネットワークを用いる必要があり，通常のインターネット用のトラフィックと混在することができない．何故なら ExpEther のトラフィックと TCP のようなパケット欠落に基づく制御方式のトラフィックを混合させた場

合，パケット欠落に基づく制御方式のトラフィックがネットワークリソースを多く消費し，ExpEther のトラフィックにイーサネットの帯域リソースが割り当てられなくなるためである．

イーサネットパスの集約:

ExpEther で接続した I/O デバイスの性能ボトルネックの要素の 1 つにホストと I/O デバイスの通信帯域がある．一般にイーサネットの帯域は PCIe より小さい．例えば 4 章で述べる ExpEther ブリッジの試作器のイーサネットインターフェースは 10Gb/s であるが，PCIe バスのインターフェースは Gen2 x8 であり 32Gbp/s である．さらに，PCIe の TLP をイーサネットフレームにカプセル化するとカプセル化オーバヘッドが生じるため実行帯域がさらに低下する．

この性能ボトルネックを緩和するため，ExpEther では複数のイーサネットパスを集約し単一の PCIe バスのトンネリングを行うことができる．そのため ExpEther のブリッジチップには複数のイーサネットポートが実装されている．送信側のブリッジは TLP をカプセル化したイーサネットフレームをこれらのポートからラウンドロビンで送信する．一方受信側のブリッジは，集約したパス間の通信遅延のスキューに対応するため，バッファを用いて受信フレームの並べ替えを行う．これに対し一般に用いられている IEEE802 リンクアグリゲーションは，この目的に適用することができない．なぜなら，リンクアグリゲーションは送信元/先のアドレスのハッシュ値に基づいてイーサネットフレームを送信するパスを選択するからである．従って，同一の送信元/先のペアに属するイーサネットフレームは IEEE802 を用いても集約による帯域向上を実現できない．

システム管理:

デバイスプール内の I/O デバイスの管理や所望のホストへの割り当てにはシステムマネージャを用いる．システムマネージャは管理ソフトウェアであり，システムマネージャのホストは，ホストと I/O デバイスを分離するイーサネットに接続する．ここでシステムマネージャのホストはイーサネットに ExpEther ブリッジではなく標準の NIC (Network Interface Card) を用いて接続する．人のデータセンター管理者はシステムマネージャにコマンドを入力することで I/O デバイスのホストへの割り当てを変更できる．このとき，システムマネージャは下流 ExpEther ブリッジに VLAN を変更する管理フレームを送信する．またデータセンター管理者はホストと I/O デバイスの状態をシステムマネージャを用いて監視できる．ホストと I/O デバイスの状態は ExpEther ブリッジが定期的に送信する情報を集約して作成する．

また，システムマネージャは OpenStack 等のデータセンターのリソース管理ソフトウェアからアクセスできる API (Application Programming Interface) も提供する．

表 3.1　ExpEther のアーキテクチャの特徴と制約

Item	Description
I/O bus protocol	PCI Express
Network protocol	Ethernet
BIOS	Commercially available BIOS
OS	Commercial available OS
I/O device	PCIe-compliant commercially available device
PCIe space isolation among hosts	VLAN
Ethernet region in PCIe tree	PCIe switch
I/O assignment	System manager
Device monitoring	System manager
Maximum hosts	4096
Maximum I/O devices assigned to each host	32
Device driver for interconnection	Not needed
Hot plug and removal	Native PCIe events
Reliable TLP transport	Supported

3.3.2　ExpEther アーキテクチャの特徴と制約

表 3.1 に ExpEther のアーキテクチャの特徴と制約をまとめる．これらの大半は前節で詳細を述べた．本研究で提案した ExpEther は I/O デバイスを任意のホストに柔軟に割り当てることでリソースの利用効率を向上させることができる．

ExpEther は接続するホストや I/O デバイス数が多い大規模なシステムにも適用できる．システムに収容するホストの最大数は VLAN 番号の最大値で制限される．一方，各ホストに割り当てられる I/O デバイスの数は PCIe アドレスの 5 ビットのデバイス番号により 32 個に制限される．これは 5 ビットのデバイス番号により一つの上流 ExpEther ブリッジに接続できる下流 ExpEther ブリッジの数が制限されるからである．

ExpEther は分散仮想 PCIe スイッチを構成することでイーサネットの領域をホストに対して透過とする．I/O デバイスのホストへの割り当ては PCIe のホットプラグイベン

トとして処理される．このような構成により，ExpEther では従来の OS やデバイスドライバ，イーサネットスイッチ，PCIe に準拠した I/O デバイスを変更なく使用できる．ExpEther は OS やドライバのバージョンにも非依存である．また，イーサネットが透過であるため，イーサネットのための特別なデバイスドライバは必要ない．ExpEther はハードウェアだけで実現される仮想化システムである．

ExpEther のシステムマネージャはホスト間の柔軟なデバイスの割り当てを大規模なシステムでスケーラブルに行う手段を提供する．またシステムに接続するホストと I/O デバイスの監視も行う．

さらに ExpEther の遅延に基づく輻輳制御方式がイーサネットにおけるキューイングの蓄積を抑制し低遅延の PCIe の通信を実現する．また，ExpEther では複数のイーサネットのパスを集約することで広帯域の PCIe の通信を提供する．これらの手法により通信遅延と通信帯域による I/O 性能のボトルネックを緩和し，高性能な I/O デバイス分離システムが実現される．

3.4 複数の I/O パケットの集約

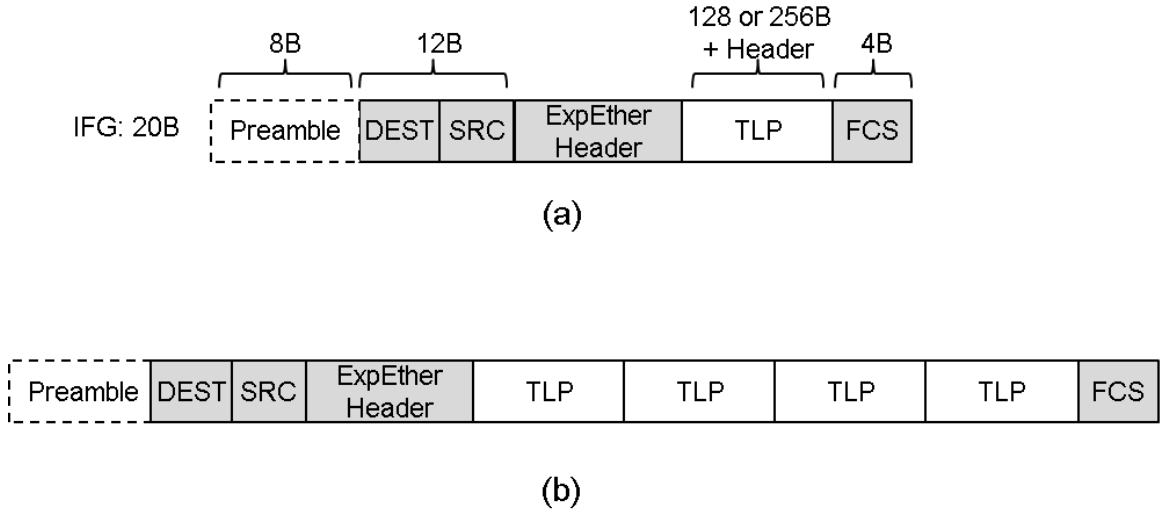


図 3.4 TLP のイーサネットフレームへのカプセル化. DEST: 送信先アドレス. SRC: 送信元アドレス. IFG: インターフレームギャップ. FCS: Frame Check Sum. (a) 個別 TLP のカプセル化. (b) TLP の集約.

3.4.1 TLP カプセル化のオーバヘッド

ExpEther では TLP をイーサネットフレームにカプセル化する際にカプセル化オーバヘッドが生じる．図 3.4(a) に示すように，カプセル化では TLP にイーサネットフレームの送信先/元アドレスや ExpEther ヘッダを付与する．さらに，イーサネットフレームを送信する場合にはイーサネットのプリアンブルやインターフレームギャップにも帯域リソースを割く必要がある．ここで TLP のサイズは一般的に小さくシステムにより 128 や 256 バイトである場合が多い．そのためカプセル化オーバヘッドが増大しイーサネットの PCIe の実効帯域が低下する．結果，広帯域な PCIe 通信を行う I/O デバイスでは，カプセル化オーバヘッドで低下した PCIe の帯域が性能ボトルネックになる．そのような通信を行うデバイスの例として PCIe SSD や GPU がある．

3.4.2 複数の TLP の同一イーサネットフレームへの集約

このようなカプセル化のオーバヘッドは複数の TLP を同一のイーサネットフレームに集約してカプセル化することで低減できる．図 3.4(b) に示すように，複数の TLP を集約する場合，アドレスフィールドや ExpEther ヘッダ等のオーバヘッドの要因を集約した TLP の間で共有できる．これにより，イーサネットの PCIe の実効帯域をイーサネットの物理帯域に近づけることができる．

ここで ExpEther の設計指針の 1 つは汎用のイーサネットスイッチを利用することである．そのため TLP の集約は ExpEther ブリッジ間の End-End で行い，イーサネットスイッチには透過にする必要がある．なぜなら商用のイーサネットスイッチを TLP の集約に対処できるように変更するコストは現実的ではないからである．

別の重要な観点は，集約に伴う追加の通信遅延である．従来の End-End 間の集約では, 送信側で一定時間後続パケットの到着を待ち，その間に到着したパケットを集約していた [32]．しかし，I/O デバイスの性能は TLP の通信遅延に大きく影響を受けるため，追加の遅延付与は避けるべきである．また汎用の低遅延イーサネットスイッチを I/O デバイス分離のインターコネクトに用いる場合，イーサネットフレームのホストと I/O デバイスとの間の往復遅延は 10 μ 秒以下となる [7]．この遅延は 10Gb/s のネットワークに長いイーサネットフレームを送信する時間と同程度である．つまり送信側で TLP の集約を行うために一定時間 TLP の到着を待つことは，RTT と同程度の通信遅延を TLP に追加で付与することに等しい．

そこで本研究で用いる集約手法では，ExpEther ブリッジの間で End-End 間で適応的

に TLP の集約を行う．TLP の集約数をイーサネットのパス内のボトルネックリンクの帯域に適応して制御し，追加の通信遅延の付与は行わない．また ExpEther ブリッジの間で集約を行う事により標準のイーサネットが I/O デバイスの分離に利用できるようにする．提案手法では送信キューにあて先が同一の複数の TLP が格納されていた場合，そのあて先の TLP に送信権が与えられたタイミングで格納されている全ての TLP を集約する．集約するパケットは適応的に決定され，TLP を集約するために追加の通信遅延の付与は行われない．この点で提案は従来手法と異なっている．また，TLP を集約する送信側がネットワークのボトルネックリンクと隣接していなくても適応的に集約を行うことができる．以下本論文では提案手法を” 適応的な集約” と呼ぶ．

3.4.3 適応的な集約のアーキテクチャ

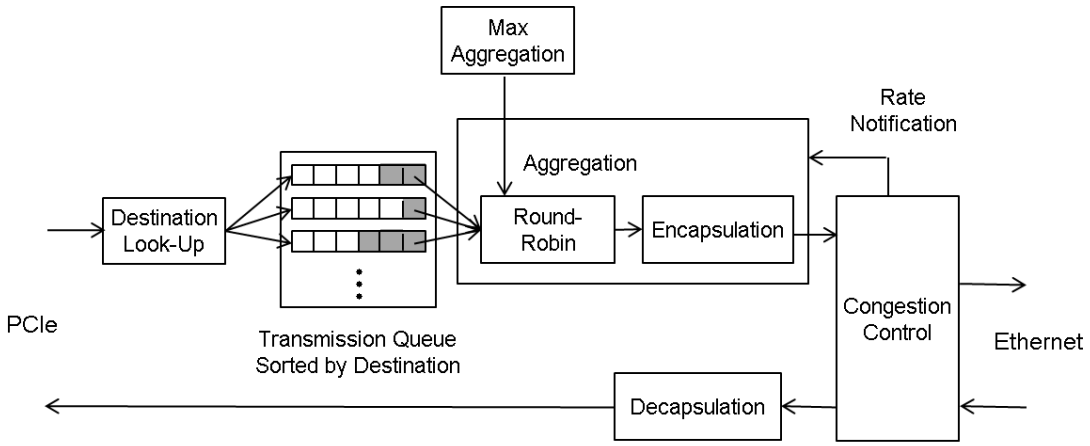


図 3.5 TLP の集約のための ExpEther ブリッジのハードウェア構成

適応的な集約では，イーサネットの PCIe 帯域が接続する I/O デバイスの性能ボトルネックとなる場合自動的に集約を行う．図 3.5 は適応的な集約のための ExpEther ブリッジのハードウェア構成である．図の上部は PCIe バスから受信した TLP を集約しイーサネットフレームにカプセル化して送信する機能モジュール群である．一方下部はイーサネットフレームをデカプセル化し集約された TLP を抽出して PCIe バスに送信するモジュール群である．

あて先検索モジュール (Destination Look-Up Module) は TLP を PCIe バスから受信し，あて先の MAC アドレスを検索する．あて先の MAC アドレスは TLP のあて先である PCIe ノードが接続する ExpEther ブリッジのアドレスである．続いてあて先検索

モジュールは TLP を送信キュー (Transmission Queue) の内部キューに格納する．送信キューはあて先の ExpEther ブリッジ毎に内部キューを保持しており，TLP はあて先毎に分けて格納される．ここで内部キューのリソースは静的に確保されている．PCIe バスから受信した TLP は内部キューに順番に格納され，同一あて先には先に受信した TLP が先にイーサネットに First-In-First-Serve(FIFS) で送信される．

集約モジュール (Aggregation Module) は複数の TLP を単一のイーサネットフレームに集約する．集約モジュールはラウンドロビンモジュール (Round-Robin Module) とカプセル化モジュール (Encapsulation Module) から構成される．ラウンドロビンモジュールは送信キューに格納されたあて先が同一の TLP を集約して取り出し，カプセル化モジュールはそれらを単一のイーサネットフレームにカプセル化する．

ラウンドロビンモジュールは送信キューの各内部キューをラウンドロビンで検索し，検索時にキューに格納されている全ての TLP を取り出す．各々の内部キューから TLP を取り出す帯域はそれぞれのあて先に至るイーサネットのパスのボトルネックリンクの帯域に調整する．そのため内部キューから TLP を取り出すレートは TLP のあて先毎に異なる．これらの帯域の値は輻輳制御モジュール (Congestion Control Module) により設定される．各内部キューから異なるレートで TLP の引き抜きを行うため，ランドロビンモジュールは不足ラウンドロビンの手法を用いている．不足ラウンドロビンでは 1 つのキューから複数の TLP を取り出す場合，取り出されたキューは取り出した TLP のデータ長に対応するクレジットを消費する．そして取り出されたキューは消費したクレジットが貯まるまで探索が行われない．また一度に内部キューから取り出される TLP の最大数は最大集約モジュール (Max Aggregation Module) から設定される．

3.3 節で述べたように，輻輳制御モジュールはイーサネットにおける輻輳を抑制するためホストと I/O デバイスを接続するそれぞれのイーサネットのコネクションに対し通信遅延に基づく送信レート制御を行っている．輻輳制御モジュールは通信遅延が増加すると，輻輳を抑制するために送信レートを減少させ，輻輳がない場合は送信レートを増大させる．従って輻輳制御モジュールはイーサネットのコネクションが使用しているネットワークパス上のボトルネックリンクの帯域情報を保持している．つまり輻輳制御モジュールが設定している各コネクションの送信レート自体がネットワークパス上のボトルネックリンクの帯域に等しい．

このような構成により，適応的な集約ではあて先毎に分類されている送信キューを探索した際，格納されている複数の TLP を同時に取り出して集約し，同一のイーサネットフレームにカプセル化する．送信キューから TLP を取り出すレートは TLP のあて先に至るイーサネットパス内のボトルネックリンクの帯域に調整される．適応的な集約では

イーサネットの PCIe 帯域が I/O デバイスの性能ボトルネックとなる場合に TLP の集約を行い，イーサネットの PCIe の帯域を増加させる．逆に PCIe 帯域が性能ボトルネックとならない場合は TLP の集約を行わない．また，適応的な集約は TLP を集約するために新たな通信遅延を付与せず，低遅延の TLP 通信を実現する．さらに，これらの集約は ExpEther ブリッジの End-End 間で行われるため，システムには汎用のイーサネットスイッチが使用できる．

3.4.4 適応的な集約の定式化

本節では適応的な集約が PCIe の実効帯域を増加させる効果の定式化を行う．TLP 集約の目的はイーサネットの PCIe の実効帯域が接続する I/O デバイスの性能ボトルネックとなる場合，そのボトルネックを緩和することである．もし低帯域の I/O デバイスが接続された場合，PCIe の帯域はボトルネックとならないため TLP の集約を行わなくともデバイス本来の性能を実現できる．反対に，広帯域の通信を行う I/O デバイスが接続された場合，PCIe の帯域がボトルネックとなるため TLP を集約する必要がある．集約では ExpEther ブリッジが PCIe バスから入力される PCIe の通信帯域に追随するように，イーサネットの PCIe の実効帯域を増加させる必要がある．

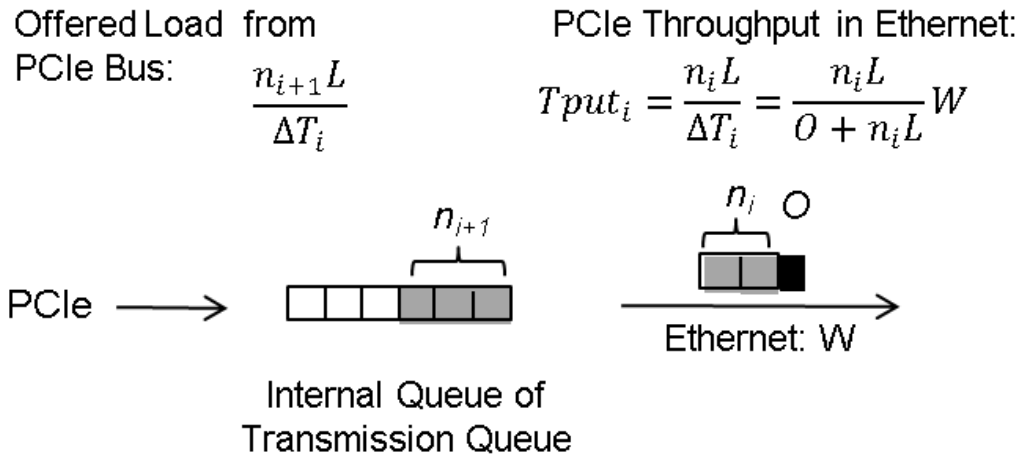


図 3.6　TLP の集約を記述するパラメータの関係. n: 集約する TLP の数, L: TLP のデータ長, ΔT: TLP を集約したイーサネットフレームをイーサネットに送信するために要する時間, $Tput$: イーサネットの PCIe 帯域, O: カプセル化オーバヘッド, W: 割り当てられたイーサネットの帯域.

本節ではイーサネットによる PCIe の実効帯域を定式化し，TLP の集約で，その帯域が

PCIe バスの入力帯域に追随することを示す．以下では図 3.6 を用いて説明する．ここでイーサネットの PCIe 帯域 $Tput$ は，式 3.1 に示すようにイーサネットフレームに集約する TLP の数 n, 集約されるそれぞれの TLP の長さ L, カプセル化オーバヘッド O，イーサネットのパス上のボトルネックリンクに割り当てられている通信帯域 W で決まる．カプセル化オーバヘッドはイーサネットフレームのヘッダやインターフレームギャップなどに起因する．

$$Tput = \frac{nL}{O + nL} W \tag{3.1}$$

式 3.1 より，イーサネットの PCIe 帯域は n の増加により単調に増加する．ここでは議論の簡単化のため，それぞれの TLP の長さは等しいとし，カプセル化オーバヘッドは集約する TLP の数に依存しないとする．今，i 番目のイーサネットフレームに n_i 個の TLP を集約しイーサネットに送信したとする．また ΔT_i は ExpEther ブリッジからイーサネットにこのフレームを送信しきるまでにかかった時間とする．この ΔT_i の間は，イーサネットの PCIe の帯域 $Tput_i$ は式 3.1 で示される．また，n_{i+1} は ExpEther ブリッジが ΔT_i. の間に PCIe バスから受信する TLP の数とする．すると n_i と n_{i+1} の大小関係は ΔT_i の間のイーサネットの PCIe 帯域と PCIe バスから受信した帯域の大小関係に一致する．前述したように TLP の集約の目的はイーサネットの PCIe 帯域を ExpEther ブリッジが PCIe バスから受信する TLP の帯域に追随させることである．従って，所望の機能は式 3.1 で表される帯域の集約 TLP 数に対する単調増加の性質を用いることにより，次の ΔT_{i+1} の期間では n_{i+1} 個の TLP を集約してカプセル化することで実現できる．

ここで，一つのイーサネットフレームに集約できる TLP の数は最大集約モジュールから設定された値で制限される．そのため ExpEther ブリッジが最大数の TLP を集約した場合でも，イーサネットの PCIe 帯域が I/O デバイスが発行する PCIe 帯域より小さければイーサネットで接続した I/O デバイスの性能ボトルネックとなる．この場合，ExpEther ブリッジが PCIe バスから受信する帯域の方がイーサネットの PCIe の帯域より大きいため，ExpEther ブリッジの送信キューに格納されるデータが増大し，やがて ExpEther ブリッジから PCIe バスの対向コンポーネントに送信停止を依頼するバックプレッシャーが発生する．これにより，PCIe バスからの受信は ExpEther ブリッジの送信キューに空きができ，ExpEther ブリッジが PCIe バスから再度データを受信できるようになるまで停止する．

以上の定式化により，適応的な集約手法は，イーサネットの PCIe 帯域がホストから分離した I/O デバイスの性能ボトルネックとなる場合，TLP を集約してカプセル化するこ

とでイーサネットの PCIe 帯域を PCIe バスの入力帯域に追随させることを示した.

第 4 章

ネットワークによる I/O デバイスの分離方式の実装と評価

本章では第 3 章で提案したホストから I/O(Input/Output) デバイスを分離する ExpEther について，技術の実現性を検証する．本章ではまず，実証のために試作した ExpEther のプロトタイプについて述べる．また，プロトタイプを用いて行った性能評価について述べる．性能評価では ExpEther を用いることでデータセンターで用いられている汎用 I/O デバイスを小さい性能オーバヘッドで従来のソフトウェアやハードウェアの変更なくホストから分離できることを示す．また，I/O パケットを集約することでイーサネットの実効 PCIe 帯域を向上させる手法も評価する．

4.1 I/O デバイス分離システムの実装

PCIe (PCI Express) とイーサネットのブリッジを行う ExpEther ブリッジを FPGA (Field-Programmable Gate Array) を用いて試作した．図 4.1 に試作したブリッジの内部モジュールの構成を示す．PCI-PCI ブリッジの IP コアは PCIe バスを終端するための物理層とデータリンク層の機能を保持する．PCIe-イーサネットブリッジ (PCIe-to-Ethernet Bridge) は TLP (Transaction Layer Packet) をイーサネットフレームにカプセル化する．また，3.3 節で述べたイーサネットが仮想分散 PCIe スイッチを構成するようにホストに疑似するため，ホストにアクセスさせる仮想レジスタを保持する．一方イーサネット転送エンジン (Ethernet Forwarding Engine) は再送輻輳制御を行う．マルチアービトレーションモジュール (Multi-Path Arbitration Module) は TLP をカプセル化したイーサネットフレームを複数のイーサネットのパスを用いて送信する．なお，図 4.1 には

イーサネットポートを 2 つ記載しているが，本方式のポート数に制限はない．試作した PCIe のインターフェースは Gen2 x8，イーサネットのインターフェースは 10GbE が 2 ポートである．また ExpEther ブリッジは複数のデバッグインターフェースと管理用レジスタを保持しており，ブリッジの状態の監視やコンフィグレーションの設定が行えるようになっている．

また適応的な集約のプロトタイプでは，ExpEther ブリッジに図 3.5 に示した拡張を行った．このとき，図 3.5 に示した輻輳制御モジュール以外の機能は図 4.1 の PCIe-イーサネットブリッジに実装した．また輻輳制御モジュールは原理確認のためイーサネット転送エンジンを単純なレートリミッタに置き換え，後述する 4.2 節で基本的な機能の検証を行った．

図 4.2 は作成したホストバスアダプタ (Host Bus Adapter) と I/O デバイスを収容する I/O リソースボックスのプロトタイプである．ホストバスアダプタは ExpEther ブリッジを保持し，ホストの I/O スロットに挿入する PCIe カードである．また I/O リソースボックスは PCIe に準拠した I/O デバイスを 2 個まで収容する．また横にして並べると 19 インチラックに搭載できる．I/O リソースボックスはそれぞれの I/O スロットに ExpEther ブリッジが実装されている．それぞれの I/O スロットの VLAN 番号は独立で設定でき，それぞれのスロットに挿入された I/O デバイスを異なるホストに割り当てることができる．

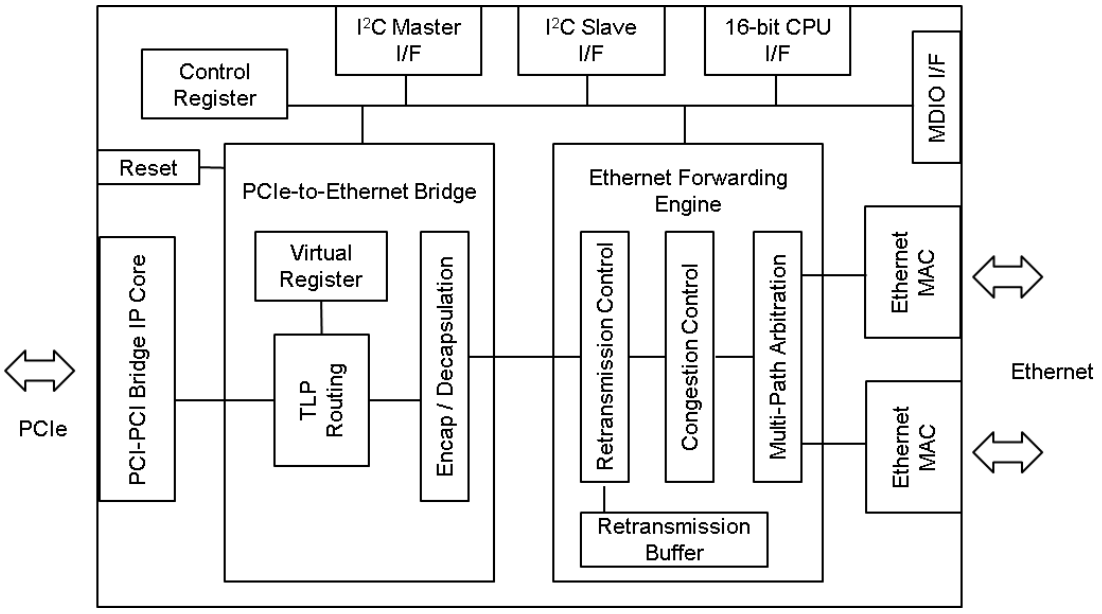


図 4.1　ExpEther ブリッジの内部機能構成

(a)

(b)

図 4.2　ExpEther のプロトタイプ. (a) ホストバスアダプタ. (b) I/O リソースボックス.

