Vol.2012-ARC-201 No.25
2012/8/2

CPU/GPU 間データ通信向け先読み機構の検討

薦田 登志矢[1,a)]　三輪 忍[1,b)]　中村 宏[1,c)]

概要：HPC を中心として，GPU コンピューティングの重要性が高まっている．一般的な構成の GPU コンピューティングシステムでは，汎用 CPU と GPU が物理的に異なるメモリを持ちこれらがシステムバスを介して接続される．これまでシステムバスにおけるデータ転送オーバーヘッドは，プログラマがアプリケーションの特性を考慮しつつデータ転送処理を最適化することで対処されてきた．しかし，手動によるデータ通信の管理・最適化はアプリケーション開発の生産性を大きく低下させることから，このようなデータ転送処理の自動化・自動最適化が望まれている．そこで本研究では，システムメモリとグラフィクスメモリの間で生じるデータ転送を対象とし，自動で計算と転送の並列実行を実現する先読み機構を提案する．提案システムは，アプリケーションのデータ通信パターンを実行時に解析し，次に転送対象となるデータを予測する．予測対象データは，非同期転送を用いて計算処理の裏で GPU 上のメモリへと先読みされる．本稿ではこのような先読み機構の設計と実装を示し，初期評価実験の結果を通じて性能向上の可能性を検討する．

1. イントロダクション

近年，高い計算能力を必要とする応用分野において GPU コンピューティングが注目を集めている．これは，データ並列性を有する計算処理に最適化されたアーキテクチャを持つ GPU を利用することで通常の CPU に比べて，高い性能を少ない消費電力で実現できるためである．

一般的な構成の GPU コンピューティングシステムでは，システムメモリとは別に GPU が物理的に異なるメモリ (グラフィクスメモリ) を持ちこれらがシステムバスを介して接続されている．物理的に異なるメモリを GPU に用意するメモリアーキテクチャは，システムメモリとは異なる性能要求を持つ GPU 上での処理性能を最大化するためには都合がよい．一方でこのようなメモリアーキテクチャでは，物理的に異なるシステムメモリとグラフィクスメモリ間でデータ転送が生じることとなり，しばしば GPU を利用するデータ並列アプリケーションの性能向上を妨げる原因となっている．

現在，GPU コンピューティングにおいて広く用いられている CUDA や OpenCL では *memcpy*() に類似したデータ転送 API を利用して，プログラマが手動で転送を管理するのが一般的である．特に，データ転送が性能向上のボトルネックになる場合にはアプリケーションのメモリアクセスパターンを意識したデータ転送処理の最適化が必要となっている．システムメモリとグラフィクスメモリ間におけるデータ転送処理の最適化は，GPU による性能向上を達成するために重要である一方，手動による最適化作業はヘテロジニアスなメモリアーキテクチャの知識を必要とし，システム開発の生産性を大きく低下させ，GPU コンピューティングにおいて良く知られているアプリケーション開発の難しさの原因の一つである．

このため，GPU の物理メモリの存在を隠蔽し GPU プログラミングにおけるデータ管理の煩雑さを軽減しようとする試みがコンパイラ，およびランタイムシステムを用いてなされてきた [1] [3] [5]．これらのシステムでは，コンパイラが半自動的にデータ転送処理を行うコードを生成したり，またはデータ転送なしでシステムメモリ上のポインタをデータ並列カーネル関数の引数として直接渡すことが可能であり，プログラマによる明示的なデータ転送処理の記述が必要ない．しかし，これら従来のシステムの共通の問題として，データ転送処理遅延を隠蔽するための非同期通信を用いた計算と通信の並列実行が行われていないことが挙げられる．このため，データ転送遅延が性能のボトルネックになるアプリケーションでは，計算と通信の並列実行なしでは十分な性能向上を達成できないため，これらのデータ転送自動化システムを適用することは難しい．

本稿では，先行研究の自動データ転送システムのバックエンドとして動作することを想定し，非同期通信を用い

[1] 東京大学大学院情報理工学系研究科
113–8656 3-5-1 Hongo Bunkyo-Ku Tokyo, Japan
[a)] komoda@hal.ipc.i.u-tokyo.jp
[b)] miwa@hal.ipc.i.u-tokyo.ac.jp
[c)] nakamura@hal.ipc.i.u-tokyo.ac.jp

て，計算と CPU・GPU 間通信の並列実行を先読みによって自動化するシステムを提案する．提案システムは，ユーザープログラムから要求されてくる同期的な通信処理をそのセマンティックを守りつつ自動的に非同期通信に置き換える．本稿では，このような機能を実現するための先読みデータ転送の手順，投機的な先読み転送を可能にするグラフィクスメモリの管理機構，およびデータ並列処理に内在するメモリアクセスの規則性を捉えることができるデータ転送予測器を提案し，プロトタイプシステムによる初期評価の結果を報告する．

本稿の構成は以下のようになっている．まず，第 2 章で GPU が混在するコンピュータシステムのメモリ階層について述べ，グラフィクスメモリとシステムメモリの間で発生するデータ移動を自動化する試みとその課題を説明する．続く，第 3 章で本研究が提案する CPU・GPU 間のデータ通信向け先読み機構を提案し，その内部構成について詳しく述べる．第 4 章において，プロトタイプシステムを用いた実験の詳細と結果を示す．第 5 章で関連研究を説明し，第 6 章でまとめと今後の課題を述べる．

2. 背景

2.1 グラフィクスメモリとメモリ階層

ハイエンドな GPU ではデータ並列処理に合わせて特別にチューニングされたグラフィクスメモリを用いている．GPU のような SIMD 型のデータ並列処理が要求する高い性能を供給するためには，従来の DRAM では十分でないためであり，GPU ではそのメモリコントローラもデータ並列処理の特性に合わせて最適化されたものとなっている [2]．高いスループット性能を達成するために，グラフィクスメモリは GPU と同一のボード上に直接搭載され GPU に接続されるが，これにより DIMM モジュールを介して接続されるシステムメモリと比較して高性能化が容易になる一方，同時にメモリの大容量化が難しくなる．表 1 では，2011 年のデータをもとにシステムメモリとグラフィクスメモリのスペックを比較している．大規模なデータを扱うシステムでは，システムメモリ容量がグラフィクスメモリの数倍から十数倍となることもある．

GPU コンピューティングにおけるグラフィクスメモリは，GPU の最下位キャッシュとシステムメモリの間に存在する中間的なメモリ階層と考えることができる．

	スループット	容量 (GB)	Demand Paging
System Memory(DDR on DIMM)	～52GB/sec	～256	Yes
Graphics Memory(GDDR)	～192GB/sec	～6	No

表 1 システムメモリとグラフィクスメモリの比較

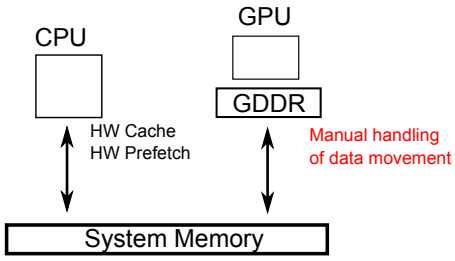

図 1 CPU/GPU システムのメモリ階層 (シングル GPU)

CPU/GPU が混在するコンピュータシステムにおけるメモリ階層を図 1 に示す．良く知られているように，コンピュータシステムにおけるメモリシステムは上位にあるメモリほど小容量で高速アクセス可能なメモリとなっているが，グラフィクスメモリも例外ではない．一方でメモリ階層間のデータ転送方式を考えた場合，グラフィクスメモリとシステムメモリの間はその他多くのメモリ階層と異なり，非透過的なものとなっている．すなわち，グラフィクスメモリとシステムメモリの間のデータ転送はアプリケーション開発者が手動で行うことが主流となっている．

2.2 CPU・GPU 間通信の自動化に関する先行研究

CUDA や OpenCL を用いてアプリケーションを開発する場合，限定されたグラフィクスメモリサイズとアプリケーションのメモリアクセス特性を考慮しつつ，システムメモリとグラフィクスメモリ間でのデータ転送を記述しなければならない．システムメモリとグラフィクスメモリの間の転送性能が限定的であることを考えた場合，このような手動によるデータ移動管理はデータ通信がアプリケーション性能のボトルネックにならないようデータ転送処理を最適化するのには都合がよい．一方で，分離された 2 つのメモリ空間の間で必要となるデータ転送処理の記述は，多くの開発者にとって GPU を利用したプログラミングの難しさを高め，開発生産性を低下させる原因となっているのも事実である．このため，グラフィクスメモリを対象としたデータ転送の自動化を目標とした試みがこれまでにも行われてきた．ここでは，コンパイラによるアプローチおよび動的なメモリ管理機構によるアプローチに分けてこれらの試みについて概略を述べる．