4.2 性能評価

本節では ExpEther のプロトタイプを用いて I/O デバイスをホストから分離し，リソースプール化して複数のホスト間で共有するシステムにおける I/O デバイスの性能を評価する．評価には PCIe SSD (Solid-State Drive) と GPU (Graphics Processing Unit) を用いる．これらは今日のデータセンターで一般的に用いられている汎用 I/O デバイスである．本節の評価により，これらの汎用デバイスが小さいオーバヘッドでホストから分離可能であることを示し，提案技術が一般のデータセンターに適用可能である事を実証する．

評価に用いたホストは個別の評価で異なるが汎用のワークステーションかラックマウントサーバである．OS (Operating System) には CentOS や Ubuntu 等の一般的な Linux ディストリビューションを用いた．ホストのメインメモリには I/O デバイスのベンチマークを動作させるために十分な容量を搭載した．ホストと I/O デバイスを接続するイーサネットのトポロジーは，説明が必要であれば各評価で述べる．I/O デバイスの性能評価は 2 つの場合について行った．1 つ目は ExpEther により I/O デバイスをホストから分離した場合であり，2 つ目はホストと I/O デバイスの間で通信される TLP を適用的な集約を用いて 1 つのフレームに集約した場合である．

4.2.1 ExpEther による I/O デバイス分離システムの性能評価

本評価では，ホストから分離する I/O デバイスとして PCIe SSD と GPU 用いて性能評価を行った．これらの I/O デバイスはホストと ExpEther ブリッジを用いてイーサネットで接続した．SSD の接続では，3.3 節で述べたリンクアグリゲーションの手法を用いて 2 本のイーサネットのパスで接続した．一方 GPU はイーサネットの PCIe の帯域が性能ボトルネックとならなかったため，1 本のイーサネットで接続した．

評価に用いた PCIe SSD は Intel SSD 910 シリーズである．Intel SSD は 2 つのストレージボリュームを保持するため，片方のみを評価に用いた．ベンチマークアプリケーションには fio を使用した．一方 GPU は Nvidia K20 を用いた．GPU のベンチマークアプリケーションには，グラフアルゴリズムの SSSP (Single-Source Shortest Path) を用いた．SSSP の実装の詳細は [63] で述べた．SSSP はグラフ上の 1 つの頂点から他の全頂点に対する最短経路を求める問題である．SSSP は繰返し計算を伴うアプリケーションであり，アルゴリズムが収束するまで各イテレーション毎に CPU と GPU の間で通信が

発生する．

図 4.3 にホストから分離した PCIe SSD の性能評価に用いた実験系を示す．評価のベースラインとして，PCIe SSD をホストの I/O スロットに直接挿入した場合の性能を取得した．これを図中 (a) ではローカルバスの性能としている．一方 (b) では SSD が ExpEther でホストから分離された場合の性能を取得した．ホストと SSD の間にイーサネットスイッチを挿入した場合の性能も測定し，通信遅延の増加が性能に与える影響も評価した．

図 4.4 は測定した SSD の IOPS(Input/Output Operations per Second) である．測定はランダムリード及びライトについて fio の iodepth を変化させて行った．またベンチマークのスレッド数は 1 に設定した．結果，リードでは SSD をホストから分離しても性能が低下しないことがわかった．一方ライトではホストと SSD の間にイーサネットスイッチを挿入した場合にオーバヘッドが拡大し性能が 14% 低下した．この結果からライトにおける性能オーバヘッドの原因はホストと SSD の間の通信遅延であると判断した．

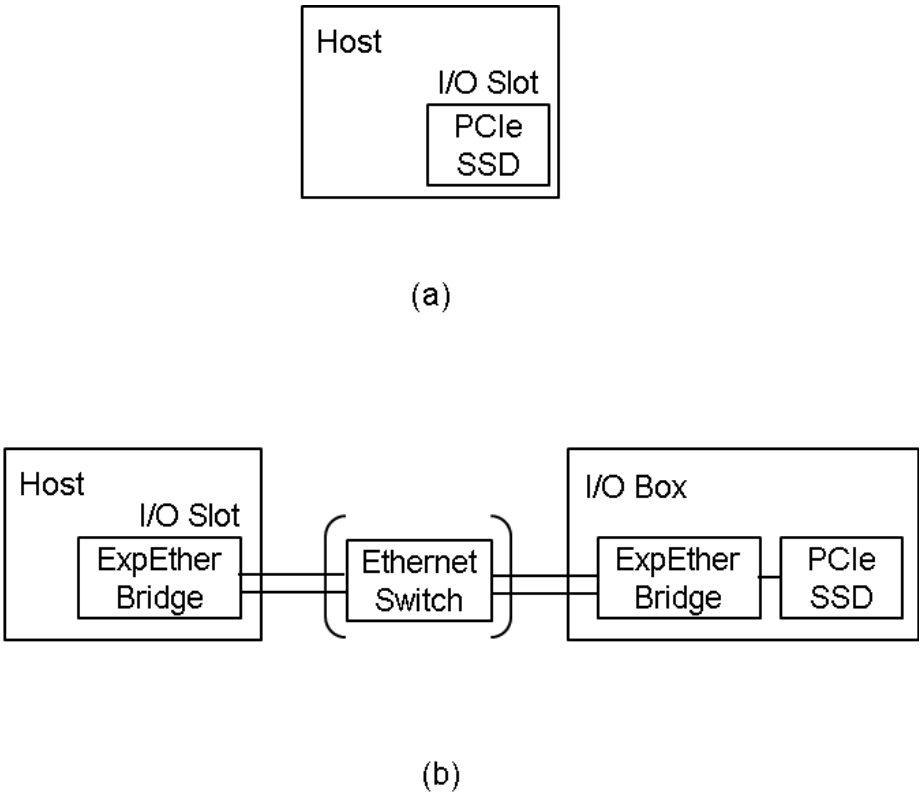
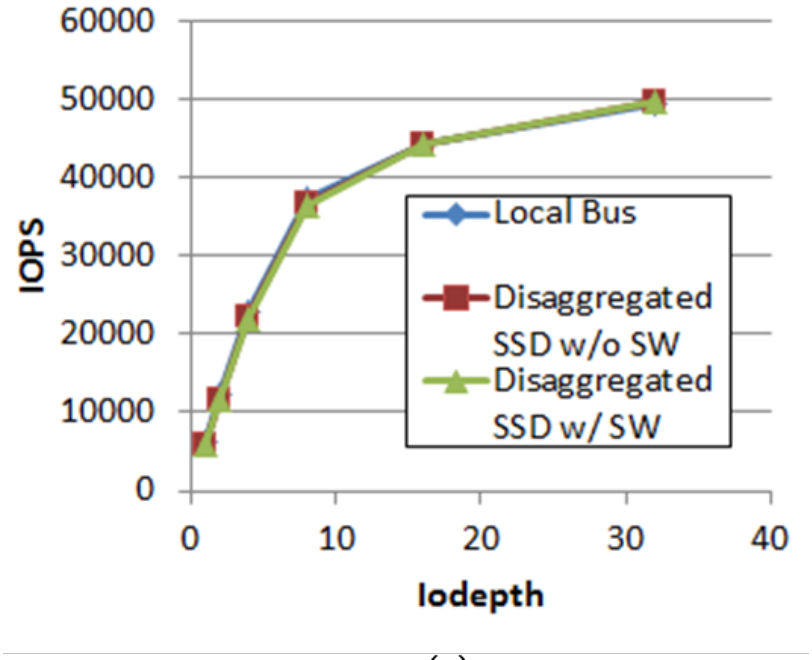
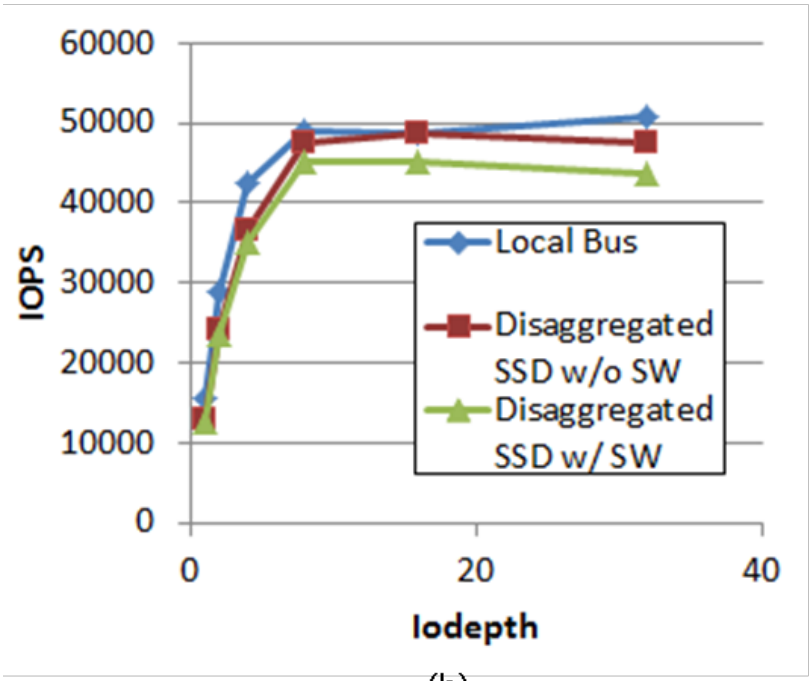


図 4.3　ホストから分離した PCIe SSD の性能評価．(a) ローカルバスに接続した SSD. (b) ホストから分離した SSD(スイッチ有り及び無しの場合).



(a)



(b)

図 4.4　ホストから分離した PCIe SSD の IOPS 性能．ブロックサイズは 4KB に設定．(a) リード．(b) ライト．

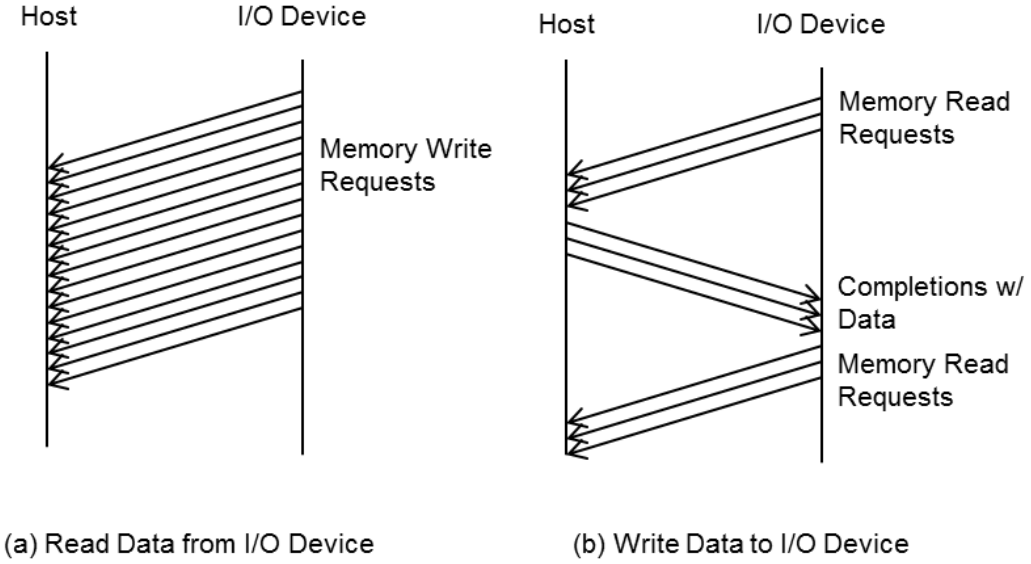


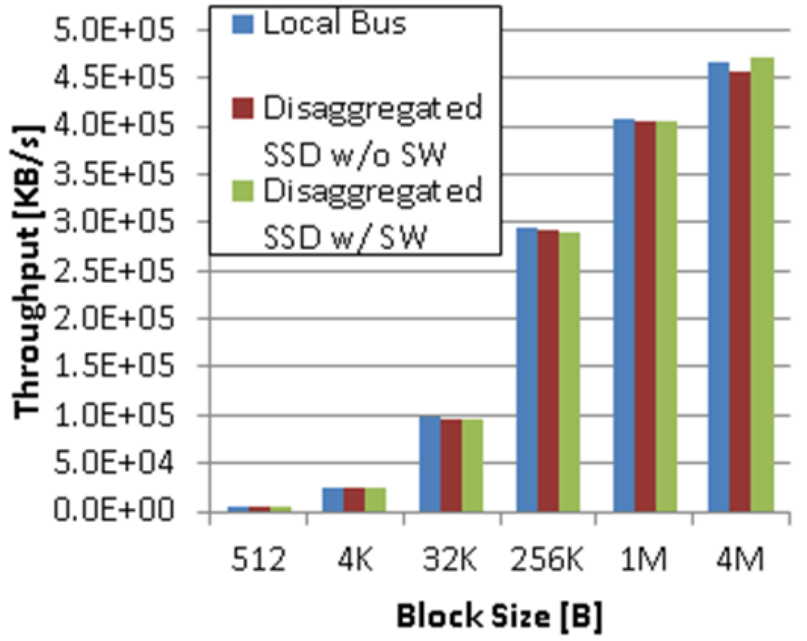
図 4.5　ホストと I/O デバイスの間の I/O トラフィック.

ここでライトの性能がリードより通信遅延の影響を受けやすい理由を，測定したホストと I/O デバイスの間の通信トラフィックの性質から説明する．図 4.5 にリードとライトのトラフィックの比較を示す．リードの場合，データが DMA (Direct Memory Access) によって I/O デバイスからホストのメインメモリに書き込まれる．PCIe ではデータの書き込みは Posted Request のトラフィックに該当する．Posted Request は図 4.5 に示す通り，I/O デバイスから連続して送信することが可能である．一方ライトの場合，データは I/O デバイスの DMA によってホストのメインメモリから読み出される．データのリードは PCIe では Non-Posted Request に該当する．この場合，I/O デバイスは要求したデータを保持する TLP が返ってきた後でなければ次の TLP を発行できない．また I/O デバイスは複数の Non-Posted Request を並列して発行することで通信遅延を隠ぺいし高い性能を実現できるように設計されている．しかし，ExpEther のようにホストと I/O デバイスの間の通信遅延が PCIe より増大する場合，ホストと I/O デバイスを接続するイーサネットを通信データで埋められなくなり，遅延が性能ボトルネックになる．よってこれらのリードとライトのトラフィックの相違がそれらの間で性能オーバヘッドが異なる理由である．

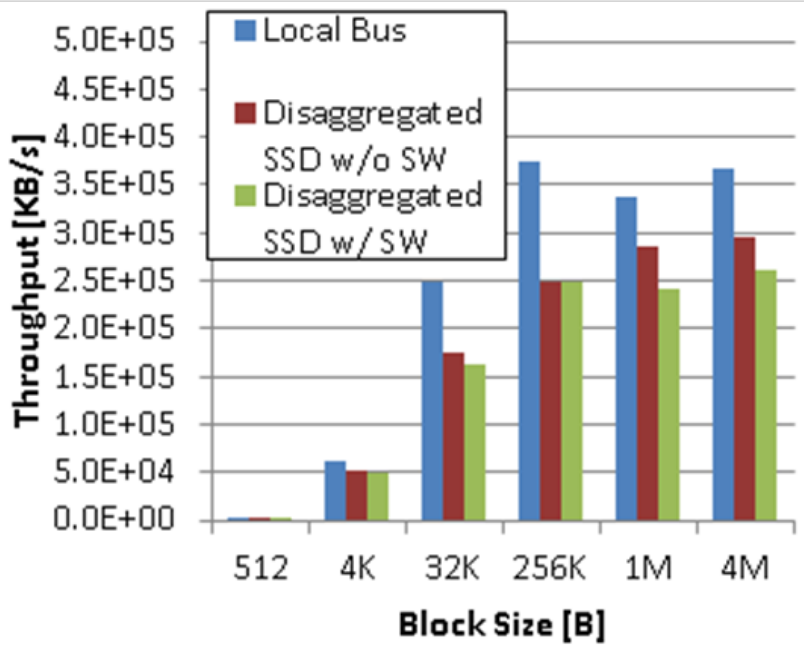
またライトの性能オーバヘッドは，I/O デバイスが並列にさらに多数の要求を送信するように設計すれば，低減することができる。あるいは ExpEther ブリッジの実装に今回の FPGA ではなく ASIC (Application Specific Integrated Circuit) を用いる事でも

通信遅延を短縮し性能を改善することができる．今回の FPGA によるプロトタイプ実装では，ExpEther ブリッジのユーザロジックの遅延はブリッジ全体の遅延の 47% である．回路を ASIC で実現する場合，ユーザロジックの動作周波数は FPGA に対して経験的に約 2 倍に増加できる．ここで回路の動作周波数を 2 倍に増加すると処理遅延は半分になる．従って，もしブリッジの 47% の領域の動作周波数を 2 倍に増加した場合，ExpEther ブリッジ全体の処理遅延を 24% 削減できる．

次に図 4.6 に PCIe SSD の帯域の評価結果を示す．fio のアクセスパターンはランダムリードまたはライトとし，アクセスするブロックサイズを変更した．また iodepth は 1 に固定した．帯域の評価でも IOPS の評価と同様，リードにおける性能オーバヘッドが殆どない一方，ライトにおけるオーバヘッドはブロックサイズが増加すると増大した．ライトのオーバヘッドによる性能低下の平均は 25% である．ここでブロックサイズが増加するほど性能オーバヘッドが増大する理由は，10GbE を 2 本集約してもイーサネットの PCIe の実効帯域が PCIe バスの帯域より小さかったため，帯域がボトルネックになったと考えている．ブロックサイズが大きい場合，ホストと I/O デバイスの間でより広帯域な通信が行われる．従って，イーサネットの帯域がボトルネックの場合，ブロックサイズが大きいほど大きいオーバヘッドの結果となった可能性がある．



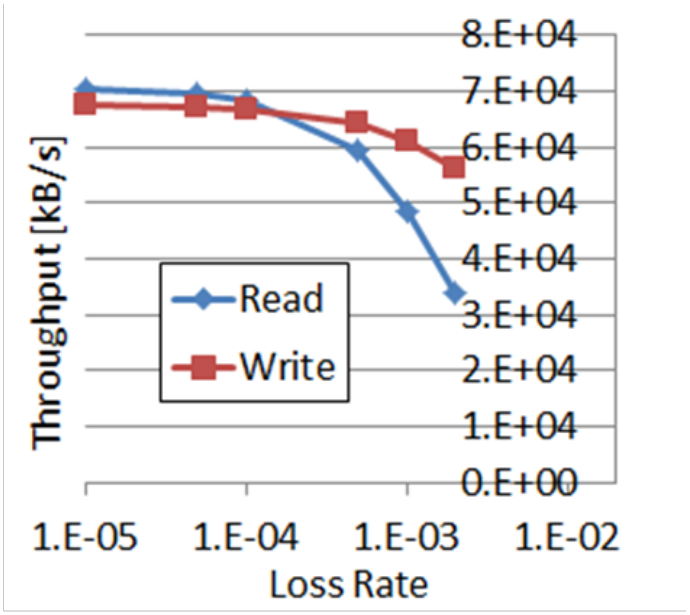
(a)



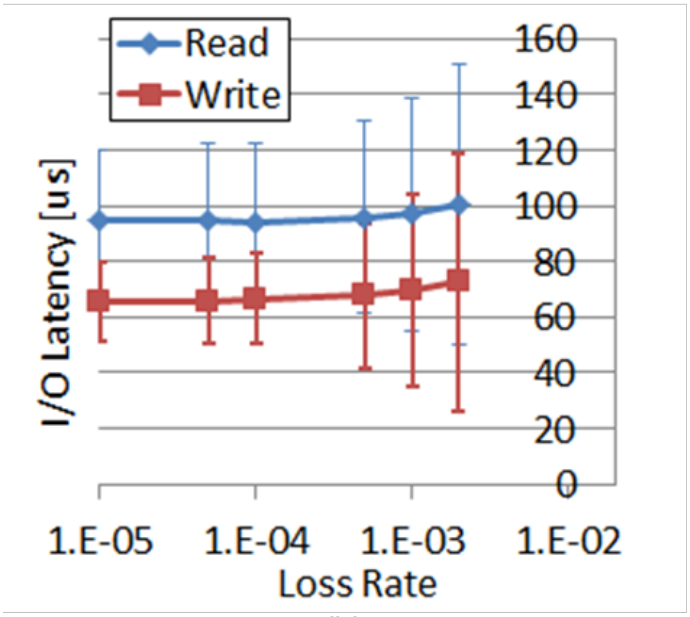
(b)

図 4.6　ホストから分離した PCIe SSD の帯域．fio の iodepth は 1 に設定．(a) リード．(b) ライト

次にイーサネットにおいて TLP をカプセル化したフレームが欠落した場合，接続した I/O デバイスの性能に与える影響を評価した．図 4.7 はフレームが欠落した場合の SSD の性能である．この評価ではホストと SSD の間にロスジェネレータを挿入した．ロスジェネレータは設定に従って通過パケットをランダムに廃棄する．本測定に限り，用いたロスジェネレータが 1GbE のインタフェースしか保持していなかったため，1GbE の ExpEther プロトタイプと Intel SSD750 を用いて評価を行った．ExpEther ではイーサネットでフレームの欠落が発生すると，3.3 節で述べた再送手法が欠落したフレームを再送する．図 4.7 が示す評価結果からロスレートが 0.4% 以上になると SSD が正常に動作しなくなることがわかる．ここで ExpEther ではフレームの欠落が連続して発生するとイーサネットのパスに障害が発生したとしてイベントを検知する設計としている．このしきい値は設定で変更でき，今回の測定では 3 に設定した．この値をチューニングすることで I/O デバイスが対応できれば条件の悪いイーサネットでも I/O デバイス分離システムに用いることが可能である．



(a)



(b)

図 4.7　イーサネットのフレームロスがホストから分離した SSD の性能に与える影響.
(a) 帯域. (b)I/O の遅延と標準偏差.

次に GPU を用いた評価結果を図 4.8 に示す．評価では GPU で SSSP アルゴリズムの計算を行った．この評価ではホストと GPU の接続にイーサネットスイッチを用いず,

GPU をホストに直接接続した．また評価では SSSP の演算を行うグラフの頂点数を変更することで様々な規模のグラフを評価した．図 4.8 に得られた結果は，I/O デバイスをホストから分離するための性能オーバヘッドの平均が 27% であることを示している．このオーバヘッドの原因は，SSSP の繰返し計算の間に発生するホストと GPU の通信の遅延が増加したためである．本処理の性能オーバヘッドを低減する方法として，先に PCIe SSD について述べた手法に加え，SSSP の実現アルゴリズムを変更することで CPU と GPU の間の通信回数を削減することが考えられる．

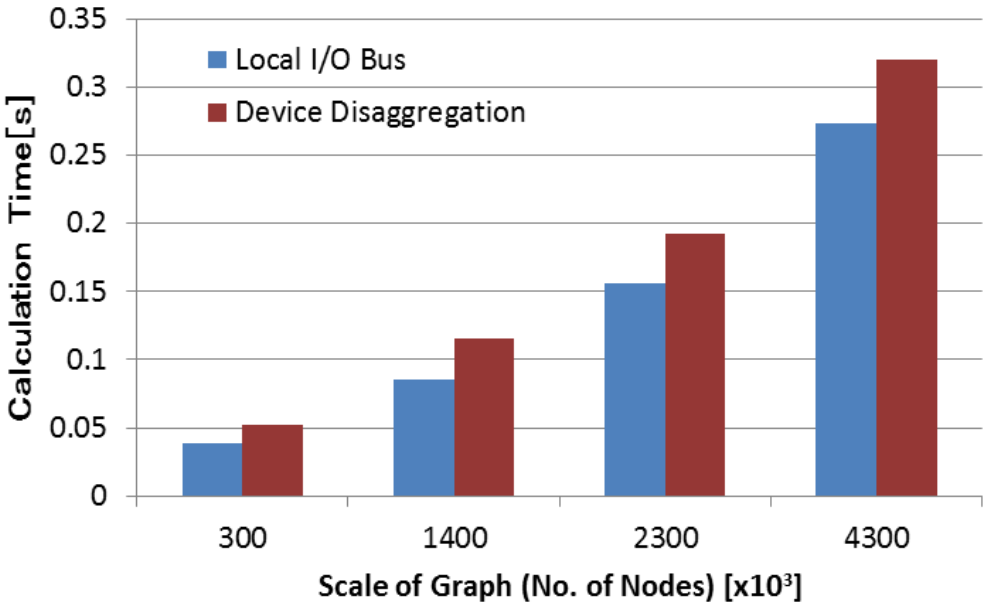


図 4.8　GPU を用いた SSSP アルゴリズムの計算時間

4.2.2 適応的な集約による PCIe 帯域向上の性能評価

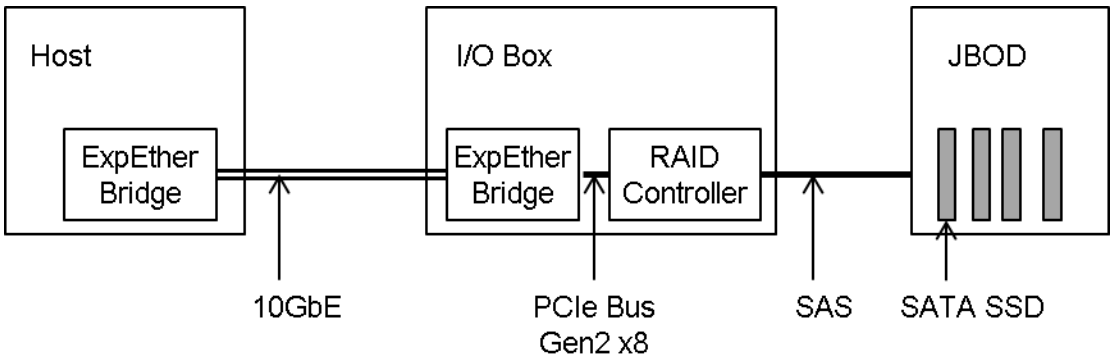


図 4.9　複数の TLP の適応的な集約の性能評価の実験系

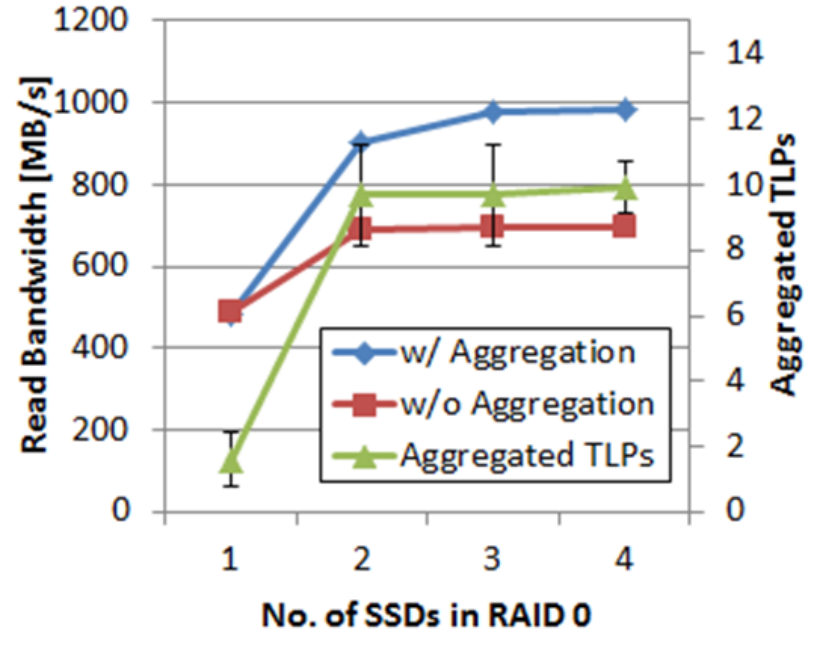
本節では，適応的な集約により複数の TLP を同一のイーサネットフレームにカプセル化した場合の I/O デバイスの性能を評価する．性能評価に用いた実験系を図 4.9 に示す．評価に用いたシステムの TLP の最大ペイロードサイズは 128B だった．I/O デバイスには RAID (Redundant Arrays of Inexpensive Disks) コントローラを使用した．この RAID コントローラを I/O リソースボックスに収容しホストと 10GbE で接続した．RAID コントローラは SAS (Serial Attached SCSI) で JBOD に接続した．JBOD には Serial ATA (SATA) SSD を 4 枚収容し，これらの SSD を用いて RAID-0 のボリュームを作成した．評価では RAID を構成する SSD の数を変化させ，様々な I/O 帯域の特性を持つ I/O デバイスの評価を行った．また適応的な集約に用いる ExpEther 内部の輻輳制御モジュールとして単純なレートリミッタを試作した．これは 3.3 節で述べた ExpEther に用いている遅延に基づく輻輳制御モジュールと異なるものである．このレートリミッタにより適応的な集約の基本的な機能の評価を行った．