2.2.1 コンパイラによる自動化

GPU コンピューティングをより簡単に利用できる環境を提供することを目的として，ディレクティブベースの並列化コンパイラが研究開発されている．この中で，CPU と GPU の間で必要になるデータ転送もコンパイラにより自動的に処理される．

GPU コンピューティングのプログラミング環境に関する初期の研究において，OpenMP ディレクティブから GPU カーネルコードを生成する試みがなされた [5] ．この研究では，データ転送処理と GPU 上でのデータ並列処理を逐次

的に行うシンプルなデータ転送方式を用いている．近年このような研究の延長として，OpenMP に GPU コンピューティング特有のディレクティブを追加しようとする試みがなされている．特に OpenACC は次世代の GPU プログラミング環境として期待されている．OpenACC では，前述の素朴なデータ転送方式によるアプリケーション性能低下を防ぐため，データ転送を最適化するためのヒント情報を記述するためのディレクティブをプログラマに提供している．

2.2.2 ダイナミックなメモリ管理機構による自動化

コンパイラによるアプローチに対して，ユーザーからのヒント情報を用いず，ランタイムシステムによって得られる動的な情報のみを用いてシステムメモリとグラフィクスメモリ間でのデータ通信を自動化しようとする試みがなされている．これらのシステムでは，GPU 上ではデマンドページングを利用できないという制約の中で効率的に自動データ転送を実現するメカニズムについて研究が行われている．

GMAC [1] は，システムメモリ・グラフィクスメモリ間のデータ転送を自動化する初期のシステムとして知られている．GMAC では，CPU 上での処理および GPU 上での処理で共有されるデータ領域を確保する特別なメモリ確保 API を提供することで，データ転送の候補となるシステムメモリ上の領域を判別する．このデータ領域に対して，グラフィクスメモリ上にコピーが自動的に作成され，GPU 上での処理はこのコピーデータに対して実行される．このとき，システムメモリとグラフィクスメモリで共有されるデータのコヒーレンスをとる必要があるが，グラフィクスメモリとシステムメモリの非対称性を利用した効率の良いコヒーレンス機構が提案されている．CGCM [3] では，共有データ領域を明示的なメモリ確保 API を用いて宣言しなければならなかった GMAC の制約を取り払い，システムメモリ上のデータに対してデータ並列カーネルを実行することを可能にしている．グラフィクスメモリとのデータ転送は，データ並列処理が実行されるタイミングで，自動的に処理される．CGCM ではデータ転送の単位が一度の *malloc*() によって返されるメモリ領域全体 (アロケーションユニット) となっており，非常に粗粒度でのデータ転送管理方式となっている．

2.3 先行研究の問題点

前述の先行研究では，システムメモリとグラフィクスメモリの間のデータ転送を機能的に自動化できる一方性能の観点から考えた場合課題がある．特に非同期転送を用いて計算の裏に通信遅延を隠蔽する性能最適化を効果的にサポートしていないことは，大規模データに対するストリーム処理のようにデータ転送が頻発しその遅延が問題となるアプリケーションへの応用を難しいものとしている．

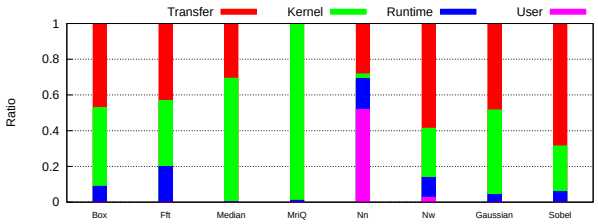

図 2 転送時間が全体の実行時間に占める割合

図 2 に，いくつかの GPU を利用するベンチマークアプリケーションにおいてグラフィクスメモリとシステムメモリ間のデータ転送遅延がアプリケーション全体の実行時間に占める割合を示す．図中，*Transfer* はデータ転送，*Kernel* はデータ並列処理，*Runtime* が GPU ドライバ等のランタイムシステムの起動オーバーヘッド，*User* がその他の CPU 上での処理の時間をそれぞれ表している．実行時間の計測は 4 章で詳述するプラットフォームで行っており，アプリケーションは NVIDIA SDKSample から 4 つ，Rodinia，Parboil ベンチマークスイートから各 2 つずつのアプリケーションを使用している．これらのアプリケーションでは，データ転送および GPU 上でのデータ並列処理はシーケンシャルに実行される実装となっている．図から，GPU を利用するアプリケーションではデータ通信遅延が無視できない割合を占めることも多く，データ並列処理だけでなくデータ通信遅延までを考慮した性能最適化が重要であることが分かる．