図 4.10 は RAID-0 のボリュームを構成する SSD の数を変化させたときの I/O 帯域である．この評価ではレートリミッタによる帯域の制限は行っていない．RAID ボリュームを構成する SSD の数を増加させると，イーサネットの PCIe 帯域が I/O デバイスの性能のボトルネックとなる．評価結果は，イーサネットの PCIe 帯域が性能ボトルネックとなる場合，適応的な集約により PCIe 帯域が向上し性能ボトルネックが緩和することを示している．

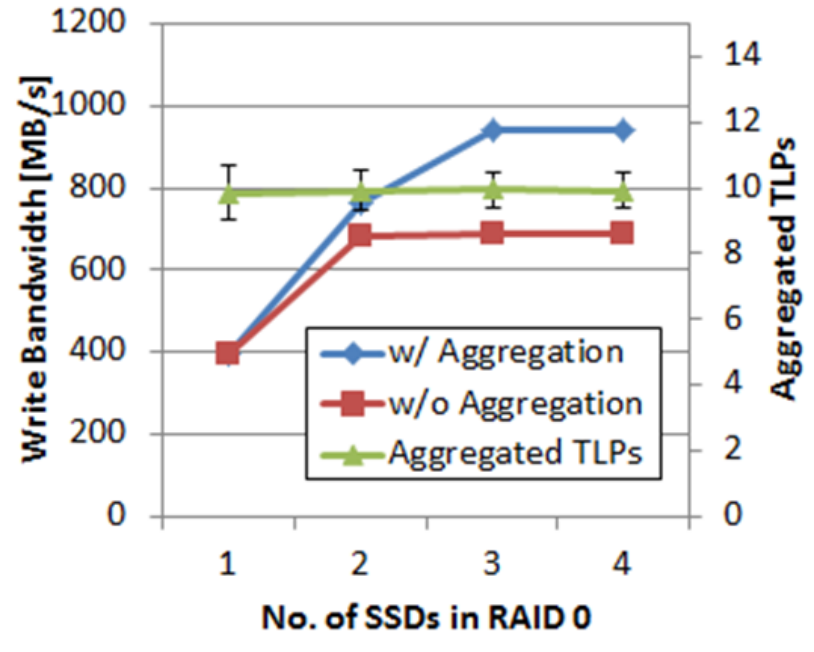
図 (a) のリードでは 1 枚の SSD を RAID に用いた場合は集約の有無で I/O 性能は変わらない．一方 2 枚以上の SSD を RAID に用いた場合，性能向上が確認できた．4 つの

SSD を用いた場合，性能改善は 41% である．図 4.10 にはイーサネットに集約する平均 TLP 数とその標準偏差も示した．PCIe の通信帯域が増大すると，イーサネットの PCIe 帯域を増加させるため TLP の集約が開始される．また，それぞれのフレームに集約される TLP の数は集約する瞬間の通信トラフィックに依存してバラつきがあることもわかる．4 つの SSD を用いた場合，集約される TLP の平均数はおよそ 10 個であり，1 つのイーサネットフレームに集約できる最大数に等しい．従って，最大 TLP が 128B のシステムでは 41% の性能向上が実現し得る最大の改善であると考えられる．

同様の性能改善は (b) のライトでも得られた．ただしリードとライトの通信トラフィックの相違により改善はわずかにライトの方が小さい．またライトではイーサネットの帯域がボトルネックとならない場合でも TLP の集約が確認された．ライトにおける最大の性能改善は 37% だった．



(a)



(b)

図 4.10　RAID-0 を構成する SSD の数を変更した場合の I/O 性能．複数の TLP が集約してイーサネットフレームにカプセル化される．(a) リード．(b) ライト．

別の評価では，RAID0 に用いる SSD を 1 個とし，レートリミッタを用いてイーサネットの帯域を変化させる評価を行った．その結果，レートリミッタでイーサネットの帯域を制限すると TLP が集約され始めることを確認した．この集約により，SSD の性能ボトルネックとなるイーサネットの PCIe の実効帯域が改善した．

次に，4 KB と 64 MB のランダムアクセスの I/O 要求の遅延を測定することで適応的な集約手法の TLP の通信遅延の増加を評価した．ここで 4 KB のアクセスではイーサネットの PCIe 帯域は I/O デバイスの性能ボトルネックとならない．一方 64 MB のアクセスではボトルネックとなる．fio のスレッド数は 1 に設定した．得られた測定結果を表 4.1 に示す．4 KB のアクセスでは I/O 要求の遅延は集約の有無で変わらなかった．この結果から適応的な集約は TLP の通信遅延を増加させないことがわかった．一方 64 MB のアクセスでは，適応的な集約がイーサネットの PCIe 帯域を改善するため，I/O 要求の遅延は改善した．

表 4.1　ランダムアクセスにおける I/O 要求の遅延 [us]

	Read	Write
4 KB w/ Aggregation	70.68	98.18
4 KB w/o Aggregation	69.5	98.21
64 MB w/ Aggregation	65180	65038
64 MB w/o Aggregation	91622	89993

これらの評価結果から，適応的な集約によりイーサネットの PCIe の実効帯域が接続する I/O デバイスの性能ボトルネックとなる場合，各イーサネットフレームに TLP が自動で集約され接続する I/O デバイスの性能を改善することを確認した．適応的な集約はイーサネットフレームに追加の通信遅延を付与しないため，イーサネットの帯域がボトルネックではない I/O デバイスは性能を劣化させないことも確認した．

4.3　本章の評価のまとめ

本章では第 3 章で提案したホストから I/O デバイスを分離する ExpEther のプロトタイプと，プロトタイプを用いて行った性能評価の結果を示した．作成したプロトタイプは FPGA を用いて実装し，ホスト側は I/O スロットに挿入する HBA，I/O デバイス側は I/O リソースプールを実現する I/O リソースボックスを試作した．試作した FPGA のインターフェースは PCIe Gen2 x8 と 10GbE を 2 ポートとした．

本章では 2 つの性能評価を行った．1 つ目の評価では ExpEther でホストから分離した I/O デバイスとして SSD と GPU を用いて性能評価を行い，デバイス分離の性能オーバヘッドを検証した．その結果，オーバヘッドは概ね 20% 程度と小さく，実システムに適用できる性能が実現できることを示した．また性能オーバヘッドは通信トラフィックのパターンにより，I/O デバイスにデータを送信する場合とデバイスから受信する場合で異なることがわかった．これらの評価では SSD と GPU という今日のデータセンターで一般に用いられている I/O デバイスをホストから分離した．これにより，提案手法はデータセンターで用いられている汎用 I/O デバイスに広く適用可能であることを示した．また 2 つ目の評価では，TLP の適応的な集約によるイーサネットの PCIe の実効帯域向上効果を基礎的なレートリミッタを用いて評価した．その結果提案手法はイーサネットの PCIe 帯域が I/O デバイスの性能ボトルネックとなる場合に TLP を自動で集約してカプセル化し，PCIe 帯域を向上させることがわかった．TLP の集約による PCIe 帯域の向上は最大 41% だった．

第 5 章

分離した複数の計算アクセラレータを用いた計算の最適化方式

第 3 章で提案したネットワークによる I/O (Input/Output) デバイス分離方式により，ホストに対する I/O デバイスの割り当てを必要な時に必要な数だけ行えるようになった．本章では，I/O デバイスの分離アーキテクチャにおいて I/O デバイスとしてアクセラレータである GPU (Graphics Processing Unit) を用いる場合に，GPU の演算リソースを効率的に使用し，用いる GPU 数に比例する処理性能が自動で得られる技術を提案する．その中でも特に，容量がホストより小さい GPU メモリに格納しきれない大容量のデータ処理を行う Out-of-Core 処理に着目した技術を提案する．

複数の GPU を用いた Out-of-Core 処理の課題は，GPU のメモリに格納しきれない大容量のデータを分割して GPU に入れ替えながら処理を行うためのデータスワップの I/O である．一般にホストのメインメモリと GPU メモリを接続する I/O バスは帯域が小さく，GPU 処理のボトルネックとなる．それに対し I/O デバイスの分離アーキテクチャではホストと I/O デバイスを接続するネットワークの帯域はホスト内の I/O バスよりさらに帯域が小さい．そのため GPU の演算リソースの利用が非効率となり，I/O デバイスの分離アーキテクチャで任意の数の GPU を割り当てられても，所望の処理性能が得られない．以下では Out-of-Core 処理の課題としてホストと GPU の間のデータ通信の帯域を論じる際，接続手段のホストの内部バスである I/O バスや，I/O デバイスの分離アーキテクチャにおけるネットワークを区別せず I/O バスとして論じる．

本章で提案する Victream と呼ぶミドルウェアは，複数の GPU を用いた Out-of-Core 処理において，性能ボトルネックとなる GPU のデータ I/O を自動で最小化する．Victream はデータパラレル処理を対象とする DAG (Directed Acyclic Graph) 型ミドル

ウェアであり，アプリケーションの処理を示す DAG を受け取り，その DAG の GPU での処理を最適化する．Victream は GPU へのデータ I/O を最小化するため，新しいスケジューリング方式を採用する．GPU を用いた計算が CPU (Central Processing Unit) を用いた計算と異なる点は，データ I/O のオーバヘッドを抑制するため，将来実行するタスクの入力データを先行して GPU メモリにロードするプリフェッチをソフトウェアの制御で行う点である．本章で提案する新しいスケジューリング方式は，GPU のデータプリフェッチを行いながら，GPU へのデータ I/O 量の最小化を同時に実現する．

以下本章では，5.1 節で Out-of-Core 処理の課題と Victream の目的，5.2.3 節で Victream の研究の動機，5.3 節で Victream のアーキテクチャ，5.4 節で Victream のスケジューラの詳細について述べる．

5.1 Out-of-Core 処理の課題と Victream ミドルウェアの目的

近年，GPU は高性能なデータパラレル処理を実現する計算アクセラレータとして注目されている．これまでにも大容量データ処理フレームワークである Spark[23] や機械学習フレームワークの TensofFlow[9] への適用が行われた．

また，単一の GPU を超えた処理能力を求める要求から，ホストやコンピュータクラスタで複数の GPU を用いる需要が高まっている．このような要求に応えるコンピューティングフレームワークとして DAG に基づくフレームワークがある．DAG 型フレームワークは複数の計算リソースに協調してデータ処理を行わせることで使用する計算リソースに比例した性能向上を実現する [31, 78]．本研究では特に，データパラレル処理を対象とした DAG 型フレームワークに着目する．GPU 処理のために提案されたデータパラレル処理向けの DAG 型フレームワークには PTask [52] がある．このようなフレームワークでは，ユーザプログラムの処理を示す DAG が作成される．作成されたグラフの頂点は処理するデータに適用するユーザ定義関数である．コンピューティングフレームワークは作成された DAG の頂点のタスクのデータパラレルの並列性を利用し DAG の処理を複数の計算リソースに分散して実行する．

しかし，複数の GPU を用いた Out-of-Core 処理ではデータパラレル処理向けのフレームワークの性能が低下する．何故なら，処理する大容量のデータを分割して GPU に入れ替えながら処理を行うデータスワップの I/O が性能ボトルネックとなるからである．GPU を用いた大容量のデータ処理では GPU のメモリ容量が 20 GB 程度と小さく，Out-of-Core 処理となりやすい．この GPU のメモリ容量は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の容量より 2 桁小さい．さらに，Out-of-Core 処理におい

て GPU メモリとホストのメインメモリでデータのスワップを行う際にデータが通過する I/O バスは PCI Express (PCIe) の場合，帯域が GPU のローカルメモリより 1 桁以上小さい．従って GPU で扱うデータ処理容量が増加し Out-of-Core 処理となった場合，I/O バスによる急激な帯域低下によりデータスワップが性能ボトルネックになる．

しかし，従来の GPU 向けフレームワークでは GPU の演算リソースを効率的に使用することに着目しており，Out-of-Core 処理においてデータスワップの I/O 量を最小化する手法に着目していない．

従来のフレームワークでは GPU の Out-of-Core 処理に関して 2 つの課題がある [52, 23]．1 つ目は，これらのフレームワークは DAG に含まれるタスクが実行可能になった順に First-In First-Out(FIFO) で実行することである．このようなスケジューリングは Out-of-Core 処理においてスケジュールされたタスクの入力データのローカリティを考慮しない．もしスケジュールされたタスクの実行待ちキューの先頭のタスクの入力データが GPU からスワップアウトされている一方，キューの中程のタスクの入力データが GPU メモリに存在していた場合，キューの先頭のタスクを実行することでキューの中程のタスクの入力データをスワップアウトすることになる．このように将来実行するタスクの入力データをスワップアウトすると，GPU のデータ I/O 量を増大させる原因となる．

2 つ目の課題は，GPU のデータプリフェッチとデータスワップの I/O 量最小化を両立できないということである．Out-of-Core 処理では，将来実行する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，演算と並行して将来使用するデータを GPU にロードするデータプリフェッチが有効である．データプリフェッチによってボトルネックリソースである I/O バスの帯域を有効に活用できるからである．ここでプリフェッチは将来実行するタスクの入力データのロードであるから，プリフェッチを行うためには演算リソースが空いてから次に実行するタスクをスケジュールするのではなく，タスクの演算と並行して将来実行するタスクのスケジュールを行う必要がある．ここで，従来のスケジューラでスケジューリング時に選択できるタスクの候補は DAG 内でタスクが依存する上流の親のタスクの実行が完了しているタスクである．しかし，5.2.2 節で論じるように従来の定義による候補の中から将来実行するタスクをスケジュールすると，データを出力するタスクと，そのデータを入力とするタスクの実行の間に他のタスクをスケジュールすることになる．その結果 GPU のデータスワップの I/O 量が増大する．

これらの課題のため，Out-of-Core 処理では GPU の演算リソースを効率的に使用できず，GPU 数を増大させても処理性能を効率的に向上させることができない．

そこで本研究ではこれらの課題を解決するため “Victream” と呼ぶフレームワークを提案する．Victream は Spark [78] のオペレータを拡張した DAG に基づくフレームワーク

である．Victream は複数の GPU を用いた Out-of-Core 処理において GPU のデータプリフェッチを行う事で I/O オーバヘッドを抑制すると同時に，GPU のデータスワップの I/O 量を最小化する．Victream では画像や行列等の様々なデータ形式のデータパラレル処理を実行できるように API (Application Programming Interface) の抽象化を行っている．

Victream のフレームワークは GPU のデータプリフェッチとデータスワップの I/O 量最小化を同時に実現するために 2 つの新しい手法を用いたスケジュール手法を採用する．1 つ目はタスクの入力データのローカリティを意識したローカリティアウェアスケジューリングである．Victream はタスクをスケジュールする場合，GPU メモリに既に入力データが存在し，発生させるデータスワップの I/O 量が最小のタスクを選択する．2 つ目はローカリティアウェアスケジューリングを，GPU がデータプリフェッチを行えるように拡張することである．そのため Victream は演算リソースが空いてからタスクをスケジュールするのではなく，演算の実行中に将来実行するタスクをスケジュールし，GPU がプリフェッチを行えるようにする．また将来のタスクをスケジュールするためにスケジュール可能なタスクの再定義を行う．

5.2 研究の動機

5.2.1 GPU メモリの容量制約と Out-of-Core 処理

表 5.1 NVIDIA Tesla P100 GPU 緒元 [44]

Single-precision Floating Point Performance	9.3 Tflops
Memory Size	16 GB
Memory Bandwidth	732 GB/s
I/O Bus	PCIe 3.0 x 16

一般に GPU のメモリ容量は小さく，GPU で大容量のデータ処理を行うと Out-of-Core 処理となる．表 5.1 に本論の執筆時にハイエンド GPU にあたる NVIDIA Tesla P100 の緒元を示す．GPU のメモリ容量は最大でも 20GB 程度以下であり，今日用いられているホストのメインメモリより容量が二桁小さい．

GPU で Out-of-Core 処理を行う場合，GPU とホストのメインメモリの間で I/O バスを介してデータのスワップを行う必要がある．これは多くの計算で性能ボトルネックとな

る．何故なら GPU からスワップアウトされたデータを GPU に戻すための帯域は GPU メモリ上のデータをアクセスする帯域の 10 分の 1 以下だからである．P100 の場合，表 5.1 を参照するとメモリ帯域が 732GB/s である一方，I/O バスは PCIe 3.0 x 16 であり帯域は 16GB/s である．

ここでデータスワップの性能ボトルネックを緩和するには 2 つの相互に補完的な対策を同時に行う必要がある．1 つ目はスワップに関する GPU のデータ I/O 量を最小化することである．また 2 つ目はボトルネックとなる GPU の I/O リソースを常に稼働させ，アイドル時間を作らないことである．ここで前者は従来の計算フレームワークでは注目されて来なかった．GPU 処理におけるデータスワップのデータ I/O 量の総和は計算を構成する複数のタスクを GPU で実行する順番で決まる．何故なら，データスワップを抑制し，GPU メモリが保持するデータをタスクの入力データとして再利用する程度がタスクの実行順に依存するからである．従って，データスワップに関する GPU のデータ I/O 量を最小化するタスクの実行順を実現するスケジューリングが重要になる．一方，後者の I/O リソースを稼働させ続ける観点は従来のフレームワークでも GPU のデータプリフェッチで実現されてきた．データプリフェッチでは GPU におけるタスクの演算と，将来 GPU で実行するタスクの入力データの GPU へのロードをオーバラップさせることで，ボトルネックである GPU の I/O リソースを常にデータスワップで稼働させる．

ところで従来の GPU を用いた一般的な計算では，GPU の Out-of-Core 処理で性能ボトルネックとなるデータスワップを最小化するスケジューリングの行う場合，その実現はアプリケーションプログラマに委ねられている．GPU を用いたアプリケーションプログラムの作成は CUDA (Compute Unified Device Architecture) を用いて行う．CUDA は NVIDIA 社が提供している並列計算のためのプラットフォームである．CUDA では，アプリケーションプログラマが GPU における演算の実行やデータ I/O の制御を行うプログラムを記述する必要がある．これは Out-of-Core 処理のように演算とデータ I/O の制御が複雑になる場合，プログラマにとって負担である．従って，これらの最良の制御がシステムプラットフォームとして自動で提供されることが望ましい．

5.2.2 従来手法のスケジューラ

従来手法のスケジューラには Out-of-Core 処理で GPU のデータスワップに関するデータ I/O 量を増加させる 2 つの課題がある．本節ではこれらの課題について述べる．

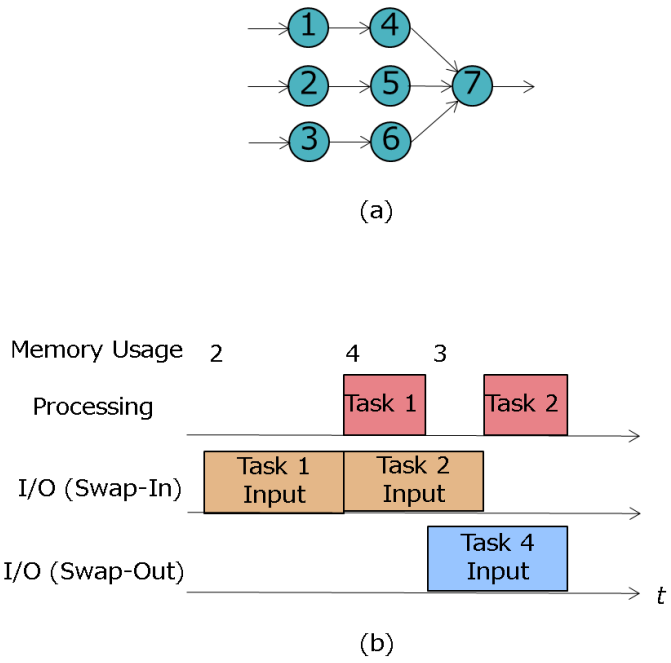


図 5.1　従来手法の DAG スケジューリング．(a)DAG. (b) タイムチャート．

FIFO タスクスケジューリング

従来の DAG コンピューティングフレームワークでは，タスクは DAG 内で依存する上流の親のタスクの実行が完了すると実行可能になる．図 5.1(a) の DAG の頂点はタスクを示している．この DAG ではタスク 1 が完了するとタスク 4 が実行可能になる．またタスク 7 はタスク 4-6 が完了すると実行可能になる．従来手法のスケジューラではタスクは実行可能になった順に実行待ちキューにキューイングされる．スケジューラはキューに積まれたタスクに FIFO で空いた GPU の演算リソースを割り当てることにより演算リソースを効率的に稼働させるようにする [52].

しかし，このようなスケジューリングは Out-of-Core 処理においてスケジュールされるタスクの入力データのローカリティを考慮していない．もし実行待ちキューの先頭タスクの入力データがスワップアウトされており，キューの中程のタスクの入力データが GPU メモリに存在する場合，キューの先頭のタスクを実行するとキューの中程のタスクの入力データがスワップアウトされる．この場合，将来実行するタスクの入力データをスワップアウトすることになり，スワップに関する GPU のデータ I/O を増大させる．

また，GPU に関して先行する研究である PTask のフレームワークでは入力データを考慮したスケジューリングを行っているが，Out-of-Core 処理はスコープ外である [52].