3. CPU・GPU 間通信向け先読み機構

ここでは，アプリケーションプログラムもしくは先行研究の自動データ転送システムが生成するデータ転送リクエスト系列を受け取り，先読みを用いて計算と通信の並列実行を自動的に実現するデータ転送管理機構を提案し，その詳細について述べる．

3.1 システム概要

図 3 に，提案する先読み機構を含むシステム階層図を示す．提案システムは，上位レイヤのシステムが生成したデータ転送のリクエスト系列を受け取り，先読みリクエストが挿入されたデータ転送系列を下位レイヤの GPU ドライバへ発行する役割を担う．上位レイヤにおいて提案システムを利用するユーザープログラムとしては，直接 CUDA や OpenCL で記述されたアプリケーション，もしくはディレクティブコンパイラや GMAC のような自動データ転送システムのどちらでも対応することが可能である．

次に提案システムによって，データ転送処理が具体的にどのように変換されるのかを簡単な例を用いて説明する．今，上位レイヤから 1. CPU から GPU 方向へのデータ転送，2. データ並列処理の実行，3. GPU から CPU 方向へのデータ転送，の順に処理がリクエストされたとする．図 4

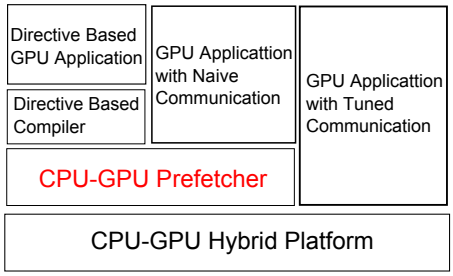

図 3 CPU-GPU プリフェッチャのシステム階層

では，このようなリクエストがシーケンシャルに処理される様子をシーケンス図を用いて表している．ここで，*User* は提案システムのユーザープログラム，*Runtime*＋*Driver* は GPU コンピューティングにおけるランタイムシステムライブラリ，*Bus* はグラフィクスメモリとシステムメモリを接続するデータ転送路を表している．

ここで 2. データ並列処理の実行，と 3.CPU から GPU 方向へのデータ転送の間にデータ依存関係がない場合，この 2 つの処理は並列実行可能である．図 5 ではこの並列性を利用して，提案システムがデータ転送処理を計算処理の裏に隠蔽する様子をシーケンス図を用いて表している．*Prefetcher* が，提案する先読み機構を表している．提案システムでは，上位レイヤからデータ転送リクエストを受け取った場合，まず当該データがすでに先読みされていないかをチェックする．図中では，一回目のデータ転送リクエストでは当該データは先読みされておらず (図中，Prefetch Miss)，二回目のデータ転送リクエストでは当該データが先読みされている (図中，Prefetch Hit)．当該データが先読みされていなかった場合には，上位レイヤからのデータ転送リクエストをそのまま GPU ドライバに伝え，シーケンシャルにデータ転送処理を行う．一方，当該データがすでに先読みされていた場合にはリクエストされたデータ転送を行う必要はないため，リクエストされたデータ転送をキャンセルすることができ，これが提案システムによる性能向上の原理である．. なお，先読みは上位レイヤからのデータ転送リクエストを受け取るたびに行われる (図中，Prefetch Next Data)．

3.2 投機的なデータ転送機構

提案システムは，GPU 上での処理と並列に先読み通信を行うために特別なメモリ領域 (Shadow Buffer) をグラフィクスメモリ上に確保する．図 6 では，システムメモリ上のデータが Shadow Buffer 領域を転送先として先読みされている様子を示している．Shadow Buffer 領域は，GPU 上で実行されるデータ並列処理によって使用されることはないため，安全に先読み転送を行うことができる．