PTask のスケジューラは実行待ちキューの先頭で実行待ちのタスクの入力データの場所を考慮する．もし演算リソースが空いた GPU が，実行を待っているタスクの入力データを一番保持している GPU と異なれば，データを保持している GPU の演算リソースが空くまでしばらく待つ．しかし，実行を待っているタスクの入力データがスワップアウトされていた場合，GPU の入力データに関する優先度はどれも等しいため，演算リソースが空いた GPU にスケジュールされる．この場合，スケジューリング結果は前述の FIFO と同じになる．このようにキューの先頭のタスクの入力データがスワップアウトされる Out-of-Core 処理では，実行可能になった順にタスクをキューイングしたキューの先頭タスクの入力データだけを考慮するスケジューリングでは必ずしも GPU のデータ I/O を最小化しない．

データプリフェッチに起因するデータスワップ

従来手法のスケジューラでは将来実行するタスクの入力データをホストのメインメモリから GPU のメモリに予めロードするプリフェッチを行う．これにより GPU のデータ I/O とタスクの演算の実行をオーバラップし，GPU の計算の性能ボトルネックである I/O リソースの稼働を継続することで効率的なリソースの利用が実現できる．データプリフェッチは Out-of-Core 処理においてもデータスワップのために I/O リソースを効率的に利用する観点で重要である．

しかし，従来手法のデータプリフェッチでは，プリフェッチの結果将来実行するタスクの入力データをスワップアウトする．従って，スワップアウトされたデータを再び GPU にロードする必要が生じ GPU のデータ I/O 量が増大する．

この問題を図 5.1 を用いて説明する．説明の簡単化のため，図に示した DAG は単一の GPU で処理すると仮定する．また，タスク 1-6 の入力データ及び出力データの格納に必要なメモリ容量は 1 とし，GPU のメモリ容量は 4 とする．GPU はメモリ使用量が 3 に到達するとスワップアウトを開始する．GPU は双方向の I/O の並列実行が可能とする．

従来手法のスケジューラはプリフェッチを行うため，将来実行するタスクをスケジュールし，そのタスクの入力データをプリフェッチする．このとき将来実行するタスクとして選択するタスクはスケジューリングの時点で実行可能なタスクである．図 5.1 に動作例を示す．まず従来手法のスケジューラはタスク 1 の入力及び出力データのメモリ領域を確保し，タスク 1 の入力データのフェッチを行う．それが完了すると，スケジューラはタスク 1 の演算を開始する．同時に次に入力データのプリフェッチを行うタスクを探索する．このときスケジュールできる実行可能のタスクはタスク 2 及び 3 である．そのためスケジューラはタスク 2 のメモリ領域を確保しプリフェッチを行う．その間に，タスク 1 の処

理が完了する. ここでタスク 1 の入力データは今後他のタスクで使用しないため GPU メモリからスワップアウトせずに消去できる．すると GPU のメモリ使用は 3 となる．この段階で GPU のメモリ使用率はスワップアウトのしきい値を超えており，またタスク 2 に関するメモリ領域はロックされているため，タスク 4 の入力データがスワップアウトするデータとして選択される．しかしタスク 4 の入力データはタスク 4 を実行する際に GPU に再度ロードする必要があり，データスワップに関する GPU のデータ I/O 量が増大する．このように従来手法のスケジューラはデータプリフェッチを行うことで将来実行するタスクの入力データをスワップアウトし，データ I/O 量を増大させる．

上記は単純な例だった．しかし，データプリフェッチを行うために将来実行するタスクをスケジュールする場合，従来手法のようにスケジュール時点で実行可能なタスクから選択するとデータを出力するタスクと，そのデータを入力とするタスクの間に別のタスクの実行を挟む結果となる．上記の例ではタスク 2 がタスク 1 と 4 の間に挿入された．このようなスケジューリングは GPU のメモリ容量が制限されている場合，データスワップに関する GPU のデータ I/O を増大させる原因となる．

5.2.3　応用例: パンシャープン

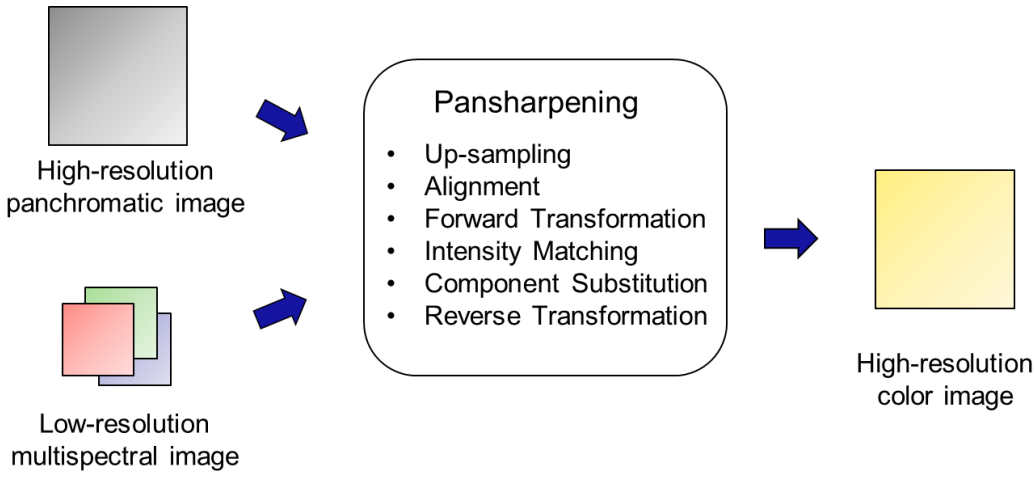


図 5.2　パンシャープン画像処理

Victream の応用の 1 つにパンシャープン画像処理がある．パンシャープン画像処理は高精度なパンクロ画像と低精度なマルチスペクトラル画像から高精度なカラー画像を生成する用途に用いられる．

パンシャープン画像処理の処理内容の大半はデータパラレル処理である．また，全体の

計算フローは DAG で表される．パンシャープン画像処理は衛星画像のような大型の画像処理に広く用いられている．従って複数の GPU をパンシャープン画像処理に適用し計算アクセラレーションを行うことが有効である．また GPU を用いる場合，GPU のメモリ容量の制限のため Out-of-Core 処理となることが多い．

パンシャープン画像処理の処理ステップを図 5.2 に示す．パンシャープンはアップサンプリング，アラインメント，前方変換，強度マッチング，代入，後方変換の 6 つのステップで構成される [47]．第 5 章で評価を行うパンシャープンの DAG は 25 のタスクから構成している．

パンシャープンには複数のアルゴリズムがある．あるアルゴリズムは GPU 演算のコストが低く，データスワップの Out-of-Core 処理が性能ボトルネックとなる．また別のアルゴリズムはイメージフィルタや行列演算等の異なるデータ形式の混合処理を行う必要がある．この場合，行列演算では画像のピクセルが行列の要素として扱われる．Victream は 5.3 節で後述するように GPU による様々なデータ形式のデータパラレル処理をサポートしている．従って，Victream を使用するパンシャープン画像処理の開発者は複数のアルゴリズムを Victream の単一のフレームワークを用いて試すことができる．

5.3 Victream のアーキテクチャ

5.3.1 概要

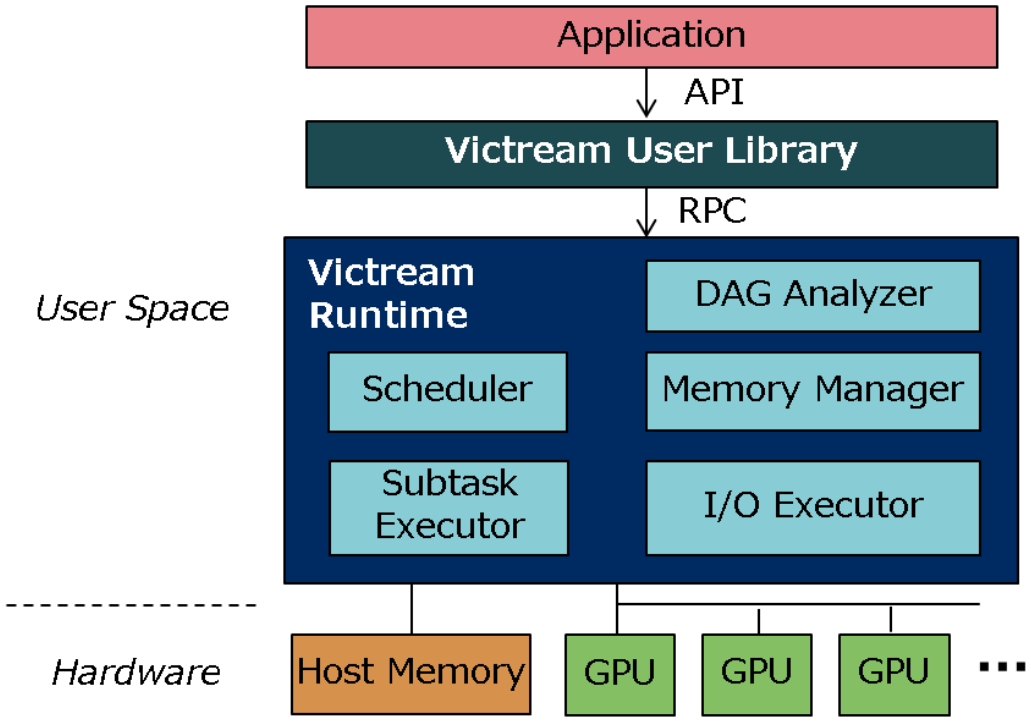


図 5.3 Victream のアーキテクチャ

本節では Victream のアーキテクチャについて述べる．またスケジューラの詳細は次節で述べる．Victream は Spark のオペレータ [78] を拡張し様々なデータ形式のデータパラレル処理を GPU で実行できるようにする．しかし本研究では Spark 自体の実装を拡張する方法はとらない．なぜなら Spark が備える CPU による計算と GPU の計算を組み合わせたヘテロジニアス環境の計算のスケジューリングは本研究の対象外だからである．そこで本研究のプロトタイプ試作では GPU 計算に特化したミドルウェアを C++ で実装した．

Victream ではアプリケーションの処理を示す DAG をアプリケーションプログラムが作成し，その DAG をミドルウェアが実行する．図 5.3 に示すように，Victream のアーキテクチャはユーザ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の作成ではライブラリが提供する API を用いてコーディングを行う．このときアプリ

ケーションプログラム内で API の関数を呼ぶとライブラリの内部で DAG が生成される．そしてライブラリを介して RPC (Remote Procedure Call) でランタイムを呼び出すと，ランタイムによる DAG の実行が始まる．DAG の処理はホストの I/O スロットに挿入した複数の GPU か，ExpEther を用いたシステムではイーサネットで接続した GPU を用いて行う．

Victream では DAG の頂点でユーザ定義関数を用いてデータパラレル形式で処理されるデータオブジェクトを GRDD (GPU Resilient Distributed Dataset) と呼ぶ．またユーザ定義関数を実行する DAG の頂点の処理をタスクと呼ぶ．GRDD はランタイムによってデータパーティションに分割され，タスクは各 GPU でデータパーティションを単位として実行される．データパーティション毎に分割されたタスクをサブタスクと呼ぶ．Victream は GRDD にデータ形式の属性を付与し，その属性に依存したデータパーティションの管理手法を提供する．現在 Victream がサポートしているデータ形式は画像，密行列，疎行列，キーバリューペアである．また GPU のデータ I/O と演算のスケジューリングは全てのデータ形式を共通に扱う層で実現しており，様々なデータ形式の DAG や複数のデータ形式が混在する DAG の計算のスケジューリングを最適化できる．

GRDD のデータパーティションは GPU メモリに存在するか，GPU メモリからスワップアウトされ，ホストのメインメモリに存在する．データパーティションの管理とそれに対する演算の実行は Victream のフレームワークで行われ，アプリケーションプログラムには透過である．これにより，これまでアプリケーションプログラマの負担となってきたデータの演算の管理がシステムプラットフォームで提供される．またデータをパーティションすることで，Victream では生成する GRDD の総和が使用する GPU のメモリ容量より大きい Out-of-Core 処理に対応することができる．アプリケーションプログラムからはデータの容量に注意する必要はない．今後の検討として，データパーティションを NVM (Non-Volatile Memory) Card のようなストレージデバイスに格納する方向性がある．

以下の節では Victream のユーザライブラリとランタイムについて述べる．

5.3.2 Victream ユーザライブラリ

表 5.2 GRDD オペレータの例

Transformations	GRDD::map(f, MM) GRDD::zip($GRDD$) GRDD::mapPartitions(f, MM) GRDD::groupByKey(MM) GRDD::reduceByKey(f, MM) GRDD::join(f, $GRDD$, MM)
Actions	GRDD::reduce(f_{gpu}, f_{cpu}, MM) GRDD::outputFile($filePath$)

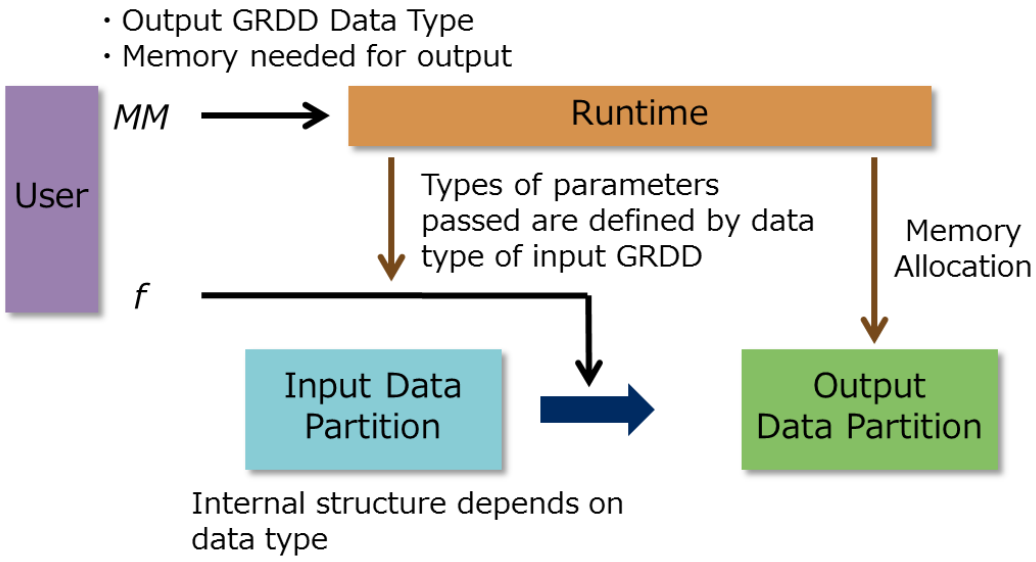


図 5.4 データパーティションの処理

Victream のユーザライブラリはアプリケーションプログラムに取り込まれて使用される．ユーザライブラリの内部ではアプリケーションプログラムから呼び出される API によって処理のデータフローを示す DAG を生成し，生成後にランタイムに要求して遅延実行を行う．Spark の RDD と同様，ユーザ定義関数の GRDD への適用は GRDD のオペレータを用いて行う．GRDD オペレータは GRDD のメソッドとして定義されている．オペレータを呼ぶと，ライブラリ内で生成される DAG に新たな頂点が追加される．表 5.2 にオペレータの一例を示す．

遷移オペレータ (Transformation Operator) は GRDD にユーザ定義関数を適用し，得られた出力の GRDD を返り値として返す．遷移オペレータの呼び出しによりライブラリ内で生成している DAG に頂点が追加される．また実行オペレータ (Action Operator) が呼ばれるとライブラリで生成された DAG の実行が RPC でランタイムに依頼される．

GRDD オペレータの変数 f はユーザ定義関数のクラスを示す．f は CUDA で記述され，GRDD の各要素に適用されるカーネル関数を内包する．この f はアプリケーションプログラマによって与えられる．それに対しユーザ定義関数の実行で生成される GPU スレッド数は Victream のランタイムで管理される．一方変数 MM は Method Meta-Information を意味し，f を実行するために必要なメタ情報をアプリケーションプログラマが指定する．MM が保持する情報の例は出力される GRDD のデータ形式，出力される GRDD の各データ要素のデータ形式，出力される GRDD のために確保すべきメモリ容量である．またオペレータの種類によっては出力 GRDD のメモリ容量はランタイムで自動で計算できるため省略することが可能である．

f がランタイムに呼ばれ，GPU で実行される際にランタイムから f に渡されるパラメータは処理される GRDD のデータ形式に依存する．GRDD には画像や密行列等のデータ構造があり，それに基づいたデータパーティションの管理がランタイムで行われる．これにより，様々なデータ形式のデータが共通のフレームワークで扱えるようになる．図 5.4 にこの手法を図示する．ランタイムから f に渡されるパラメータには，各 GPU スレッドが処理を担当するデータ要素を見つけるために必要なパラメータを含む．

一例として，GRDD が画像であり，GRDD にステンシル処理を行う場合について述べる．アプリケーションプログラマは MM に各 GPU スレッドがステンシル計算で参照する隣接要素数を指定する．Victream は GPU メモリに MM で指定された要求以上の冗長領域を保持するデータパーティションを用意する．またステンシル計算を内包する f を実行する GPU スレッドは，ランタイムから処理するデータパーティションの画像先頭からのオフセットを受け取る．この情報により画像全体の中で端のピクセルを担当するる GPU スレッドも適切に処理を行うことができる．

また別の例として，GPU 向けの汎用ライブラリで定義された関数を Victream のフレームワークで再利用する場合を述べる．ここで汎用ライブラリ関数の再利用を実現するため，Victream では GPU で実行するカーネル関数ではなく，ホストの CPU で実行するカーネル関数を内包する別の f を定義している．この CPU で実行するカーネル関数は内部で GPU ライブラリの関数を呼ぶ．それにより，呼び出された GPU の汎用ライブラリの関数の処理が GPU で実行される．例えば処理する GRDD が疎行列である場合を考える．データパーティションは分割されたブロック行列である．f は内部に CPU か

ら cuSPARSE の関数を呼び出すカーネル関数を内包している．これにより cuSPARSE の関数が入力 GRDD のデータパーティションに適用される．処理するデータパーティションのメモリ領域へのポインタは f を実行する際にパラメータとしてランタイムから渡される．また cuSPARSE の関数に渡す他のパラメータも f が呼ばれる際に渡される．Victream では f が呼ばれる際にランタイムから渡されるパラメータは入力 GRDD のデータ形式に依存する．このような方法で GRDD のデータ形式毎の対処をフレームワークから提供することで，Victream では Thrust, cuSPARSE, cuBLAS の関数の再利用を実現している [45].

次にアプリケーションプログラマが Victream のフレームワークを使用する手順を説明する．プログラマはユーザプログラムに Victream ライブラリを取り込んでコードの作成を行う．始めに処理する GRDD オブジェクトをインスタンス化する．インスタンス化時にはデータが格納されているファイルのパスをパラメータとして指定する．また GRDD のメタ情報も必要である．このメタ情報により，GRDD のデータ形式，各データ要素のデータ形式，GRDD オブジェクトの分割方法などが指定される．もし GRDD が画像であれば，画像の横幅や縦幅の幾何情報を指定する．インスタンス化した GRDD にオペレータを適用し，ユーザ定義関数をオペレータの引数として渡すとライブラリの内部で DAG が形成される．そして実行オペレータを呼ぶとそれまでにライブラリ内部で生成した DAG の計算の実行が RPC でランタイムに依頼される．

次に表 5.2 に示した GRDD オペレータの概要を説明する．

map: ユーザ定義関数 f が入力 GRDD の各要素に適用され，出力 GRDD が生成される．GRDD のデータの要素数，その順番，データパーティションは入力と出力の GRDD の間で保存される．

zip: GRDD は引数として与えられた別の GRDD と結合される．返り値の GRDD は 2 つの GRDD が関係付けられているという情報を保持する．応用例として，zip の出力 GRDD に map を適用し 2 入力の map 処理を行う場合がある．

mapPartitions: map と類似するが，データの要素数が入力と出力パーティションの間で保存されない．一方データのパーティションは保存される．

groupByKey: 入力 GRDD の各データ要素がキーバリューペアであることを想定している．同一のキーを保持するペアを同一のデータパーティションにグループ化した GRDD を出力する．

reduceByKey: groupByKey と同様に入力 GRDD の各データ要素がキーバリューペアであることを想定している．同じキーの値を共有するペアは引数として与えられたユーザ定義関数 f によりリダクションされる．

join: 入力 GRDD のデータ形式が行列であることを想定している．引数として与えられる行列との行列積を計算する．
reduce: 入力 GRDD を単一の値へリダクションする．出力される値はユーザプログラムに RPC の返り値として返される．始めに，f_{gpu} を用いてそれぞれのデータパーティションのリダクションを行う．f_{gpu} は GPU で実行されるカーネル関数を内包している．次に f_{gpu} の出力を f_{cpu} を用いてリダクションする．f_{cpu} は CPU で実行されるカーネル関数を内包する．Victream ライブラリでは一般的に用いられるリダクション関数を予め用意している．また Thrust [45] のライブラリ関数を f_{gpu} に再利用することが可能である．
outputFile: GRDD をホストのファイルシステムに格納する．ファイルパスを引数として与える．

5.3.3 Victream ランタイム

Victream のランタイムは図 5.3 に示す要素から構成される．DAG アナライザはユーザライブラリから受信した DAG の解析を行う．DAG アナライザは DAG が含むそれぞれのタスクについて GPU で処理するデータパーティション毎にサブタスクを生成する．サブタスクはユーザプログラムにおいて GRDD オペレータに渡されたユーザ定義関数をデータパーティションに対して実行する．ここでタスクの種類によっては 1 つのタスク内で複数のステップのサブタスクを生成することがある．例えば reduceByKey は第一ステップのサブタスクで GRDD が含むレコードをキーの値でソートし，第二ステップのサブタスクで同一キーが格納されたデータパーティションのレコードをリダクションする．

DAG アナライザは受信した DAG に対し生成したサブタスクの実行をスケジューラに依頼する．ここでタスクの種類によっては DAG の受信時にサブタスクの数が決まらない場合がある．再度 reduceByKey を例にとると，リダクションで実行されるサブタスクの数は入力 GRDD に含まれるキーのカーディナリティに依存する．このようなタスクに対するサブタスクの生成では，DAG アナライザは DAG の上流のタスクの完了を待ち，タスクに対して生成すべきサブタスクの数が明らかになった後サブタスクの生成を行う．

スケジューラは GPU におけるサブタスクの演算とデータ I/O のスケジューリングを行う．スケジューラの詳細は 5.4 節で述べる．

メモリマネージャは GPU メモリと，データをスワップアウト先であるホストのメインメモリのリソース管理を行う．このためメモリマネージャはスケジューラから GPU メモリを割り当てる要求を受ける．またスケジューラに GPU メモリの使用率を知らせる．

I/O エクセキュータはスケジューラから呼びだされ，GPU メモリに対するデータパーティションのスワップインやスワップアウトなど GPU のデータ I/O の実行を行う．

サブタスクエクセキュータはスケジューラから呼びだされ，GPU におけるサブタスクの演算の実行を行う．

5.4 スケジューラの詳細

5.4.1 スケジューリング手法

Victream を含む DAG 型フレームワークの目的は，GPU で実行する DAG が与えらた場合に最短の計算時間で実行を完了することである．つまり最短の計算時間で計算を実行できるように GPU での実行を最適化することである．ここで DAG が含むサブタスクはアプリケーションで定義されるため，Victream のフレームワークは各サブタスクの実行時間を知ることはできない．また，異なる種類のサブタスクを実行した場合にどのサブタスクが早く完了するかを知ることもできない．従ってスケジューラは DAG を実行する前に最適なスケジュールを計算することはできない．そこで Victream 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する．

Victream では，GPU を用いた Out-of-Core 処理における性能ボトルネックは PCIe バス，あるいは ExpEther の場合は PCIe をトンネリングするイーサネットの I/O 帯域であると想定している．実際にこれらの I/O バスの帯域が計算の性能ボトルネックであることは第 6 章の評価で確かめる．従って，計算時間を最短とするには，DAG の計算の完了までに GPU が実行する，全ての I/O 処理の完了時間を最小化する必要がある．これが Victream スケジューラの目的関数である．正確には，計算に複数の GPU が用いられるため，GPU の間で最後のデータ I/O を行う GPU の I/O 処理の完了時間が目的関数であり，それを最小化するのが目的である．

GPU のデータ I/O の完了時間を最小化するには，2 つの課題を解く必要がある．1 つ目は GPU のデータ I/O の総量を最小化することである．また 2 つ目は性能ボトルネックである GPU の I/O リソースを常に稼働させ続けることである．前者は従来手法のスケジューラではスコープ外だった．一方後者は従来手法のスケジューラでもデータプリフェッチで実現されていた．従って本研究では 1 つめの課題を解決すると同時に，2 つ目のデータプリフェッチも同時に実現する．前提条件として，DAG が保持するサブタスクの実行時間は予め知ることができないが，入出力データパーティションのサイズはユーザライブラリの API の引数で渡された情報と，オペレータの種類から知ることができると

する．

Victream は GPU のデータプリフェッチとデータ I/O 量の最小化を同時に実現するために 2 つの手法を用いる．1 つ目の手法は，ローカリティアウェアスケジューリングである．Victream はデータスワップに関する GPU のデータ I/O を最小化するためにスケジュールするサブタスクの選択に貪欲法を用いる．スケジューラがサブタスクをスケジュールする場合，スケジュール候補のサブタスクの中から GPU のメモリに存在するデータを再利用し，発生させるデータスワップのデータ量が最小のサブタスクを選択する．このときデータスワップのデータ量は GPU に新たにロードする入力データのデータ量と，GPU からホストのメインメモリにスワップアウトするデータ量の和で表される．

2 つ目の手法では，Victream は GPU がデータプリフェッチを行えるようにローカリティアウェアスケジューリングを拡張する．データプリフェッチとは将来実行するサブタスクの入力データを予めロードすることである．そのためデータプリフェッチを行うには将来実行するサブタスクを予めスケジュールする必要がある．そこで Victream スケジューラはローカリティアウェアスケジューリングを将来のサブタスクのスケジュールが行えるように拡張する．

ローカリティアウェアスケジューリングの拡張ではスケジュール候補となるサブタスクの再定義を行う．まずその議論の準備として，将来実行するタスクをスケジュールすることによりプリフェッチを行う際の動作を図 5.5 を用いて説明する．スケジューラは将来実行するサブタスクのデータプリフェッチが行えるように，サブタスクエクセキュータが実行する演算と非同期に，将来実行するサブタスクのスケジュールを行う．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ローカリティアウェアスケジューリングを用いて将来実行するサブタスクを選択する．ここで選択したサブタスクがデータプリフェッチを必要とする場合，つまり選択したサブタスクの入力データのうち 1 つ以上のデータパーティションがホストのメインメモリか別の GPU に存在する場合，スケジューラは I/O エクセキュータに入力データのプリフェッチを依頼する．逆に選択したサブタスクの全ての入力データが GPU メモリに存在する場合，I/O エクセキュータの呼び出しは省略され選択したサブタスクを実行待ちキューにキューイングする．ここで選択したサブタスクの入力データを生成する親のサブタスクが GPU の実行待ちキューで実行待ちであり，まだ生成されていない場合も，その入力データはすでに GPU メモリに存在するデータとして扱う．実行待ちキューにキューイングされたサブタスクはサブタスクエクセキュータによって順に実行される．またスケジューラは I/O エクセキュータに依頼した I/O が完了後，直ちに次の将来のサブタスクのスケジューリングに移る．このようにサブタスクの演算とは非同期に将来実行するサブタスクのスケジューリングと入力データのプリフェッチを継

続することで性能ボトルネックの I/O リソースを常に稼働させ，リソースを効率的に使用する．

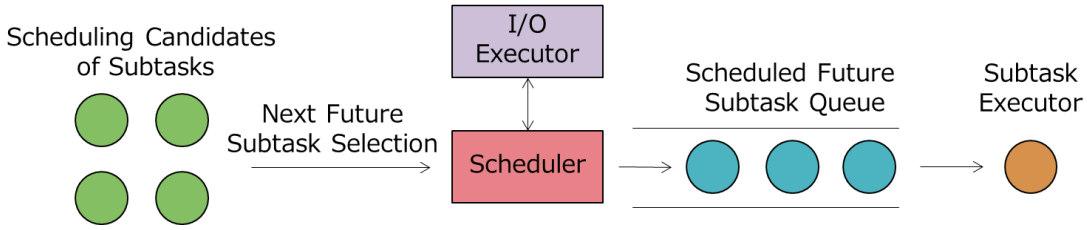


図 5.5　データプリフェッチを行うためのローカリティアウェアの拡張

次に，スケジュール候補のサブタスクの再定義を行う．従来手法のスケジューラではスケジューリング候補は DAG 内で依存する親のサブタスクの実行が完了したサブタスクだった．しかし，そのような定義では 5.2.2 節で述べたようにデータを出力するサブタスクと，そのデータを入力とするサブタスクの間に他のサブタスクをスケジュールし，GPU のデータ I/O の総量が増大する．

そこでこの課題を，スケジューリング候補のサブタスクに新たなサブタスクを追加することで解決する．追加するサブタスクは未完了の依存している上流のサブタスクが全て図 5.5 に示した同じ GPU の実行待ちキューにキューイングされているサブタスクである．このサブタスクが追加できる理由は，実行待ちキューのサブタスクはシリアライズして実行されるため，スケジュール候補のサブタスクがスケジュールされた場合，そのサブタスクが実行されるまでに依存する上流のサブタスクは実行が完了していることが保証されるからである．この拡張を基に，図 5.6 に示すようにスケジュール可能なサブタスクの再定義を行った．新たな定義では，従来手法のスケジューラのように依存する親のサブタスクが全て完了しているサブタスクは GPU にスケジュール可能である．また，未完了の親のサブタスクが全て同じ GPU の実行待ちキューに将来実行するサブタスクとしてキューイングされている場合もスケジュール可能である．ここで前者のサブタスクは全ての GPU にスケジュール可能であるのに対し，後者は未完了の親のサブタスクが実行待ちキューで実行を待っている GPU へのみスケジュール可能である．

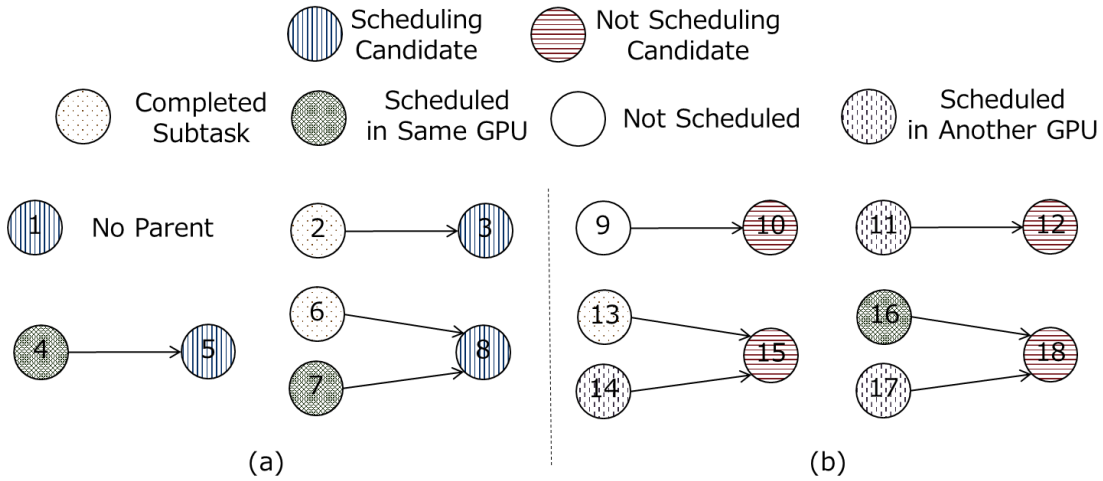


図 5.6　Victream スケジューラのスケジューリング候補と親のサブタスクとの関係.
(a) スケジューリング候補. (b) スケジューリング候補ではない.

スケジュール候補の再定義で新たにスケジュール候補として組み入れたサブタスクの基準として 2 つの判断基準を用いた．1 つ目の基準は，既に生成されているサブタスクの入力データパーティションの内，ホストのメインメモリか他の GPU にスワップアウトされているデータパーティションはスケジュール時点でプリフェッチが行えること，また 2 つ目の基準は，サブタスクが依存する DAG の上流の全ての親のサブタスクがそのサブタスクを実行するまでに完了していること，である．2 つ目の基準を満たすには，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 の実行待ちキューに既にキューイングされている必要がある．

このように提案手法ではローカリティアウェアスケジューリングを GPU がプリフェッチを行えるように拡張し，GPU のデータスワップに関するデータ I/O 量の最小化と GPU のデータプリフェッチを同時に実現する．提案手法は 5.2.2 節で述べた課題だったデータを出力するサブタスクと，そのデータを入力とするサブタスクの間に他のサブタスクをスケジュールすることをスケジュール候補の再定義により防ぐ．これにより，提案手法は GPU がプリフェッチを行ってもデータ I/O 量が最小となるスケジュールを実現する．

5.4.2　スケジューラの実装

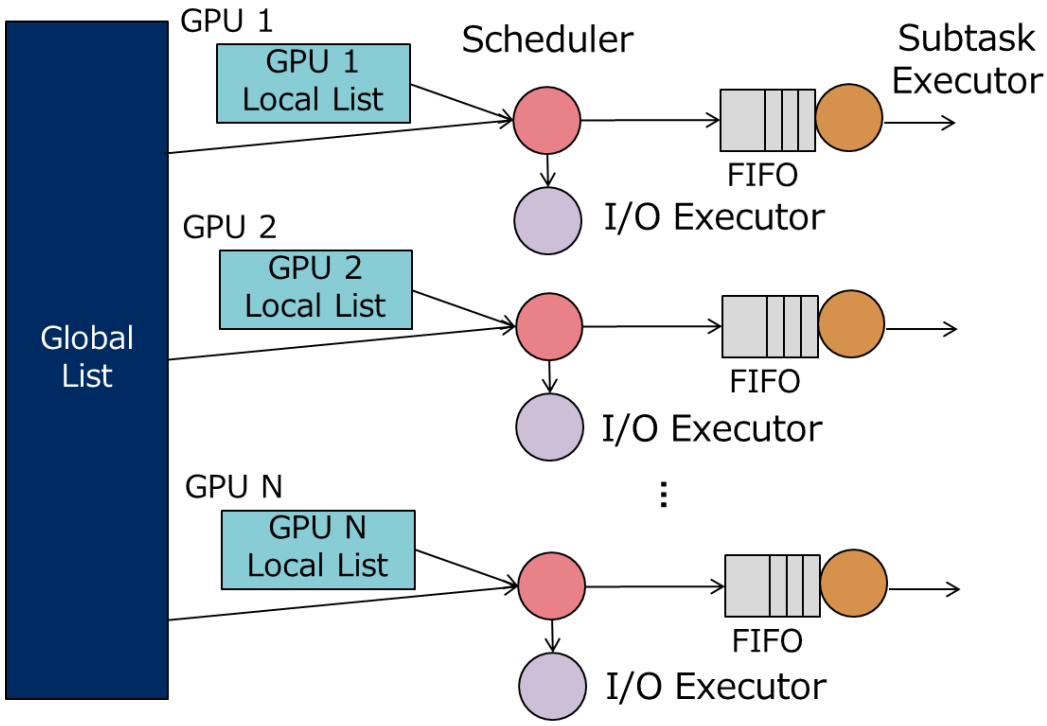


図 5.7　スケジューラの構成

本節では 5.4.1 節で述べたスケジューリング手法の実装について述べる．Victream スケジューラの構成を図 5.7 に示す．スケジューラは 5.4.1 節で述べた手法を個別の GPU に対して用いることで，GPU 毎に設けた I/O エクセキュータとサブタスクエクセキュータを制御する．

Victream のスケジューラは GPU が将来実行するサブタスクの入力データのデータプリフェッチを継続できるようスケジューリングを行う．将来実行するサブタスクをスケジュールする際，GPU メモリ上に入出力データのメモリ領域をロックする．またスケジュールされたサブタスクの入力データパーティションが GPU メモリ以外に存在し，GPU へのデータプリフェッチを必要とする場合，I/O エクセキュータにデータをプリフェッチするように依頼する．そして I/O エクセキュータへの依頼が完了するまで次のサブタスクのスケジュールをブロックする．データプリフェッチが完了するとプリフェッチが行われたサブタスクは実行を待つため FIFO の実行待ちキューにキューイングされる．逆に，スケジュールされた将来のサブタスクがデータのフェッチを必要としない場

```
subtask get_next_subtask(gpu) {
  glob_min = iomin_subtask(global_list);
  local_min = iomin_subtask(local_list[gpu]);
  if(glob_min < local_min) {
    remove(global_list, glob_min);
    return glob_min;
  } else {
    remove(local_list[gpu], local_min);
    return local_min;
}}

void schedule() {
  foreach(g in available_gpu) {
    if(mem_use[g] <= mem_th[g]) {
      if(!global_list.empty() || !local_list.empty()) {
        st = get_next_subtask(g);
        lock_mem(st);
        if(require_io(st)) {
          invoke_io(st);
        } else {enqueue_st(st)};
      }
} } }
```

図 5.8　Victream スケジューラの疑似コード

合，スケジューラは実行待ちキューにサブタスクをキューイングし，直ちに次のサブタスクのスケジューリングに移行する．この手法により GPU に将来実行するサブタスクのデータプリフェッチを継続させることができる．また，将来実行するサブタスクのスケジューリングは GPU へのデータロードを停止するメモリ使用率のしきい値まで達すると，その値を下回るまで停止する．

スケジューラが将来実行するサブタスクをスケジュールする場合，グローバルリストと，GPU 毎に用意されたローカルリストの内，スケジュールする GPU に対応するリストを探索する．グローバルリストに含まれるサブタスクは，DAG において依存する全て

の親のサブタスクの実行が完了し入力データパーティションが出力されているサブタスクである．従ってグローバルリストが保持するサブタスクはどの GPU でも実行可能である．一方，各 GPU のローカルリストが保持するサブタスクは，未完了の全ての親のサブタスクがその GPU で実行待ちであるサブタスクである．従って GPU ローカルリストに保持されるサブタスクは，その GPU でのみ実行可能である．なぜならその GPU でサブタスクをスケジュールすると，スケジュールしたサブタスクがタスクエクセキュータで実行されるまでには，スケジュール時に未完了だった親のサブタスクの完了が保証されているからである．

スケジューラはグローバルリストと GPU が該当するローカルリストが保持するサブタスクの内，将来実行するサブタスクを貪欲法で選択する．具体的には，スケジューリングに際して発生するデータスワップのデータ I/O 量が最小のサブタスクを選択する．そのため，サブタスクのスケジュール時に入力データパーティションが既に GPU メモリに存在するサブタスクほど優先して選択される．

スケジューラは将来実行するサブタスクを実行待ちキューにキューイングした後，GPU の該当するローカルリストの更新を行う．具体的には，スケジュールしたサブタスクの下流のサブタスクの内，5.4.1 節で定めた基準に該当するサブタスクが新たに生じた場合，そのサブタスクを含むよう該当する GPU ローカルリストを更新する．

以上の機能を提供するアルゴリズムの疑似コードを図 5.8 に示す．

実行待ちキューにキューイングされたサブタスクの実行はシリアライズされサブタスクエクセキュータで実行される．そしてサブタスクの実行が完了するとロックされていた入出力データパーティションのメモリ領域がアンロックされる．またサブタスクの実行完了時には，グローバルリストと GPU ローカルリストに含むべきサブタスクが変化するため，5.4.1 節で述べた基準に基づいてこれらのリストの更新を行う．

ここで I/O エクセキュータとサブタスクエクセキュータは非同期パイプラインを構成すると見ることができる．最初のステージは I/O に関するステージであり，I/O エクセキュータがスケジュールした将来のサブタスクの入力データパーティションのプリフェッチを継続する．一方第二のステージはサブタスクの実行ステージであり，サブタスクエクセキュータはスケジュールされたサブタスクの演算を継続する．またパイプラインに投入されるサブタスクのスケジューリングは GPU へのデータ I/O 量を最小化する順となるよう決定される．

また Victream のスケジューラでは GPU のメモリ使用率がスワップアウトのしきい値を超えた場合，バックグラウンドでロックされていないデータパーティションのスワップアウトを行う．スワップアウトは GPU のメモリ使用率がしきい値を下回るまで継続す

る．スワップアウトを行うデータパーティションの選択には LRU(Least Recently Used) のアルゴリズムを用いている．またスワップアウトのしきい値は前述したデータロードのしきい値より小さい値に設定する．今日の GPU は双方向の I/O の並列実行をサポートしているため，Victream ではスワップアウトとデータロードのデータ I/O を並列に行う．

5.4.3 提案手法の制約

提案手法は GPU のデータロードの I/O が GPU 間で均一になるようにスケジューリングを行う．しかし，現在の実装は結果としてサブタスクの計算が GPU 間で不均一となった場合を考慮していない．つまりデータ I/O を最小化するスケジューリングを行った結果，ある GPU では演算負荷が大きなタスクが多数実行を待っているが，別の GPU がアイドル状態となった場合の対処法を考慮していない．そのような課題が発生する場合，実行待ちキューで長時間実行を待っているサブタスクを別の空いている GPU に移動する手法を追加で導入することで解決する方法が考えられる．

第 6 章

分離した複数の計算アクセラレータを用いた計算の最適化方式の評価

本章では第 5 章で提案した複数の GPU (Graphics Processing Unit) を用いた Out-of-Core 処理向けミドルウェア Victream に対して行った性能評価について述べる．行った評価は次の 3 つである．

1 つ目の評価は，複数の GPU を I/O (Input/Output) スロットに保持するスタンドアロンのサーバを用いた．このサーバで試作した Victream を動作させ，5.2.3 節で述べたパンシャープンやマイクロベンチマークの性能評価を行った．これらの評価により，Victream が Out-of-Core 処理においてボトルネックとなるデータ I/O 量を最小化することで GPU の演算リソースを効率的に使用し，従来手法のスケジューリングより優れた性能を保持することを示す．また計算に用いる GPU 数に比例して性能向上が実現できることを示す．

2 つ目の評価では，Victream スケジューラの性能をシミュレーションにより評価した．Victream のスケジューラは 5.4 節で述べたように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．そのため Victream が選択するスケジュールが最良のスケジュールとは限らない．そこで本評価では取り得る全てのスケジュールを Brute Force でシミュレーションにより全探索した．そして全探索で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 のスケジュールの性能を評価した．シミュレーションでは 1 つ目の評価で用いたマイクロベンチマークからいくつかを選択し，その計算時間のシミュレーションを行った．

3 つ目の評価では，Victream を第 3 章で提案した ExpEther を用いて GPU をホストから分離したシステムに適用した．Victream は GPU をホストから分離したプラット

フォームで性能ボトルネックとなるイーサネットを通過するデータ I/O 量を最小化する．それにより，Victream は従来手法と比較して GPU をホストから分離した場合でも GPU の演算リソースを効率的に活用し，計算性能を GPU 数に対して効率的に向上する特性を実現できることを示す．

6.1 スタンドアロンサーバでの評価

本節では GPU を I/O スロットに保持するスタンドアロンのサーバを用いて Victream の性能を評価した結果を述べる．初めに 5.2.3 節で述べた Out-of-Core 処理の応用例であるパンシャープンの性能を評価した．次に Victream の評価用に作成したマイクロベンチマークを用いた評価を行った．

評価では 4 つの NVIDIA Tesla K20 GPU をホストの I/O スロットに挿入した．これは用いたホストに挿入できる最大の GPU 数である．それぞれの GPU は 5 GB のメモリを保持し，単精度浮動小数点の演算性能は 3.52Tflops だった．ホストは E5-2609 Xeon CPU を 2 つ保持し，Operating System (OS) には Ubuntu 14.04 を用いた．全ての入力及び出力データは RAMdisk のファイルに格納した．Victream の GPU のメモリへのデータロードのしきい値は 70% とし，スワップアウトを開始するしきい値は 50% に設定した．Victream のユーザライブラリとランタイムを含めたソースコードは c++ と CUDA7.5 で実装した．現在の実装のライン数は 10 K である．

また，Victream の性能を従来手法のスケジューラと比較するため，Victream とは別に First-In First-Out(FIFO) スケジューラを作成した．また PTask [52] の入力データの場所を考慮するスケジューラも試作した．FIFO スケジューラではサブタスクは DAG 内で依存する上流の親のサブタスクの実行が完了した段階で実行可能になる．FIFO スケジューラはサブタスクが実行可能になった順に実行待ちキューにキューイングし，それらを順に実行する．そのため FIFO スケジューラは実行待ちキューの先頭のサブタスクに対し演算リソースが空いた GPU を貪欲法により割り当てる．これにより，FIFO スケジューラは GPU の演算リソースを常に稼働させるするようする．また FIFO スケジューラはデータスワップの I/O 量は無視している．一方 PTask のスケジューラは FIFO スケジューラと同様に実行待ちキューの先頭のサブタスクから順に実行するが，スケジュールするサブタスクの入力データの場所も考慮する．もしスケジュールするサブタスクの過半の入力データを保持する GPU が演算リソースが空いた GPU と異なる場合，PTask のスケジューラは入力データを保持する GPU が少し待てば利用可能となることを期待し，そのサブタスクの GPU への割り当てを一定時間待つ．しかし，Out-of-Core 処理では実行

待ち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る場合があり，その場合は PTask スケジューラは FIFO スケジューラと同じ動作となる．

6.1.1 パンシャープン

本評価では Victream のユーザライブラリの API (Application Programming Interface) を用いて実装したパンシャープンの性能を評価した．作成したユーザプログラムの DAG (Directed Acyclic Graph) は 25 個のユーザ定義関数から構成した．

図 6.1 に実装したパンシャープンを用いて 2.5GB のパンクロマティック画像と 163MB のマルチスペクトル画像から 7.6GB のカラー画像を生成した場合の性能を示す．評価では Central Processing Unit (CPU) と GPU の性能を比較した．CPU のプログラムは Python と NumPy を用いて実装し，全ての Xeon の CPU コアがパンシャープン処理で利用されるようにした．図の縦軸のスループットは，計算時間の逆数である．従って単位時間あたりにパンシャープンの計算を実行できる回数を示す．GPU を用いた処理では計算が Out-of-Core となることも確認した．

まず CPU と GPU の結果を比較すると，Victream を用いて GPU にパンシャープンの処理をオフロードした場合，計算が Out-of-Core でも 14 倍の性能向上が得られることがわかった．また，GPU 数に応じて Victream の処理性能も向上した．また Victream で 4 つの GPU を用いた場合，1 つの GPU を用いた場合と比較して性能が 1.8 倍に向上した．ここで Victream の性能が GPU 数の増加に対して飽和する要因は入力及び出力データの格納に使用している RAMdisk の I/O 性能である．入出力データの I/O 量は一定のため，計算に用いる GPU 数が増大し GPU による処理時間が減少すると計算の全体時間に占める入出力データ I/O の時間の割合が増加する．その結果，用いる GPU 数に対し計算性能が飽和する．

また Victream を GPU の従来手法のスケジューラと比較した場合，1 つの GPU を用いた計算では Victream は PTask スケジューラより 32%，FIFO スケジューラより 31% 高い性能を示した．よって Victream は Out-of-Core 処理において GPU の演算リソースを有効に活用し，従来手法より優れた性能を実現できる．また Victream のスケジューラは GPU 数を増加させた場合も，従来手法のスケジューラより常に優れた性能を実現できる．

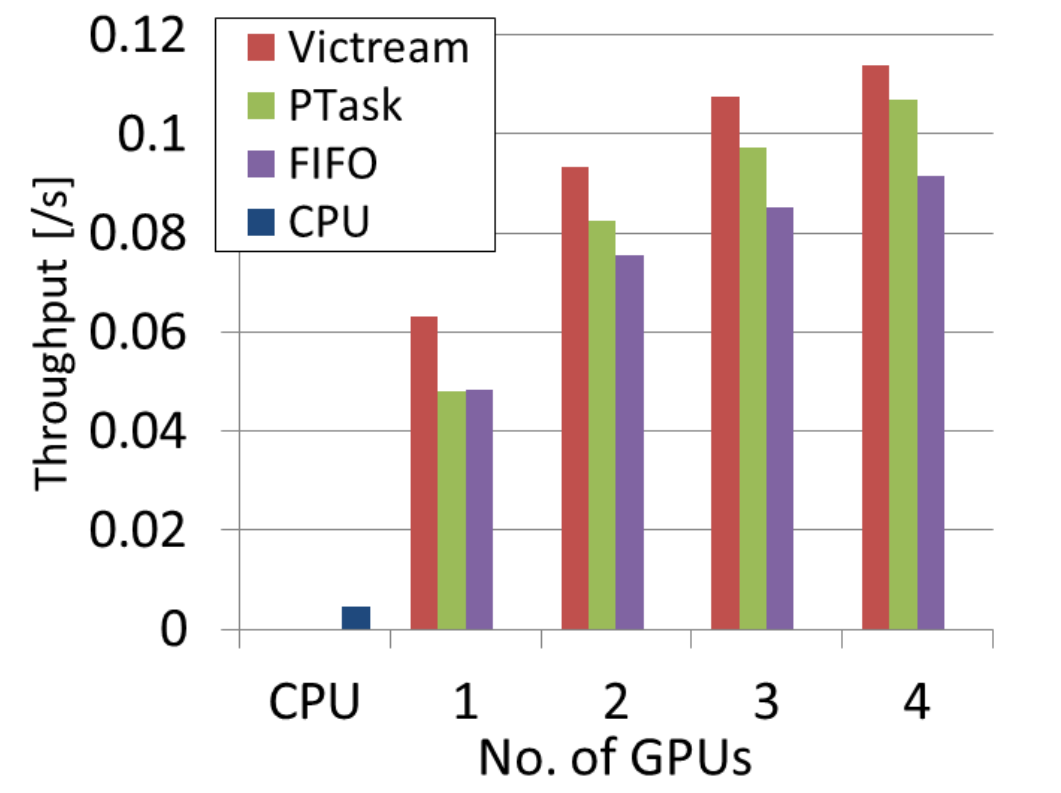


図 6.1　パンシャープンの計算スループット

6.1.2　マイクロベンチマーク

次に Victream の評価用に作成したマイクロベンチマークを用いて Victream の性能を評価した．作成したマイクロベンチマークはロジスティック回帰，ソート，複数回の画像フィルタ (ブラーフィルタ)，行列積の 4 つである．それぞれのマイクロベンチマークが扱うデータ形式は行列か画像である．これらのベンチマークは Victream の API を用いて実装した．

以下では 2 種類の評価を行った．1 つ目の評価では，計算に用いる GPU 数を変化させ，用いる GPU 数に応じた計算性能の向上を評価した．この評価では常に Out-of-Core の領域で評価を行うため，それぞれの GPU が処理するデータ量が一定となるよう調整した．つまり NGPU を用いた評価では 1 つの GPU を用いた場合の評価と比較して N 倍のデータ処理量で評価を行った．2 つ目の評価では入力データ量を変化させて計算時間を測定した．この評価では 3 つの GPU を使用した．2 つの評価におけるデータパーティションのサイズはおよそ 256MB とした．マイクロベンチマークの間でデータパーティ

ションのサイズにバラつきがあるのは，用いたデータのアラインメントのためである．

図 6.2 及び 6.3 に GPU 数を変化させた場合の計算性能を示す．評価ではすべての測定点で計算が Out-of-Core となることを確認した．図の縦軸は処理性能を示しており，処理データ量と，1 秒あたりに実行できる計算回数との積である．つまり 1 秒当たりに処理されるデータ量である．得られた結果から Victream では用いる GPU 数に対して計算性能が効率良く向上できることがわかる．また，ソートとブラーフィルタでは GPU 数に対して計算性能が飽和した．これはパンシャープンの場合と同様，入出力データの格納に用いた RAMdisk の I/O 帯域が原因である．

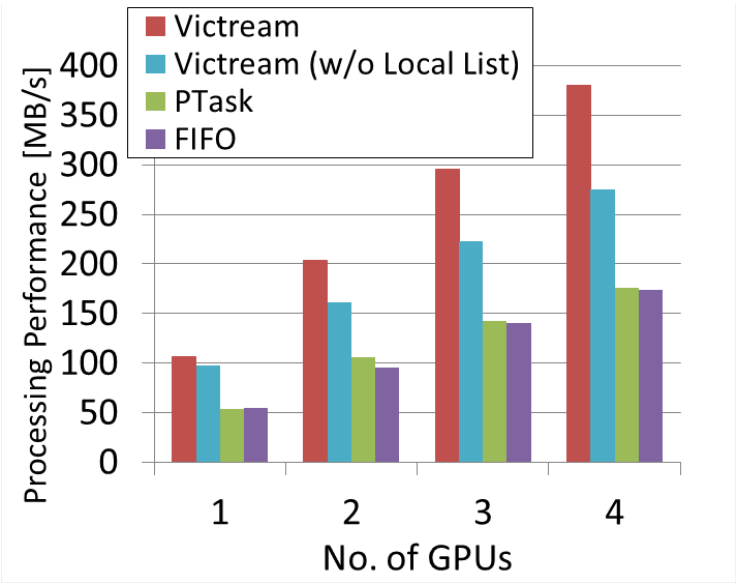
従来手法のスケジューラとの比較では Victream スケジューラが FIFO スケジューラや PTask スケジューラより 4 つ全てのマイクロベンチマークで優れていることが示された．この結果から Victream は従来手法と比較して Out-of-Core 処理において GPU の演算リソースを有効に活用し，複数の GPU を用いた場合も大きな計算性能の向上が得られることがわかった．

図 6.2(a) に示すように従来手法と Victream の性能の差異はロジスティック回帰で顕著であり 92%-117%PTask より性能が向上した．ロジスティック回帰の評価では GPU のメモリ容量より大きなデータを分割し繰返し処理を実行する．繰り返し処理の各イテレーションの開始の際，Victream スケジューラは前回のイテレーションで GPU メモリ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．また DAG においてそのサブタスクの下流に位置するサブタスクを続けて実行する．それにより GPU のメモリに存在するデータを可能な限り再利用し Out-of-Core 処理で性能ボトルネックとなるデータスワップに関するデータ I/O 量を最小化する．これに対し FIFO スケジューラは，スケジュール可能なサブタスクの入力データパーティションの場所を考慮せず，全てのイテレーションでサブタスクを FIFO の順で実行するためデータスワップ量が増大する．一方 PTask スケジューラは実行待ちキューの先頭のサブタスクの入力データパーティションの位置を考慮する．しかしこの手法の効果があるのは，そのパーティションがスワップアウトされず，いずれかの GPU に存在する場合だけである．逆に実行待ちキューの先頭のサブタスクの入力データパーティションがスワップアウトされていた場合，そのサブタスクは演算リソースが空いた GPU に割り当てられる．従って Out-of-Core 処理では PTask のスケジューラもキューの先頭のサブタスクをスケジュールすることが，キューの中ほどで実行を待っているサブタスクの入力データを GPU からスワップアウトすることになり，データスワップに関する GPU のデータ I/O 量が増大する．

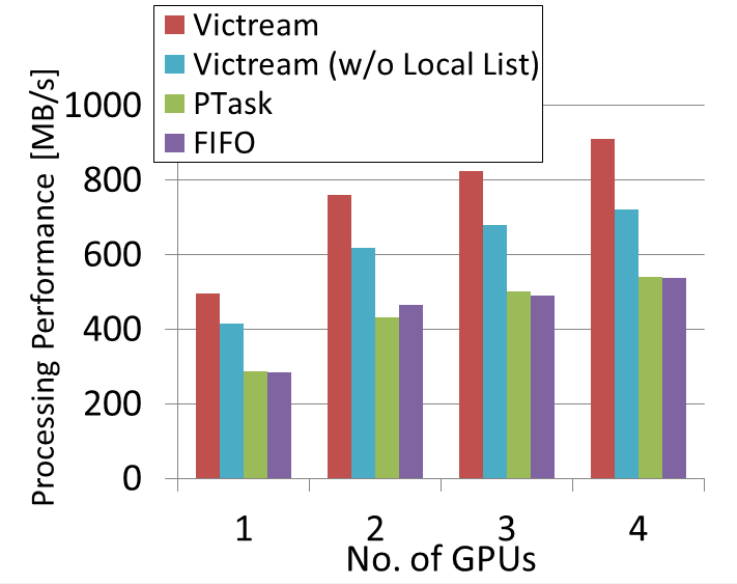
また行列積ではスケジューラ間の性能差が殆どない．これは Victream は行列積の計算