上位レイヤからリクエストされたデータが，すでに転送先メモリに存在する場合，先読みが成功したことになる．このとき，Shadow Buffer 上に先読みされていたデータはポインタのリネーミング，もしくはグラフィクスメモリ上におけるコピー処理を通じてプログラムがアクセスできるグローバルメモリ領域へ移行され，データ並列処理によって使用できるようになる．リネーミングにより Shadow Buffer 内のデータを利用する場合，余分なコピーが発生しないため性能的なメリットがある一方，当該 GPU メモリポインタに関わる全ての処理において GPU メモリポインタをリネーミングし続ける必要がある．また，グラフィクスメモリ上のポインタをデータ値として転送するようなアプリケーションには適用することができない (例えば，GPU 上の処理においてポインタの配列を利用する場合など．) Shadow Buffer から，データ並列処理に用いるメモリ領域へデータをコピーする場合には余分なデータコピーが発生する一方，このようなポインタ利用に関する制約は生じない．なお，グラフィクスメモリ上でのデータコピーはシステムメモリとグラフィクスメモリ間のデータ転送に比べて十分高速であり，先読みによる性能向上はオンチップコピーをする場合でも期待できる．なお，4 章におけるプロトタイプの実装ではリネーミングを用いた実装方式をとっている．

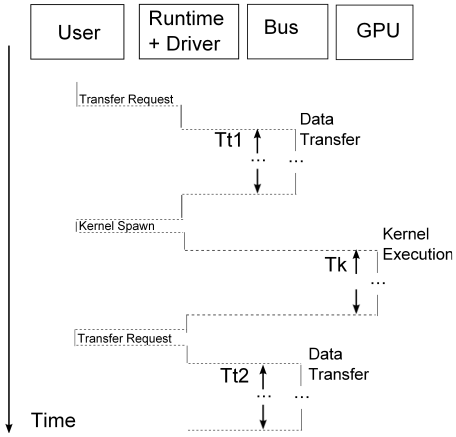

図 4 通常のデータ通信

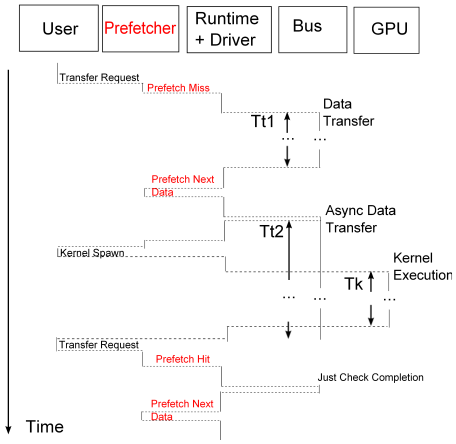

図 5 先読み機構によるデータ通信

先読み機構では，計算処理と通信処理の間の並列性を利

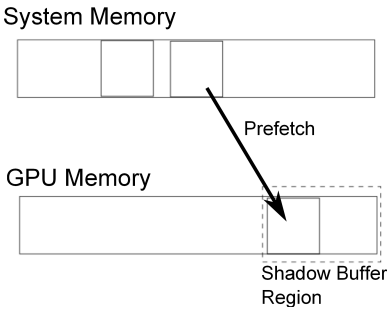

図 6 プリフェッチ用に用いられる GPU メモリ領域

用するために投機的にデータ転送を行う．投機的とはすなわち，先読みによって転送されるデータは転送先の処理において利用されないかもしれない，ということであるがこれには，1. そもそも転送元のデータ領域は使われない場合と，2. 転送元のデータ領域は使われるが，先読みされた後に書き換えられてしまう場合，の 2 パターンが存在する．1. のケースは，単純に先読みしたデータが使われないだけなのでプログラムの正当性の観点から問題はないが，2. のケースでは間違ったデータに基づいて処理が行わる可能性があるため，このようなケースを検出し，先読みしたデータをインバリデートする仕組みが必要である．

提案システムでは，メモリ保護違反命令を利用して先読みされた後に書きかえられたデータに対応する先読みデータをインバリデートする．先読みされるメモリ領域は，先読みと同時に書き込み禁止に設定される．先読みされている最中・またはその後に，当該データ領域に書き込みが行われる場合，メモリ保護違反の例外ハンドラが動作し，先読みされたデータをインバリデートする．これにより，間違った先読みによる間違った処理の実行を防ぐことができる．なお，プリフェッチが成功した場合にはそのタイミングで当該メモリ領域は書き込み可能に設定され，先読みされたメモリ領域への書き込みが可能になる．