時のデータ I/O 量を削減しているが，行列積は演算負荷が高い処理のため計算全体の性能ボトルネックが GPU の演算リソースであり，4 つのスケジューラの差が微小になったためである．

次に，提案手法である 5.4 節で述べた GPU のプリフェッチとデータ I/O 量の最小化を同時に実現するローカリティアウェアスケジューラの効果を評価した．評価では本研究で提案したスケジュール可能なサブタスクの再定義の有無でローカリティアウェアスケジューラの性能を比較した．以下ではスケジュール可能なサブタスクの再定義を行わないローカリティアウェアスケジューラを拡張前のスケジューラ，再定義を含めたスケジューラを拡張後のスケジューラと呼ぶ．図 6.2 及び 6.3 には Victream スケジューラが図 5.7 で示した GPU ローカルリストのタスクをスケジュール候補として探索しない場合の結果を示している．これは 5.4 節で述べたスケジュール可能なサブタスクの再定義を行わない拡張前のスケジューラの性能に該当する．また凡例が Victream の性能は拡張後のスケジューラの性能である．拡張前のローカリティアウェアスケジューラでは，スケジューラはグローバルリストのみ探索し，必要とするデータ I/O 量が最少のサブタスクをスケジュールする．得られた評価結果は，Victream のローカリティアウェアスケジューリングの拡張により，拡張前のローカリティアウェアスケジューリングより計算性能が向上することを示している．ロジスティック回帰の場合，提案手法による性能向上は 9-38% である．

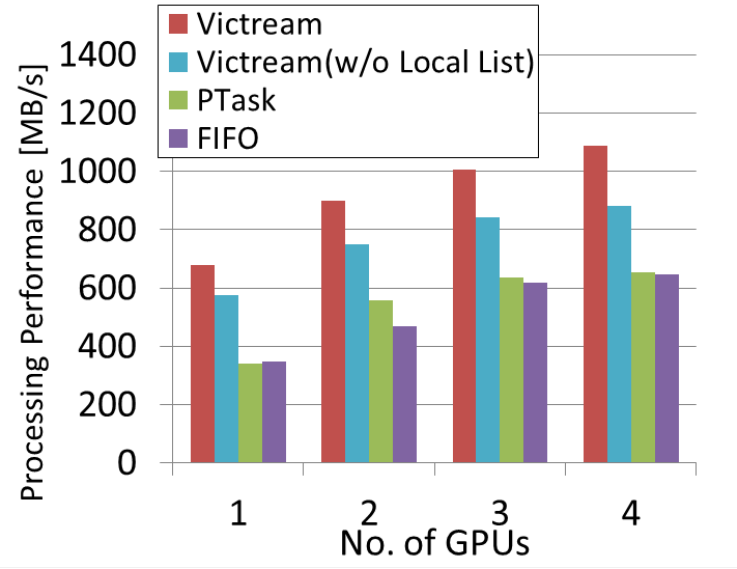


(a) ロジスティック回帰

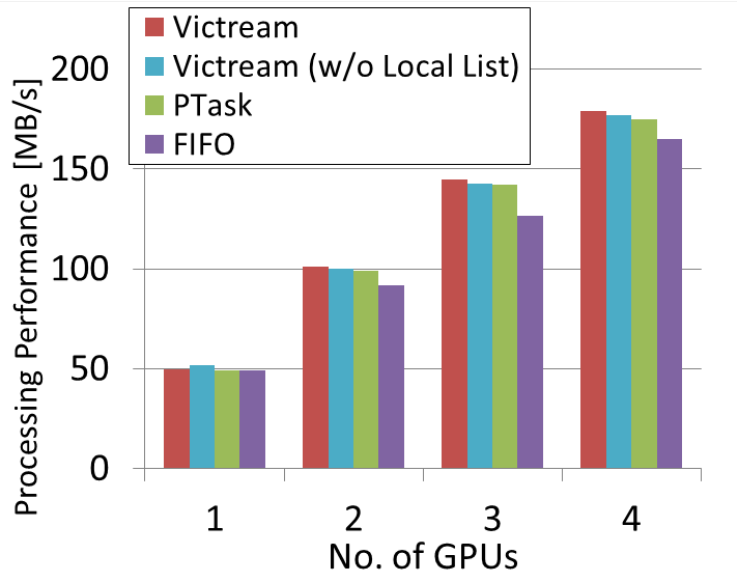


(b) ソート

図 6.2　GPU 数を変化させた場合の計算性能 (ロジスティック回帰及びソート)



(a) ブラーフィルタ



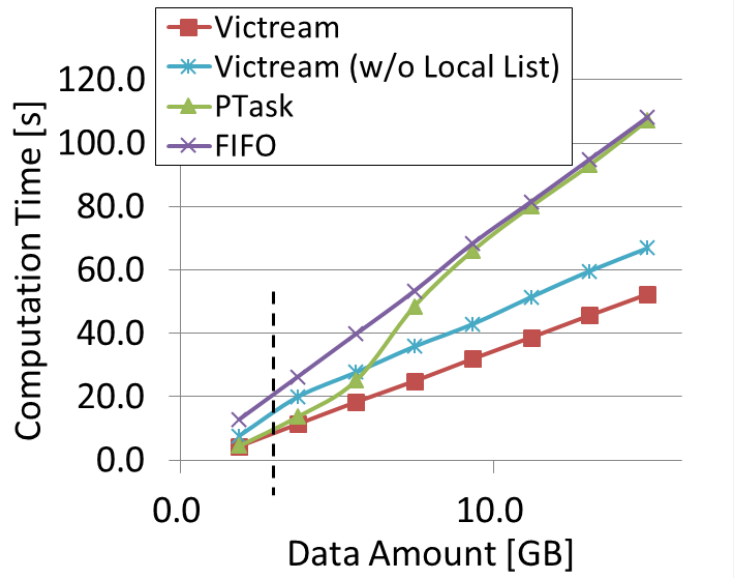
(b) 行列積

図 6.3　GPU 数を変化させた場合の計算性能 (ブラーフィルタ及び行列積)

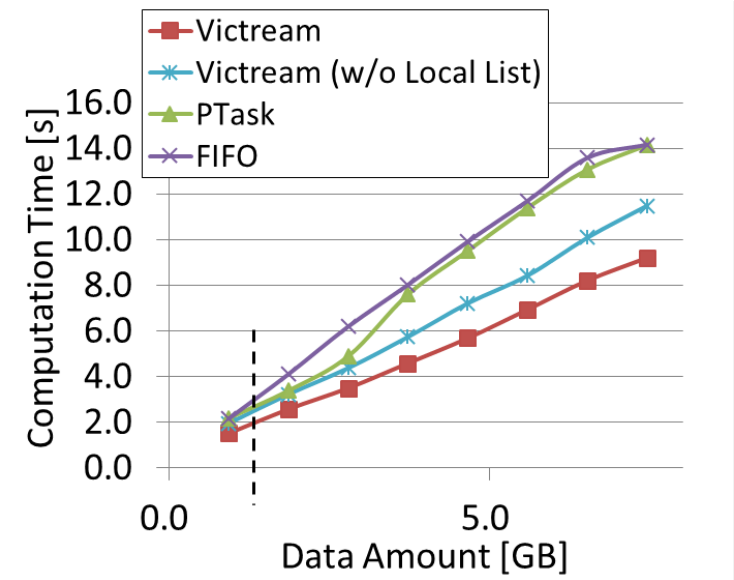
次に，2 つ目の評価である入力データ量を変化させた場合の評価結果を示す．図 6.4 及び図 6.5 は各ベンチマークの計算時間である．それぞれのグラフの破線は Out-of-Core 処理が始まるデータ量を示す．これらの評価結果は，計算が GPU の Out-of-Core 処理で

はない場合，Victream のスケジューラと従来手法のスケジューラの性能は等しいことを示している．一方，入力データが増加し Out-of-Core 処理が始まると，Victream が優れた性能を実現する．また評価結果は提案手法である拡張したローカリティアウェアスケジューリングが拡張前のローカリティアウェアスケジューリングに対して優れていることも示している．PTask スケジューラに関しては Out-of-Core ではない領域で FIFO スケジューラより性能が良い．これは PTask が入力データの位置を考慮したスケジュールを行い GPU のデータ I/O 量を抑制するためでる．また行列積の計算時間が入力データ量に対して非線形に増加するのは，行列積の計算コストが入力データ量の二乗に比例するからである．

図 6.6 及び図 6.7 には 2 つ目の評価における入力データ量に対する GPU のデータ I/O の総量を示す．データ I/O の総量は試作したミドルウェアのログを解析して算出した．このときデータ I/O 量はデータスワップに関するデータ I/O のみ考慮している．評価結果は Victream が従来手法と比較してデータスワップに関する GPU のデータ I/O 量を削減することを示している．また拡張したローカリティアウェアスケジューリングの方が拡張前のローカリティアウェアスケジューリングより削減量が多い．一方行列積のベンチマークでは図 6.5 の結果では行列積の性能ボトルネックが GPU の演算リソースだったため Victream と従来手法で性能が同じだったが，図 6.7 から Victream は I/O 量を削減することが確認できる．

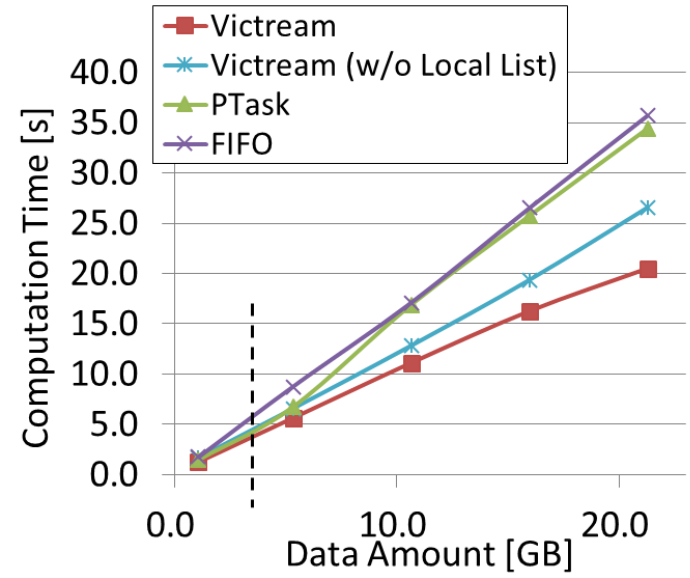


(a) ロジスティック回帰

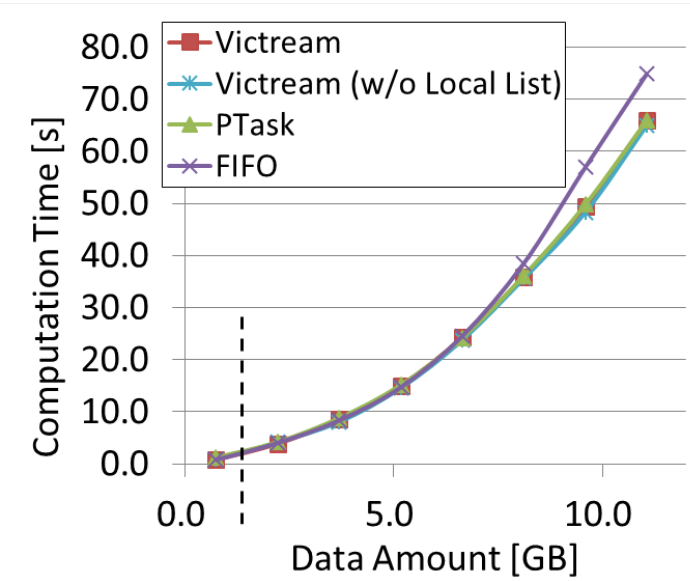


(b) ソート

図 6.4　入力データ量の変化に対する計算時間 (ロジスティック回帰及びソート)

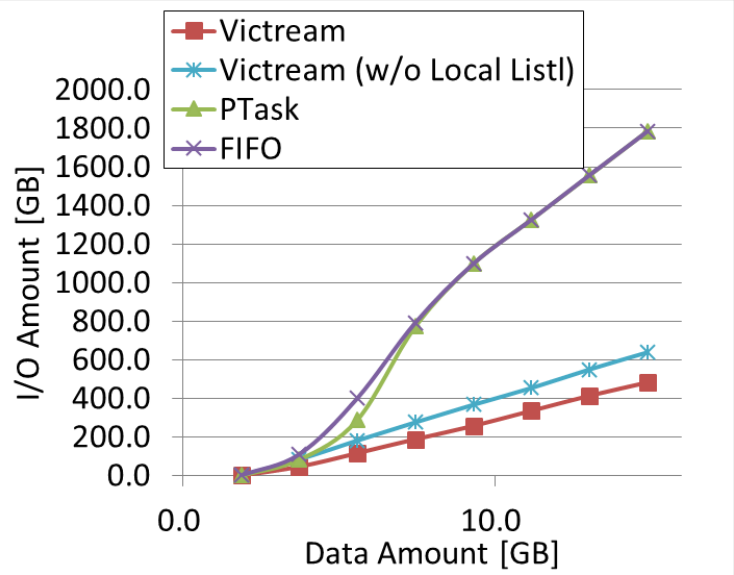


(c) ブラーフィルタ

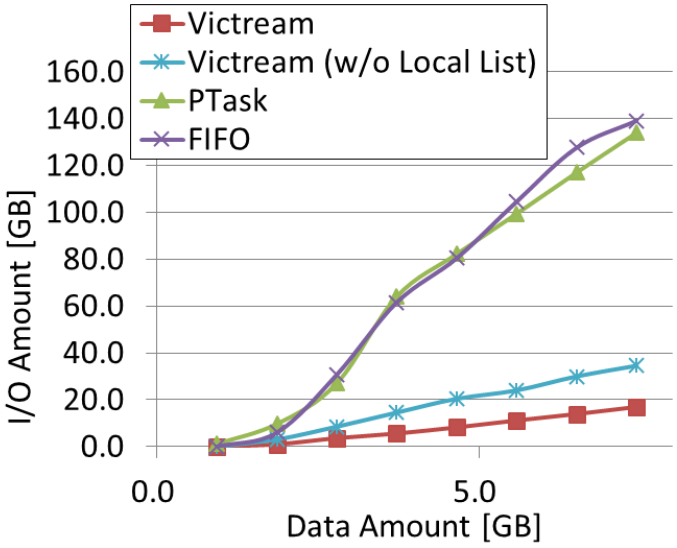


(d) 行列積

図 6.5　入力データ量の変化に対する計算時間 (ブラーフィルタ及び行列積)

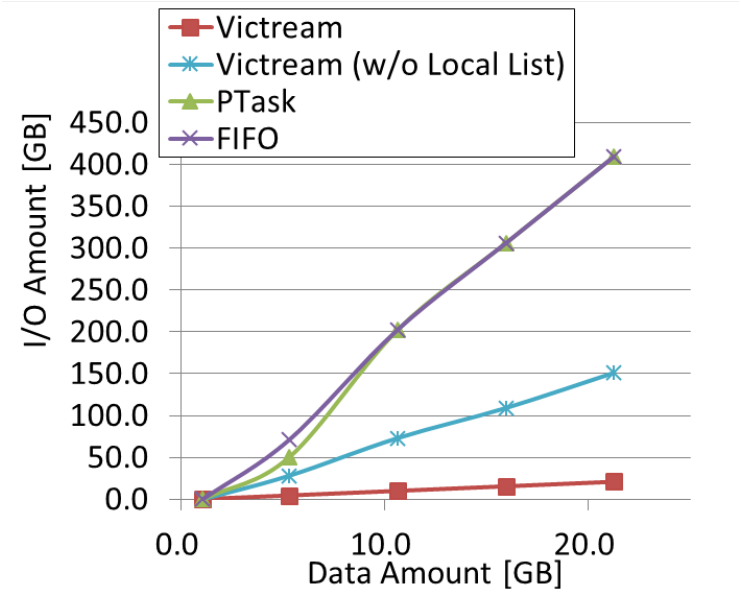


(a) ロジスティック回帰

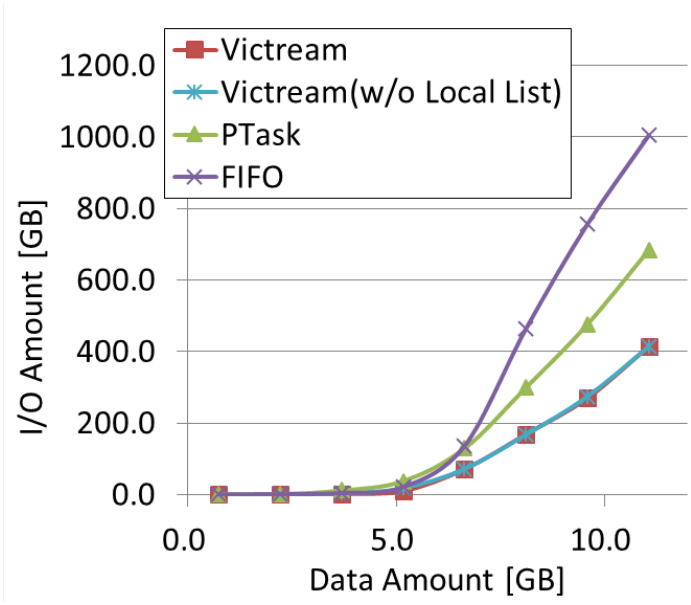


(b) ソート

図 6.6　入力データ量の変化に対する GPU のデータ I/O 量 (ロジスティック回帰及びソート)



(c) ブラーフィルタ



(d) 行列積

図 6.7　入力データ量の変化に対する GPU のデータ I/O 量 (ブラーフィルタ及び行列積)

以上本節では GPU を保持するスタンドアロンサーバで Victream の性能を評価した．評価では計算に用いる GPU 数に対する計算性能の向上と，入力データ量を変化させたと

きの計算時間の 2 つの評価を行った．得られた評価結果は，Victream が複数の GPU を用いた Out-of-Core 処理において性能ボトルネックとなるデータスワップのデータ I/O 量を抑制し，GPU の演算リソースを効率的に利用して高い計算性能が実現できることを示している．また，用いる GPU 数に応じて計算性能を効率的に向上できることも示した．従来手法のスケジューラとの比較では，Victream は従来手法より優れた計算性能を実現し，GPU 数を増加させた場合も大きな計算性能の伸びが実現できる．また，本節の評価では提案手法である GPU のデータプリフェッチとデータ I/O 量の最小化を同時に実現するローカリティアウェアスケジューリングの拡張が，拡張前のローカリティアウェアスケジューリングより GPU のデータ I/O 量の削減量が大きく，高い計算性能が実現できることを示した．

6.2 シミュレーションによる Victream スケジューラの性能評価

本節では Victream スケジューラの性能を計算シミュレーションにより評価する．Victream はデータスワップに関する GPU のデータ I/O 量を最小化するため，5.4 節で述べた動的でヒューリスティックなスケジューリングを採用している．これは Victream が実行するスケジュールが必ずしも最短の計算時間を実現しないことを示す．そこで本節では計算シミュレータを作成し，最良のスケジュールの計算時間と Victream の計算時間とを比較する．比較する計算時間は最良のスケジュールも Victream のスケジュールもシミュレーション上の計算時間である．最良のスケジュールのシミュレーションではユーザプログラムの DAG が与えられた場合に取りうるスケジュールを Brute Force で全探索し，計算時間が最短の結果を取得する．また Victream の計算時間のシミュレーションでは Victream のスケジューラの動作を疑似するシミュレータを作成した．シミュレータは 6.1 節で実施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，行列積の 3 つについて作成した．

表 6.1 に作成したシミュレータで用いたパラメータを示す．これらの値は実験で用いた評価プラットフォームの実測値である．また図 6.8 に 6.1 節で用いた評価プラットフォームにおける計算の実測値と，作成した Victream のシミュレータによる計算時間の比較を示す．計算の実測値とシミュレーション結果のかい離は最大 9% であり，有効なシミュレーションが実現できていることを確認した．

表 6.1　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

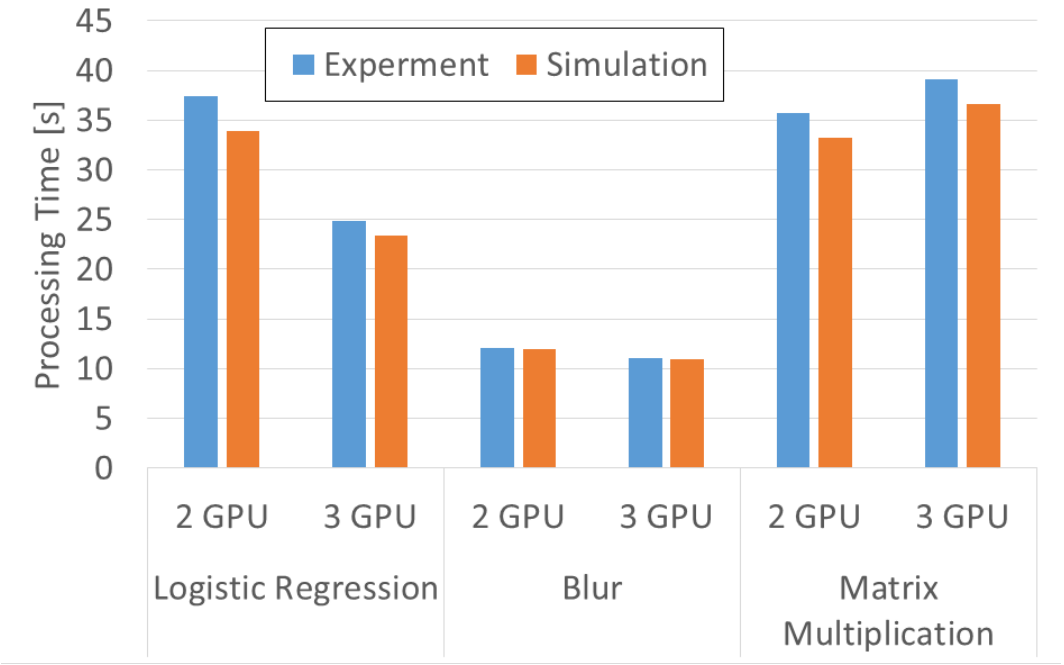


図 6.8　Victream の実験とシミュレーションによる計算時間の比較

次に Victream と最良のスケジュールの計算時間のシミュレーション結果を述べる．表 6.2 にシミュレーションに用いた各マイクロベンチマークの条件を示す．シミュレーションが小規模な DAG を対象としている理由は，DAG の頂点数の増大と用いる GPU 数の増加でシミュレータが全探索するスケジュール数が組み合わせ爆発により増大し，大規模な DAG のシミュレーションが困難だからである．表 6.2 に示した条件でも各マイクロベンチマークにつき半日程度の計算時間が必要だった．DAG の頂点数や計算に用いる GPU 数を 1 つ増加させると計算時間が大きく増加する．

図 6.9 にシミュレーションの評価結果を示す．図には Victream，最良のスケジュール，最悪のスケジュールの 3 つの計算時間を示した．Victream の結果はロジスティック回帰と入力データが 4 パーティションのブラーフィルタで最良のスケジュールと同じ性能となった．一方，入力データが 5 パーティションのブラーフィルタと行列積ではそれぞれ 2% と 10% 計算時間が増大した．

表 6.2　シミュレーションにおける各マイクロベンチマークの条件

	Logistic Regression	Blur (4 Partitions)	Blur (5 Partitions)	Matrix Multiplication
No. of GPU	1	2	2	2
GPU Memory Capacity	1650 MB			
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

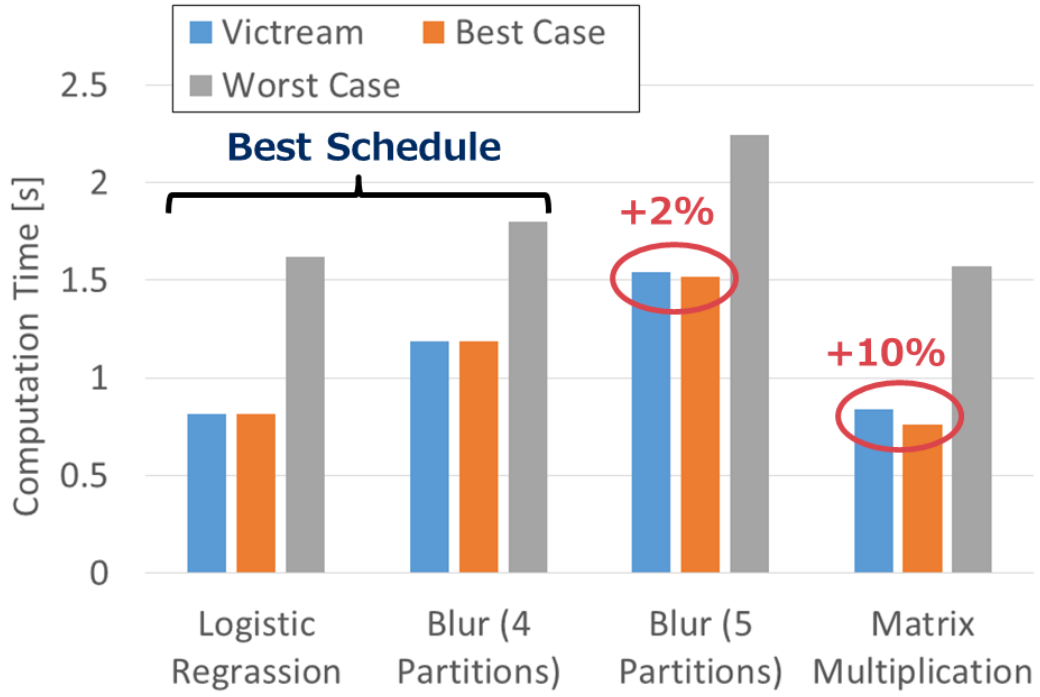


図 6.9　Victream の実験とシミュレーションによる計算時間の比較

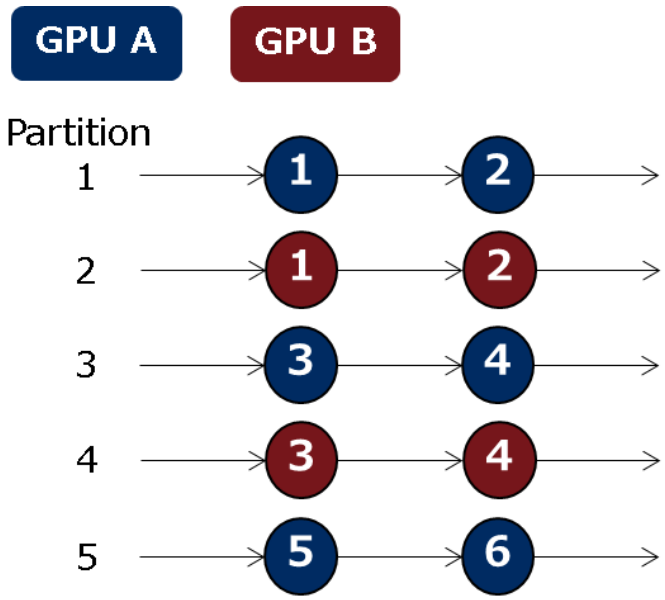
そこで性能が最良ではなかった 2 つのベンチマークで原因を解析するため，Victream と最良のベンチマークのサブタスクの実行順を比較した．比較結果を図 6.10 及び図 6.12 に示す．

ブラーフィルタでは，Victream の場合，GPU 間でサブタスクの割り当てが 6 つと 4 つであり負荷が不均一となっている．一方最良のスケジュールでは，負荷がより均一となる．最良のスケジュールの場合もサブタスクの実行数はそれぞれの GPU で 6 つと 4 つ

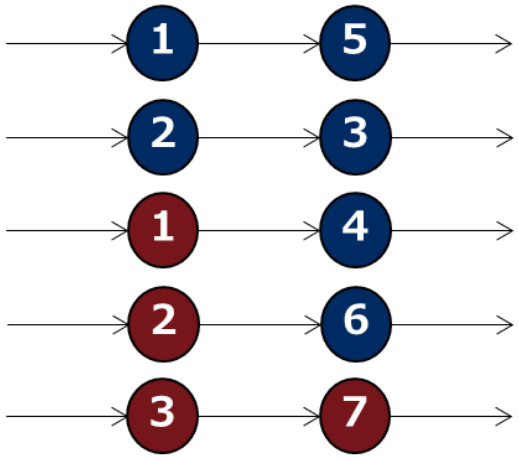
で Victream と同じだが，ブラーフィルタの処理では DAG の左側のサブタスクの方が右側のサブタスクより入力データを RAMDisk のファイルから読み込むためコストが高い．従ってサブタスクの実行数が少ない GPU B には入力データを用意するコストが高い左側のサブタスクを多く割り当てられることで GPU 間の負荷が均一となっている．

一方行列積は，図 6.12 を参照しても 2 つの GPU へのサブタスクの割り当てが類似しており原因がわかりにくい．そこでミドルウェアの動作を解析するためログを確認したところ，Victream の方が最良のスケジュールより計算の後半に GPU のリソースが空いたタイミングで残存する未実行のサブタスクの入力データパーティションが他の GPU からスワップアウト中のため直ちにサブタスクをスケジュールできていないことがわかった．このため GPU リソースのアイドル時間が多く発生していた．これを確認するため図 6.11 でサブタスクの完了数に対する経過時間を比較した．2 つの手法で差が確認できるのは各 GPU が 3 個目以降のサブタスクを処理するときである．最良のスケジュールでは 5 つのサブタスクを処理する GPU が Victream と比較してアイドル時間が短く，データのロードや演算の実行が効率的に行われている．

以上の結果から Victream では常に最良のスケジュールを実現しないことがわかったが，今回の評価で用いたベンチマークでは最良のスケジュールから 10% 以内に計算時間の増加を抑えられた．Victream が応用対象としているのはアプリケーションからユーザ定義関数が与えられて DAG を作成し，その DAG の最適化を自動で行う分野である．そのような応用ではユーザ定義関数の実行時間をミドルウェアが知ることができないため，DAG を実行する前に最適なスケジュールを計算することができない．またもしユーザ定義関数の実行時間を予め知ることができ，最適なスケジュールを計算できたとしても，今回シミュレーションを行ったような頂点数が 10 点程度の非常に小規模な DAG の最適スケジュールの計算に半日以上を要する．それに対し，Victream では例えば 6.1 節で述べた評価においても 2-3 桁規模が大きい DAG をオンラインでスケジュールし，スケジューリング時間が GPU の計算時間のボトルネックとならない高速なスケジュール手法で GPU の Out-of-Core 処理のデータ I/O 量最小化を実現できる．

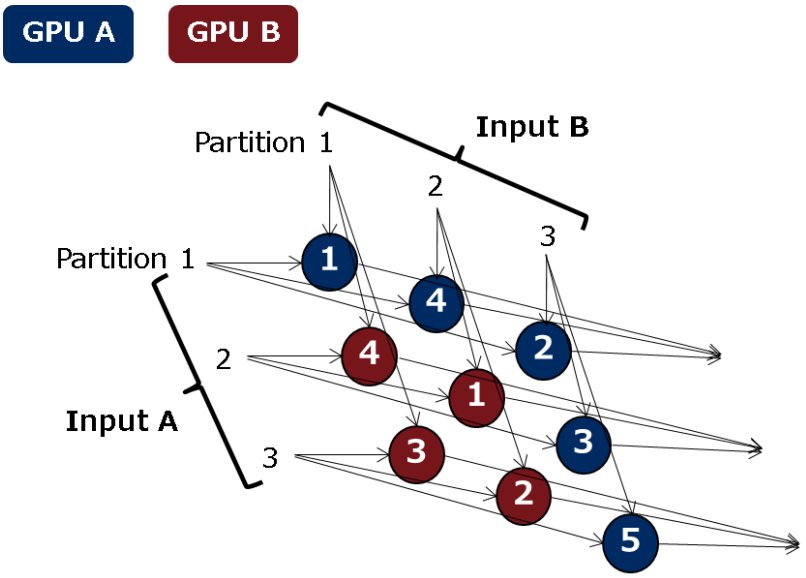


(a) Victream

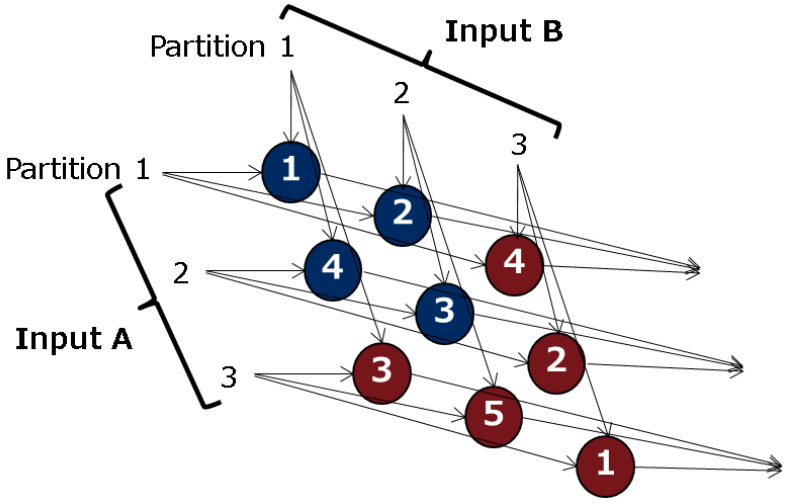


(b) 最良のスケジュール

図 6.10　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．

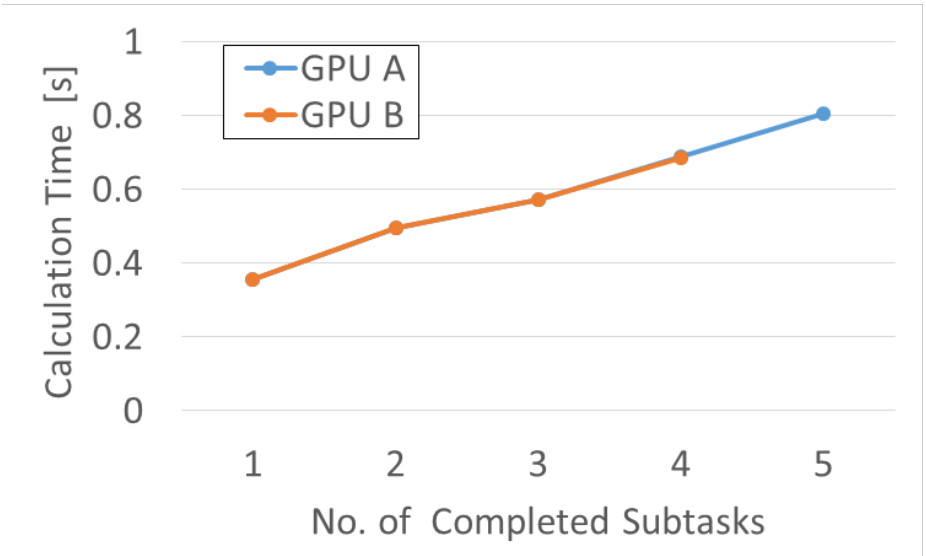


(a) Victream

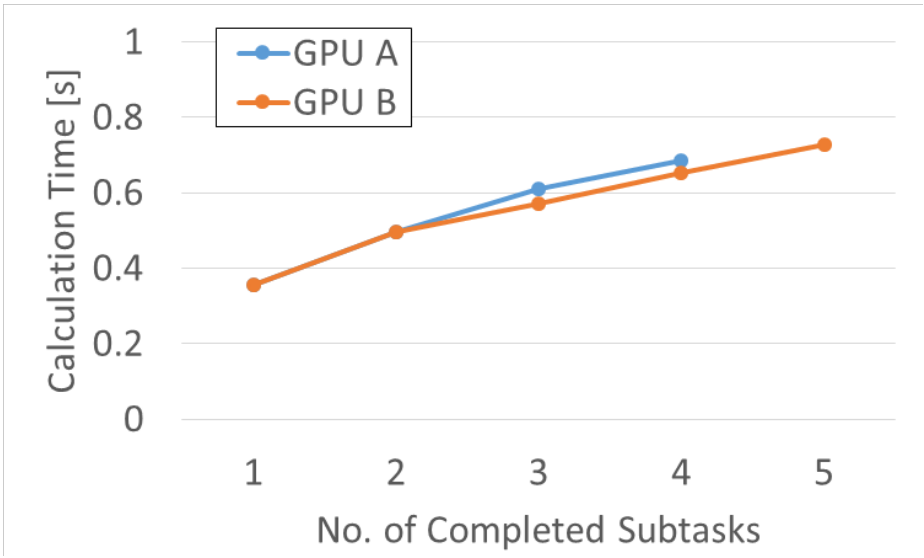


(b) 最良のスケジュール

図 6.11　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．



(a) Victream



(b) 最良のスケジュール

図 6.12　行列積の計算時間シミュレーションにおけるサブタスク完了数に対する経過時間

6.3　ホストから分離した GPU を用いた性能評価

3 つ目の評価では，第 3 章で提案した ExpEther を用いて GPU をホストから分離したプラットフォームにおける Victream の性能を評価した．用いた実験系を図 6.13 に示す．用いた ExpEther はインタフェースを 40GbE としたものである．ホストと 4 個の Nvidia K20 GPU を 2 系統のイーサネットを用いて接続した．これにより 3.3 節で述べた ExpEther のリンクアグリゲーション機能を用いて，ホストと GPU が 80 Gbps のイーサネットで接続された．

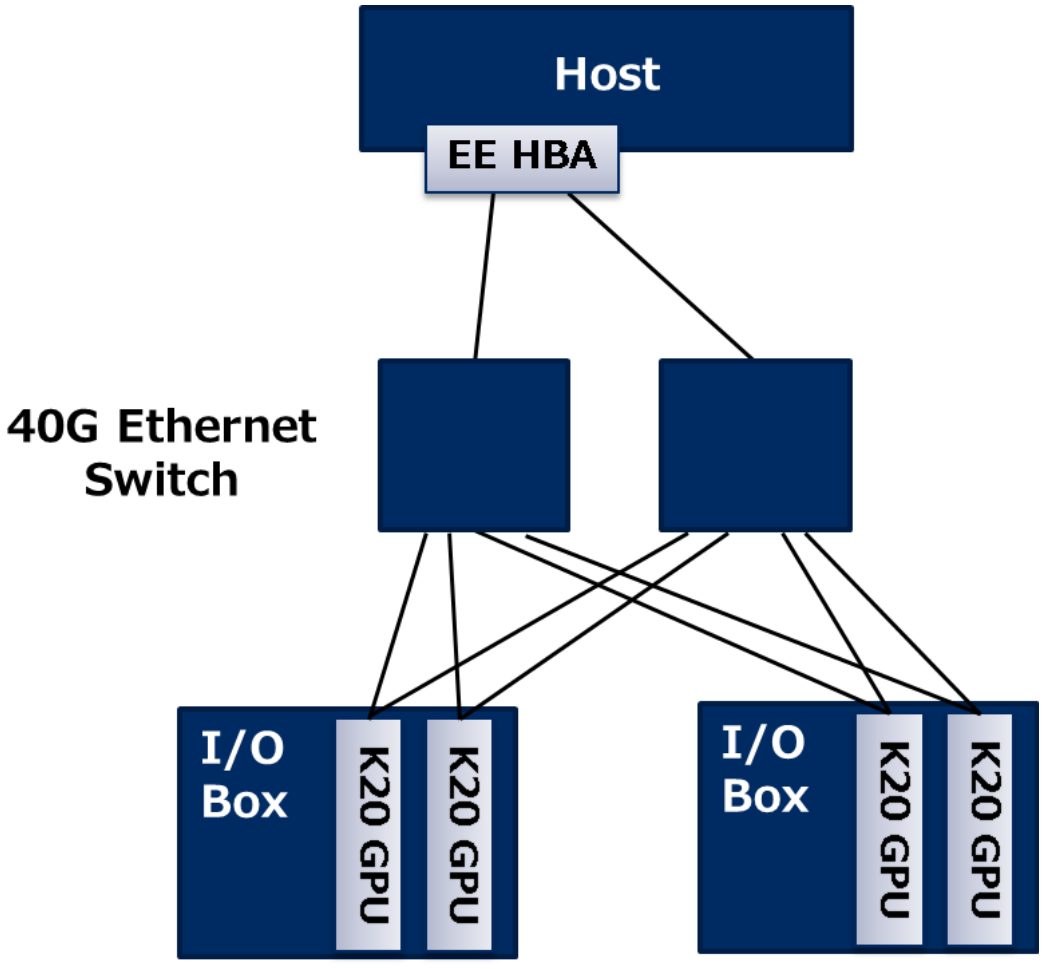
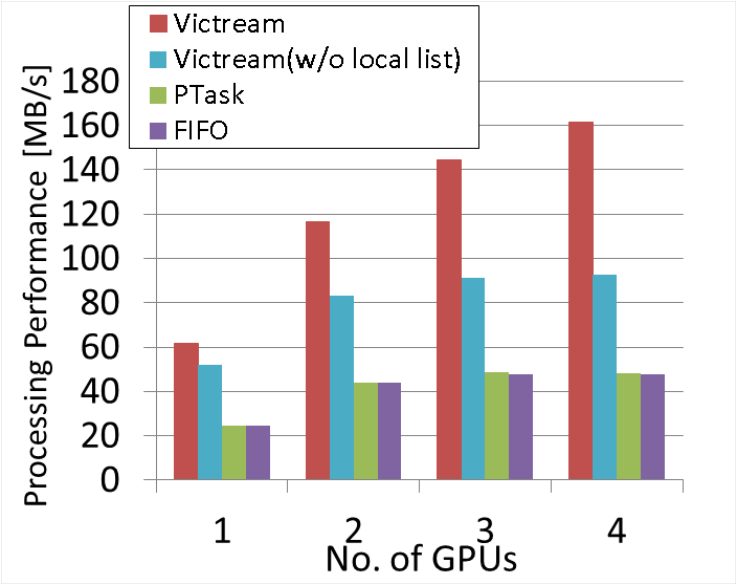


図 6.13　ExpEther によりホストから分離した GPU の性能評価の実験系

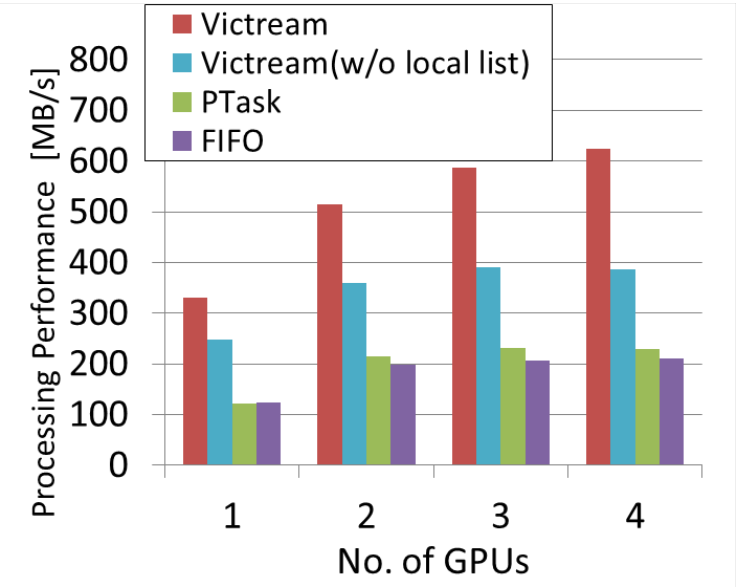
性能評価には 6.1 節で用いた 4 つのマイクロベンチマークを用いた．図 6.14 及び図 6.15 に GPU 数に対する計算性能の評価結果を示す．Victream は GPU をイーサネット

を用いてホストから分離した場合でも，計算の性能ボトルネックとなる GPU のデータ I/O 量を抑制し，従来方式より優れた計算性能が実現できることがわかる．また，GPU 数を増加させた場合の計算性能の向上も従来方式より大きい．

次に表 6.3 に今回評価した性能と，6.1 節で評価した GPU をホストの I/O スロットに挿入した場合の性能を比較した．表では GPU をホストの I/O スロットに挿入した場合の性能を 1 とした Victream の相対性能を示している．結果は，GPU をホストから分離した場合の性能はマイクロベンチマークの処理に依存するが，ホストの I/O スロットに GPU を挿入した場合と比較して概ね 50%-100% の性能が実現できることを示している．このことから，ExpEther の実現コストによるが，GPU の稼働率が概ね 50% 以下のシステムでは，個別のホストの I/O スロットに GPU を挿入するより ExpEther を用いてリソースプールに GPU を集約し，GPU アプリケーションを用いる場合だけ必要に応じてホストに GPU を割り当てる方が計算リソースを有効に活用できる可能性がある．そのとき，I/O デバイスの分離アーキテクチャのオーバヘッドによる性能低下を補うため，スタンドアロンサーバより最大で 2 倍の GPU 数を割り当てる必要がある．

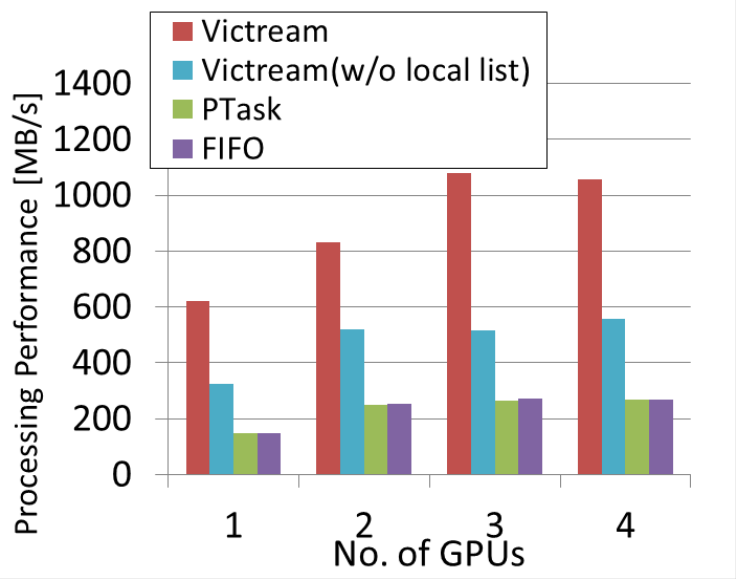


(a) ロジスティック回帰

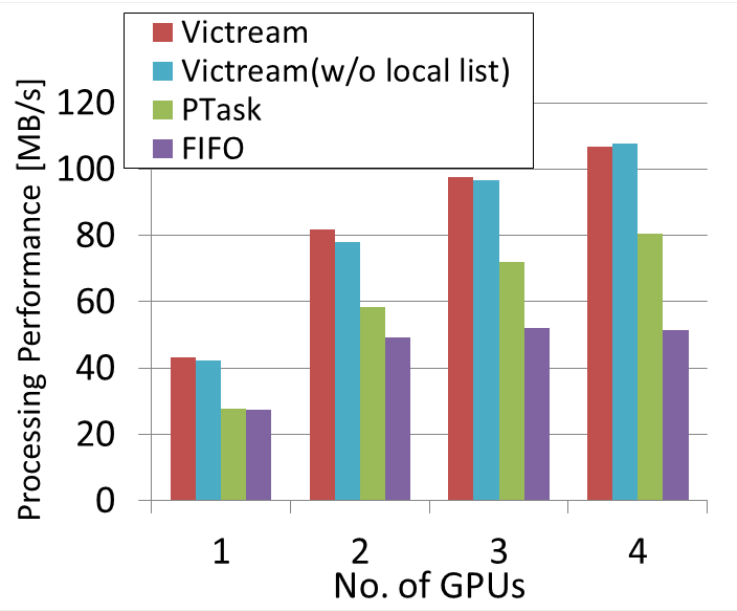


(b) ソート

図 6.14　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 6.15　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)

表 6.3　ExpEther により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

6.4　本章の評価のまとめ

本章では複数の GPU を用いた 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 Victream を評価した．スタンドアロンサーバを用いた評価では Victream が従来手法と比較して最大 117% 計算性能を向上することを示した．また，Victream は GPU のデータプリフェッチと GPU へのデータ I/O の最小化を同時に実現するために，スケジュール可能なサブタスクの再定義を伴うローカリティアウェアスケジューリングの拡張を行っている．評価ではこの新しいローカリティアウェアスケジューリングにより，拡張前のローカリティアウェアスケジューリングと比較して最大 38% 性能が向上することを示した．これらの結果から Victream は従来手法と比較して GPU のデータ I/O が性能ボトルネックとなる Out-of-Core 処理で GPU の演算リソースを効率的に使用することで優れた計算性能を実現することを示した．また GPU 数を増加させたときに従来手法より優れた計算性能の向上が実現できることを示した．

本章ではさらに，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 のスケジューラが実現するスケジューリング性能について計算シミュレーションで評価を行った．シミュレーションでは選択し得るスケジュールを全探索し，求めた最良のスケジュールと Victream が選択するスケジュールの計算時間を比較した．その結果評価に用いたベンチマークでは Victream が実現するスケジュールの計算時間は最良のスケジュールの計算時間から最大 10% 以内の増加に抑えられることを示した．Victream が対象とする応用ではユーザ定義関数の計算コストを予め知ることができず DAG 計算の実行前に最良のスケジュールを計算することができない．またもしユーザ定義関数の計算コストが分かった場合でも，組み合わせ爆発のため最良スケジュールの求解でスケジュールを全探索することは現実的ではない．そのような背景の中，Victream は最良のスケジュールに対してわずかな計算時間の増加を許容すれば，GPU の計算のオーバヘッ

ドとならない高速なオンラインスケジューリングを実現できることを示した．

また ExpEther を用いて GPU をホストから分離したシステムにおける Victream の評価でも，Victream は性能ボトルネックとなるイーサネットで通信されるデータ I/O 量を抑制し，従来手法より優れた計算性能を実現することがわかった．GPU をホストから分離したシステムでは，GPU をホストの I/O スロットに挿入した場合と比較して 50%-100% の性能が得られた．そのため計算プラットフォームにおいて GPU の平均使用率が概ね 50% 以下の場合，I/O デバイスを分離したプラットフォームを採用することでリソース利用効率が向上する可能性があることがわかった．

第 7 章

結論

7.1 本論文のまとめ

本論文では PCI Express (PCIe) をイーサネットに拡張することで I/O デバイスをホストから分離する ExpEther を提案した．ExpEther によりイーサネットに接続した I/O (Input/Output) デバイスのリソースプールから必要に応じて柔軟にホストに I/O デバイスを割り当てることができる．さらに I/O デバイスとして近年注目されている GPU (Graphics Processing Unit) を用いた場合，計算の性能ボトルネックとなる GPU へのデータ I/O を抑制し，GPU の演算リソースを効率的に用いることで従来手法より優れた計算性能や GPU 数に対するより大きな計算性能の向上を実現する GPU ミドルウェア Victream を提案した．

ExpEther は ExpEther ブリッジのハードウェアを用いてイーサネットを PCIe ネットワークとして仮想化する技術である．そのため従来の OS (Operating System) やデバイスドライバ，PCIe に準拠した I/O デバイスを変更することなく使用できる．また，イーサネットで PCIe が要求する高信頼低遅延通信を実現するため，ブリッジ間の End-End で再送制御と送信レート制御による輻輳制御を行っている．また，I/O デバイスのホストの割り当てにはイーサネットの VLAN を利用している．これらの手法により標準のイーサネットを用いて高信頼低遅延の PCIe ネットワークの拡張を実現している．本論文ではさらに，複数の I/O パケットを集約し，1 つのイーサネットフレームにカプセル化することでイーサネットの PCIe 帯域を向上する手法を提案した．提案手法の特徴はイーサネットフレームに集約する I/O パケットの数を適応的に決めることである．これにより，集約する I/O パケットには追加の伝送遅延を与えずに，イーサネットの PCIe 帯域が接続する I/O デバイスの性能ボトルネックとなるときだけ I/O パケットの集約を行い，性能

ボトルネックを緩和することができる．

また提案した ExpEther の有効性を検証するためプロトタイプを作成して性能検証を行った．プロトタイプには PCIe Gen2 x8 と 10GbE 2 ポートのインターフェースを実装した．評価では今日のデータセンターで広く使用されている汎用 I/O デバイスである PCIe SSD (Solid State Drive) と GPU をホストから分離し，提案技術が広くデータセンターに適用可能である事を示した．また性能評価では I/O デバイスを分離するための性能オーバヘッドが 20% 以内であることを確認した．さらに複数の I/O パケットを集約する手法では，I/O デバイスの性能を最大 41% 向上できることを示した．これらの結果は提案した ExpEther が I/O デバイスを分離するためのオーバヘッドは小さく，多くのシステムで許容範囲内であることを示している．

また Victream では，計算で処理するデータ容量が使用する GPU のメモリより大きい Out-of-Core 処理に着目し，複数の GPU を用いて効率的な Out-of-Core 処理を自動で実行できるミドルウェアを提案した．Out-of-Core 処理では GPU メモリのデータを入れ替えるデータスワップに関するデータ I/O が性能ボトルネックとなる．Victream は GPU へのデータ I/O 量の最小化を実現した．

GPU を用いた計算の特徴は，GPU で将来実行するタスクの入力データを演算の実行と並行してロードするデータプリフェッチを行う事である．そこで Victream ではデータプリフェッチと GPU へのデータ I/O の最小化を同時に実現する新しいスケジュール手法を採用している．まず GPU へのデータ I/O を最小化するため，スケジュールするタスクの選択に，ローカリティアウェアスケジューリングを用いた．すなわち，スケジュール候補のサブタスクの中で GPU メモリが保持するデータを入力データとしてできるだけ再利用し，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選択した．さらに，Victream は GPU のデータプリフェッチを行うため，ローカリティアウェアスケジューリングを拡張した．拡張したローカリティアウェアスケジューリングでは，将来のタスクの入力データのフェッチを可能にするため，将来のタスクをスケジュール可能とした．またそのためにスケジュール可能なサブタスクの再定義を行った．

スタンドアロンサーバを用いた Victream の評価では，Victream が従来手法より最大 117% 優れた計算性能を実現することを示した．それにより 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大きな計算性能の向上が得られることがわかった．また Victream 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が，コンピュータシミュレーションによるスケジュールの全探索で得られた最良のスケジュールと比較したところ，Victream のスケジューラは最良のスケジューラに対して評価に用いたベンチマークで計算時間の増加を

10% 以内に抑えられることを示した．さらに ExpEther と Victream を用いた結合評価では，ホストから分離した GPU に Victream を適用した場合，GPU をホストから分離した性能オーバヘッドによりアプリケーションによって性能が概ね 50%-100% となる事が分かった．この結果から，GPU の平均稼働率が 50% 以下のシステムでは，提案手法により GPU を分離した方がハードウェアリソースが効率的に運用できる可能性があることを示した．またそのとき，GPU をホストから分離するための性能オーバヘッドを補うために，最大で 2 倍の数の GPU をホストに割り当てる必要がある．

7.2 今後の研究課題

本研究では PCIe ネットワークの拡張に有線のイーサネットを用いる手法を示した．今後はネットワークとして無線を用いる研究の方向性が考えられる．また，本研究では PCIe の I/O パケットをハードウェアのブリッジでイーサネットフレームにカプセル化する手法を示した．この手法ではホストと I/O デバイスをそれぞれイーサネットに接続するために専用のハードウェアが必要である．これに対しホスト側は専用のハードウェアを用いず，ソフトウェアで I/O パケットを作成しイーサネットフレームにカプセル化した後，ホストが保持する標準の NIC (Network Interface Card) でイーサネットに送信する研究の方向性が考えられる．筆者はこれらの無線化やソフトウェア化の検討に [27] や [64] で他の研究者らと共同で着手している．

またイーサネットを用いた PCIe の拡張では，PCIe をイーサネットでトンネリングした．このとき，低信頼のイーサネットで高信頼低遅延の通信を実現するため，カプセル化を行うネットワークの End-End 間で再送輻輳制御機能を実現した．このようなネットワークのデータ転送機能は PCIe 以外のトラフィックをトンネリングする場合も有効である．実際，本論の第 1 章でも議論したインダストリー 4.0 では，各工場のインターコネクションを実現する規格は標準化が十分進んでおらず装置の製造者毎に独自規格が多い状況である．これらの様々な規格をイーサネットで延長する手段として本研究の高信頼低遅延機能を拡張する研究の方向性が考えられる．

GPU ミドルウェアに関しては，現在の提案技術は Out-of-Core 処理におけるデータスワップデバイスをホストのメインメモリとしている．このような構成では，複数の GPU を用いた場合，GPU のデータのスワップ先がメインメモリに集中し，複数の GPU とメインメモリを接続する I/O バスが性能ボトルネックとなる．そこでこのような課題を解決するため，近年データセンターで注目されている高速フラッシュデバイスである NVMe (Non-Volatile Memory Express) を複数用いる研究の方向性が考えられる．本研究で提

案したイーサネットによる技術で複数の GPU と NVMe を接続し，GPU 毎のデータスワップ先として別の NVMe が選択できれば前述の性能ボトルネックの課題を解決できる．そのとき，本研究で提案したスケジュール手法を複数のスワップデバイスを考慮できるように拡張する必要がある．

また，シミュレーションによる提案スケジューラの評価では，提案スケジューラが選択するスケジュールが最良のスケジュールではないことがわかった．その原因として，サブタスクの各 GPU への割り当てが不均一となることや，未実行のサブタスクの入力データが別の GPU からスワップアウト中でありスケジュール可能になるまで待つ必要があることがあった．そこで今後の研究の方向性として，これらの課題を解決するためにスケジュール手法を改良することが考えられる．

また，提案したミドルウェアはホストから分離した複数の GPU と接続するシステムを対象としているが，現在は 1 台のホストのみ考慮している．従って，ミドルウェアの対象を複数のホストがそれぞれの GPU を保持するシステムに拡張する研究の方向性が考えられる．そのような構成では，ミドルウェアが分散構成となること，またデータの通信を行う通信路の帯域がホスト間のネットワークと，ホスト内のメインメモリと GPU を接続する I/O バスとの間で不均一となることなどの違いが生じるため，それらを考慮したミドルウェアアーキテクチャとスケジューリング手法の拡張が必要になる．

謝辞

東京大学の喜連川優教授には私が 30 代で将来の研究の方向性を考える余地が大きい貴重な時期に，企業の研究開発に偏っていた私に研究活動を考え直す大変意義深い機会をいただきました．研究に対する考え方や，パラダイムを変化させるための視点について熱心にご指導いただきました．ご教授いただいたことが本論文に関する検討で全て体得できたとは到底言いきれませんが，今後より大きな研究成果を実現していくために本研究で得た手がかりを実践していきたいと思います．

NEC の岩田淳氏には私が NEC に入社後，本論文の研究分野に集中して取り組むための環境を与えていただきました．また企業の研究所における研究の実践方法を最初に教えていただいたのも岩田氏でした．国内外の関連技術のベンチマーキングや同業他社との協業方法についても教えていただきました．新入社員でまだわからないことが多かった私に多くの成長する機会や励ましの言葉をいただきました．

NEC の吉川隆士氏には私が主体的に研究活動に取り組める環境を与えていただきました．多くの学会や研究会に参加させていただき，20 代の頃に自分が取り組んでいるテーマに関して課題意識を養うと共に，次のことに自らチャレンジする姿勢を学ばせて頂きました．

NEC の飛鷹洋一氏と樋口淳一氏とは本研究の ExpEther について研究開発の立ち上げ期から製品化まで 10 年以上取り組んできました．様々な紆余曲折がありましたが，本研究の成果の一部は NEC の製品として世の中のシステムに使用いただくまでになりました．それまでの間，研究開発から実用化まで一生懸命一緒に取り組ませていただけたことに感謝いたします．またその中でも私は研究や学会活動といった自分にとって興味が深い分野で多くの取り組みをさせていただきました．

NEC の宮川伸也氏と竹中崇氏には私が Victream の研究着手時に方向性を定めるまで時間がかかる中，活動にご理解をいただき，検討時間を十分確保するご配慮をいただきました．Victream で一定の結論を出せたのも，最初の時期に集中して検討する機会をいた

だけたおかげです．

NEC の荒木拓也氏には Victream の開発に当たって技術面で多くのサポートをいただきました．Victream の RPC(Remote Procedure Call) は荒木氏が開発されたコードによるものです．また GPU で実行するユーザ定義関数をシリアライズしてライブラリからランタイムに送信する方法など，ミドルウェアを実現する上でそれまで私が保有していなかった実現技術を多数教えていただきました．

ウェルモの菅真樹氏には Victream を開発する上で様々な技術的アドバイスをいただきました．また菅氏が開発されていた ExpEther を用いた Key-Value Store と得られた知見を共有することで Victream ミドルウェアに関する検討をより深めることができました．

NEC の林佑樹氏は Victream のスケジューリング方式の研究開発に当たって多くの技術議論にのっていただきました．Victream のコア技術であるスケジューリング方式を確立できたのも，一番深く議論にのっていただいた林氏のおかげです．また，ExpEther を派生させた無線版やソフトウェア版の検討でも研究開発の推進に大変尽力いただいています．ありがとうございました．

参考文献

[1] Multi-Root I/O Virtualization and Sharing Specification Revision 1.0, 2008.

[2] RDMA Consortium, 2009.

[3] Single Root I/O Virtualization and Sharing Specification Revision 1.1, 2010.

[4] Synergies in Server and I/O Virtualization, Micron's IOV Switches Address I/O Limitations in Virtualized Servers. White Paper, July 2012.

[5] Thunderbolt Technology, 2012.

[6] PCI Express Base Specification Revision 3.1, 2014.

[7] Intel Ethernet Switching Components, 2015.

[8] DX2000 -Scalable Modular Server-, 2018.

[9] Martín Abadi, et al. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[10] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pp. 1–12. ACM, 2000.

[11] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, Vol. 10, p. 24, 2010.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, Vol. 23, No. 2, pp. 187–198, 2011.

[13] C Bo, R Hou, J Wu, T Jiang, and L Zhang. TCNet: Cross-node Virtual Machine Communication Acceleration. In *Proc. ACM Int. Conf. on Computing Frontiers*

(CF'13), Ischia, May 2013.

[14] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, Vol. 38, No. 1-2, pp. 37–51, 2012.

[15] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, Vol. 35, No. 1, pp. 38–53, 2009.

[16] Z Cao, X L Liu, Q Li, X. B. Liu, Z. Wang, and X. J. An. An Intra-Server Interconnect Fabric for Heterogeneous Computing. *Journal of Computer Science and Technology*, Vol. 29, pp. 976–988, November 2014.

[17] M C Castro, P Dely, J Karlsson, and A Kassler. Capacity Increase for Voice over IP Traffic through Packet Aggregation in Wireless Multi-hop Mesh Networks. In *Proc. Future Generation Communication and Networking (FGCN2007)*, pp. 350–355, Jeju, December 2007.

[18] Sangyeun Cho and Lei Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *39th Annual IEEE/ACM International Symposium on Microarchitecture 2006 (MICRO-39)*, pp. 455–468. IEEE, 2006.

[19] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM Computer Communication Review*, Vol. 44, pp. 443–454. ACM, 2014.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, Vol. 51, No. 1, pp. 107–113, 2008.

[21] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, Vol. 21, No. 02, pp. 173–193, 2011.

[22] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR*, p. 8, 2009.

[23] Max Grossman and Vivek Sarkar. SWAT: A Programmable, In-Memory, Distributed, High-Performance Computing Platform. In *Proceedings of the 25th*

ACM International Symposium on High-Performance Parallel and Distributed Computing, pp. 81–92. ACM, 2016.

[24] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *ACM SIGPLAN Notices*, Vol. 45, pp. 341–342. ACM, 2010.

[25] T Hanawa, T Boku, S Miura, M Sato, and K Arimoto. PEARL: Power-aware, Dependable, and High-Performance Communication Link using PCI Express. In *Proc. 2010 IEEE/ACM Int. Conf. on Cyber, Physical and Social computing (CPSCom)*, pp. 284–291, Hangzhou, December 2010.

[26] Jun Hasegawa, Hiroyuki Yomo, Yoshihisa Kondo, Peter Davis, Ryutaro Suzuki, Sadao Obana, and Katsumi Sakakibara. Bidirectional packet aggregation and coding for VoIP transmission in wireless multi-hop networks. In *Proc. IEEE International Conference on Communications 2009, ICC'09*, pp. 1–6. IEEE, 2009.

[27] Yuki Hayashi, Jun Suzuiki, Masaki Kan, Takashi Yoshikawa, and Shinya Miyakawa. Delay-bounded transport using rateless codes for I/O bus over wireless Ethernet. In *Proc. 2016 IEEE International Conference on Communications (ICC)*, pp. 1–6. IEEE, 2016.

[28] Bingsheng He, et al. Mars: a MapReduce Framework on Graphics Processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 260–269. ACM, 2008.

[29] R. Hou, T. Jiang, L. Zhang, P. Qi, J. Dong, H. Wang, X Gu, and S. Zhang. Cost Effective Data Center Servers. In *Proc. 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, pp. 179–187, Shenzhen, February 2013.

[30] IDC. IDC WorldWide Quarterly Cloud IT Infrastructure Tracker, Q4 2015. `https://www.networkworld.com/article/3054636/cloud-computing/idc-cloud-is-eating-legacy-systems.html`.

[31] Michael Isard, et al. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *ACM SIGOPS Operating Systems Review*, Vol. 41, pp. 59–72. ACM, 2007.

[32] A Jain, M Gruteser, M Neufeld, and D Grunwald. *Benefits of packet aggregation in ad-hoc wireless network*. PhD thesis, Univ. of Colorado, Boulder, August 2003.

[33] Jonas Karlsson, Andreas Kassler, and Anna Brunstrom. Impact of packet ag-

gregation on TCP performance in wireless mesh networks. In *Proc. IEEE International Symposium on aWorld of Wireless, Mobile and Multimedia Networks & Workshops 2009, (WoWMoM 2009)*, pp. 1–7. IEEE, 2009.

[34] Andreas Kassler, Marcel Castro, and Peter Dely. VoIP packet aggregation based on link quality metric for multihop wireless mesh networks. In *Proceedings of the Future Telecommunication Conference, Beijing, China*, 2007.

[35] Shinpei Kato, et al. Gdev: First-Class GPU Resource Management in the Operating System. In *USENIX Annual Technical Conference*, pp. 401–412, 2012.

[36] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pp. 17–30, 2011.

[37] Tamer Khattab, Amr Mohamed, Ayman Kaheel, and Hussein Alnuweiri. Optical packet switching with packet aggregation. In *IEEE International Conference on Software, Telecommunications, and Computer Networks (SOFTCOM)*, 2002.

[38] Kyungtae Kim, Samrat Ganguly, Rauf Izmailov, and Sangjin Hong. On packet aggregation mechanisms for improving VoIP quality in mesh networks. In *Proc. IEEE 63rd Vehicular Technology Conference 2006, (VTC 2006)*, Vol. 2, pp. 891–895. IEEE, 2006.

[39] V Krishnan. Evaluation of an Integrated PCI Express IO Expansion and Clustering Fabric. In *Proc. 16th IEEE Symposium on High Performance Interconnects*, pp. 93–100, Stanford, CA, August 2008.

[40] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pp. 31–46. USENIX Association, 2012.

[41] Kyungsoo Lee, Sangki Yun, Inhye Kang, and Hyogon Kim. Hop-by-hop frame aggregation for VoIP on multi-hop wireless networks. In *Proc. IEEE International Conference on Communications 2008 (ICC'08)*, pp. 2454–2459. IEEE, 2008.

[42] Yuxia Lin and Vincent WS Wong. WSN01-1: frame aggregation and optimal frame size adaptation for IEEE 802.11 n WLANs. In *IEEE Global telecommunications conference 2006 (GLOBECOM'06)*, pp. 1–6. IEEE, 2006.

[43] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine

learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, Vol. 5, No. 8, pp. 716–727, 2012.

[44] NVIDIA. NVIDIA TESLA P100 GPU ACCELERATOR. `http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf`, 2016.

[45] NVIDIA. GPU-Accelerated Libraries. `https://developer.nvidia.com/gpu-accelerated-libraries`, 2017.

[46] NVIDIA. CUDA Toolkit. `https://developer.nvidia.com/cuda-toolkit`, 2018.

[47] Chris Padwick, MICHAEL Deskevich, Fabio Pacifici, and Scott Smallwood. WorldView-2 pan-sharpening. In *Proceedings of the ASPRS 2010 Annual Conference, San Diego, CA, USA*, Vol. 2630, 2010.

[48] Rajesh Palit, Kshirasagar Naik, and Ajit Singh. Impact of packet aggregation on energy consumption in smartphones. In *2011 7th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 589–594. IEEE, 2011.

[49] Borja Pérez, Esteban Stafford, Jose Luis Bosque, Ramon Beivide, Sergi Mateo, Xavier Teruel, Xavier Martorell, and Eduard Ayguadé. Extending OmpSs for OpenCL kernel co-execution in heterogeneous systems. In *Proc. 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 1–8. IEEE, 2017.

[50] Judit Planas, Rosa M Badia, Eduard Ayguade, and Jesus Labarta. Self-adaptive ompss tasks in heterogeneous environments. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 138–149. IEEE, 2013.

[51] H. Raj and K. Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *Proc. IEEE Int. Symposium on High Performance Distributed Computing (HPDC'07)*, Monterey, CA, June 2007.

[52] Christopher J Rossbach, et al. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 233–248. ACM, 2011.

[53] Christopher J Rossbach, et al. Dandelion: a Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 49–68. ACM, 2013.

[54] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. *ACM SIGPLAN Notices*, Vol. 40, No. 7, pp. 115–126, 2005.

[55] H. Shimonishi, J. Higuchi, T. Yoshikawa, and A. Iwata. A Congestion Control Algorithm for Data Center Area Communications. In *Proc. IEEE CQR Int. Workshop*, 2008.

[56] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Out-of-core GPU memory management for MapReduce-based large-scale graph processing. In *Proc. 2014 IEEE International Conference onCluster Computing (CLUSTER)*, pp. 221–229. IEEE, 2014.

[57] D Skordoulis, Q Ni, H H Chen, A P Stephens, C Liu, and A Jamalipour. IEEE 802.11n MAC Frame Aggregation Mechanisms for Next-Generation High-Throughput WLANs. *IEEE Wireless Communications*, Vol. 15, pp. 40–47, February 2008.

[58] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: data orchestration and tuning for OpenCL devices. In *European Conference on Parallel Processing*, pp. 275–286. Springer, 2010.

[59] Jeff A Stuart and John D Owens. Multi-GPU MapReduce on GPU clusters. In *Proc. of 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pp. 1068–1079. IEEE, 2011.

[60] Weibin Sun, Robert Ricci, and Matthew L Curry. GPUstore: harnessing GPU computing for storage systems in the OS kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*, p. 9. ACM, 2012.

[61] Narayanan Sundaram, et al. A framework for efficient and scalable execution of domain-specific templates on GPUs. In *IEEE International Symposium on Parallel & Distributed Processing 2009, (IPDPS 2009)*, pp. 1–12. IEEE, 2009.

[62] J. Suzuki, Y. Hidaka, J. Higuchi, T. Baba, N. Kami, and T. Yoshikawa. Multi-Root Share of Single-Root I/O Virtualization (SR-IOV) Compliant PCI Express Device. In *Proc. IEEE Symposium on High Performance Interconnects (HOTI'10)*, pp. 25–31, Mountain View, CA, August 2010.

[63] J. Suzuki, M. Kan, Y. Hayashi, and T. Yoshikawa. Accelerating SSSP Algorithm on GPU with Adaptive Thread Creation. In *Proc. 2013-DBS-158 (in Japanese)*, Kyoto, Japan, November 2013.

[64] Jun Suzuki, Akira Tsuji, Yuki Hayashi, Masaki Kan, and Shinya Miyakawa. Device-Level IoT with Virtual I/O Device Interconnection. In *Proc. 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 67–74. IEEE, 2016.

[65] C C Tu, C T Lee, and T C Chiueh. Secure I/O Device Sharing among Virtual Machines on Multiple Hosts. In *Proc. 40th Annual Int. Symp. on Computer Architecture (ISCA'13)*, pp. 108–119, Tel Aviv, June 2013.

[66] C C Tu, C T Lee, and T C Chiueh. Marlin: A Memory-Based Rack Area Network. In *Proc. tenth IEEE/ACM Symp. on Architectures for networking and communications systems (ANCS'14)*, pp. 125–136, Marina del Rey, CA, October 2014.

[67] Kaibo Wang, et al. GDM: Device Memory Management for GPGPU Computing. *ACM SIGMETRICS Performance Evaluation Review*, Vol. 42, No. 1, pp. 533–545, 2014.

[68] Ke Wang, Xraobing Zhou, Tonglin Li, Dongfang Zhao, Michael Lang, and Ioan Raicu. Optimizing load balancing and data-locality with data-aware scheduling. In *Proc. 2014 IEEE International Conference onBig Data (Big Data)*, pp. 119–128. IEEE, 2014.

[69] Yaron Weinsberg, Danny Dolev, Tal Anker, Muli Ben-Yehuda, and Pete Wyckoff. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *ACM SIGARCH Computer Architecture News*, Vol. 36, pp. 179–188. ACM, 2008.

[70] Yuan Wen, Zheng Wang, and Michael FP O'boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *Proc. 2014 21st International Conference on High Performance Computing (HiPC)*, pp. 1–10. IEEE, 2014.

[71] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proc. IEEE 13th Int. Symposium on High-Performance Computer Architecture (HPCA-13)*, Phoenix, AZ, February 2007.

[72] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical DAG scheduling for hybrid distributed systems. In *Proc. IEEE International Parallel & Distributed Processing Symposium (IPDPS 2015)*,

2015.

[73] Shun Yao, Fei Xue, Biswanath Mukherjee, SJ Ben Yoo, and Sudhir Dixit. Electrical ingress buffering and traffic aggregation for optical packet switching and their effect on TCP-level performance in optical mesh networks. *IEEE Communications Magazine*, Vol. 40, No. 9, pp. 66–72, 2002.

[74] Richard M Yoo, Christopher J Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pp. 315–325. ACM, 2013.

[75] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *Proc. 2016 IEEE International Conference on Big Data (Big Data)*, pp. 273–283. IEEE, 2016.

[76] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pp. 265–278. ACM, 2010.

[77] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 423–438. ACM, 2013.

[78] Matei Zaharia, et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[79] 総務省. 平成24年版 情報通信白書. `http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h24/pdf/24honpen.pdf`.

[80] 総務省. 平成28年版 情報通信白書. `http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h28/pdf/28honpen.pdf`.

発表文献

査読付き論文誌

- Jun Suzuki, Yoichi Hidaka, Junichi Higuchi, Yuki Hayashi, Masaki Kan, and Takashi Yoshikawa, "Disaggregation and Sharing of I/O Devices in Cloud Data Centers," IEEE Transactions on Comoputers, vol. 65, issue 10, pp. 3013-3026, Oct. 2016.
- Jun Suzuki, Yuki Hayashi, Masaki Kan, Shinya Miyakawa, Takashi Takenaka, Takuya Araki, Masaru Kitsuregawa, "Minimization of Data I/O in Out-of-Core Processing on Multiple GPUs," IEEE Transactions on Parallel and Distributed Systems, (投稿準備中).

国際学会

- Jun Suzuki, Yoichi Hidaka, Junichi Higuchi, Takashi Yoshikawa, Atsushi Iwata, "ExpressEther - Ethernet-Based Virtualization Technology for Reconfigurable Hardware Platform," Proceedings of the 14th IEEE Symposium on High-Performance Interconnects (HOTI'06), Aug. 2006.
- Jun Suzuki, Yuki Hayashi, Masaki Kan, Shinya Miyakawa, Takashi Yoshikawa, "End-to-End Adaptive Packet Aggregation for High-Throughput I/O Bus Network Using Ethernet," Proceedings of the 2014 IEEE 22nd Annual Symposium on High-Performance Interconnects, Aug. 2014.
- Jun Suzuki, Yuki Hayashi, Masaki Kan, Shinya Miyakawa, Takashi Takenaka, Takuya Araki, Masaru Kitsuregawa, "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, pp. 179-188, Dec. 2017, (Best Paper).

その他

- 飛鷹洋一，樋口淳一，下西英之，鈴木順，柳町成行，吉川隆士，岩田淳，”イーサネットを用いたシステム仮想化技術 ExpressEther の提案 (1) システム概要とアーキテクチャ,” 2006 年電子情報通信学会ソサイエティ大会, B-6-57, 2006 年 9 月.
- 鈴木順, 飛鷹洋一, 樋口淳一, 下西英之, 柳町成行, 吉川隆士, 岩田淳, ”イーサネットを用いたシステム仮想化技術 ExpressEther の提案 (2)I/O 仮想化技術,” 2006 年電子情報通信学会ソサイエティ大会, B-6-58, 2006 年 9 月.
- 下西英之, 鈴木順, 飛鷹洋一, 樋口淳一, 柳町成行, 吉川隆士, 岩田淳, ”イーサネットを用いたシステム仮想化技術 ExpressEther の提案 (3) パケット再送方式,” 2006 年電子情報通信学会ソサイエティ大会, B-6-59, 2006 年 9 月.
- 樋口淳一, 下西英之, 鈴木順, 飛鷹洋一, 柳町成行, 吉川隆士, 岩田淳, ”イーサネットを用いたシステム仮想化技術 ExpressEther の提案 (4) 試作と性能評価,” 2006 年電子情報通信学会ソサイエティ大会, B-6-60, 2006 年 9 月.
- 鈴木順，林佑樹，菅真樹，宮川伸也，吉川隆士，”イーサネット拡張 I/O バスにおける複数 DMA パケット集約による帯域向上,” ネットワークシステム研究会, (98)NS, 2014 年 3 月.
- 鈴木順，菅真樹，林佑樹，荒木拓也，宮川伸也，喜連川優，”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,” 第 160 回 DBS・第 131 回 OS・第 35 回 EMB 合同研究発表会, セッション 1-C(23), 2014 年 11 月.
- 鈴木順，菅真樹，林佑樹，宮川伸也，喜連川優，”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,” 2015 年並列/分散/協調処理に関する『別府』サマー・ワークショップ (SWoPP2015), 2015 年 8 月.
- 鈴木順，菅真樹，林佑樹，荒木拓也，宮川伸也，喜連川優，”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,” 2016 年並列/分散/協調処理に関する『松本』サマー・ワークショップ (SWoPP2016), 2016 年 8 月.
- 鈴木順，林佑樹，荒木拓也，竹中崇，喜連川優，”GPU ミドルウェア”Victream”における I/O とプロセッシングの連携スケジューラの性能評価,” 2017 年並列/分散/

協調処理に関する『秋田』サマー・ワークショップ (SWoPP2017), 2017 年 7 月.

- 鈴木順，林佑樹，荒木拓也，喜連川優，”GPU ミドルウェア”Victream”のシミュレーションによる性能評価,” 2018 年並列/分散/協調処理に関する『熊本』サマー・ワークショップ (SWoPP2018), 2018 年 7 月, (発表予定).

受賞

- Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, Dec. 2017, Best Paper Award.
- 情報処理学会 2017 年度 (平成 29 年度) 山下記念研究賞.