3.3 データ転送の予測

前節で説明したメモリ管理機構を用いることで，グラフィクスメモリに関わるデータ転送を先読みすることが可能になる．ここでは，実際にどのデータを先読みするのかを決定する将来のデータ転送の予測方法について述べる．

GPU 上で実行されるデータ並列処理では，グラフィクスメモリ上でのメモリアクセスに非常に高い規則性が存在する [4]．提案システムで考えているのは，さらに下位のメモリ階層であるグラフィクスメモリとシステムメモリの間で生じるデータトラフィックであるが大規模データに対するデータ並列処理では，より上位の GPU メモリ階層で観測されるのと同様のメモリアクセスの規則性が，システムメモリとグラフィクスメモリの間のデータ転送においても存在すると期待できる．

この規則性を効果的に捉えるために，提案システムのデータ転送予測器では前回および前々回のデータ転送内容に基づくストライド予測，およびより長い時間における転送履歴を利用し，必ずしも連続アクセスでないストリームアクセスを検出できるマルコフプリフェッチアルゴリズム [6] の 2 つを利用する．マルコフプリフェッチが利用できる場合にはマルコフプリフェッチの予測結果を用い，そうでない場合はストライドプリフェッチを行う．どちらの予測器も，メモリアクセスの規則性を捉えるための予測器である．

CPU の最下位キャッシュにおけるプリフェッチには存在しない問題として，提案システムでは先読みを行うデータサイズを決定する必要がある．データ転送の予測に用いた転送履歴におけるデータ転送サイズの平均値を用いて，先読みを行うなどの先読みサイズの決定則を考えることができる．

3.4 提案システムによるオーバーヘッド

この章の最後に，提案システムを利用することにより生じる性能オーバーヘッドについてまとめる．提案システムによる性能オーバーヘッドは，使用されないデータの先読みにより生じるミスペナルティと提案システムが実行時に動作することによるオーバーヘッドの 2 種類がある．

使用されないデータの先読み (プリフェッチミス) が生じた場合，まずプログラムから直接リクエストされてきたデマンド転送の性能を阻害するというミスペナルティの発生が考えられる．これは，先読み転送によりバスの競合が増加するためである．また，先読みされたデータが書きこまれた場合，これをインバリデートするために例外ハンドラが起動することになり，これもミスペナルティとなる．これらの性能オーバーヘッドは，先読みデータ転送の予測アルゴリズムを改善することで減らすことができる．

一方で，図 5 に示されているように実行時における提案システムによる処理の実行はアプリケーションの性能オーバーヘッドとなる．これには，先読みデータのチェック処理，未来のデータ転送の予測処理，および先読みのための GPU ドライバ・ランタイムシステムの呼び出し処理に関わるオーバーヘッドなどが挙げられる．システムメモリとグラフィクスメモリ間でのデータ転送単位が十分に大きく時間のかかるものであれば，これらの性能オーバーヘッドは相対的に無視できるほど小さくなるが，小さな単位でデータ転送が発生する場合には，これらのオーバーヘッドは無視できないものとなる．

4. 実験

OpenCL プラットフォーム上に構築したプロトタイプシステムの上で処理を実行し，提案システムの実現可能性と基本的な特性を評価する．なお，今回の実験システムではシステムメモリからグラフィクスメモリ方向でのデータの

先読みのみを行っており，グラフィクスメモリからシステムメモリへ向かう方向のデータ転送に対する先読みは行っていない．これは，GPU 上においてメモリ保護違反を操作するための API が公開されておらず，3 章で説明した機構によってダーティデータの検出機構をそのままでは実装できないためである．GPU 上で動作するデータ並列カーネルがアクセスする領域を静的に解析するなどの方法を用いて，先読みの安全性を確保する他の方法を考えることができるが，これは今後の課題である．

4.1 実験環境

評価に用いたベンチマークアプリケーションは，並列スレッド間でデータのやり取りが全くないデータ並列処理をモデル化したシンセティックな処理を行うアプリケーションである．ベンチマークは，サイズ s の配列データの各要素に対して意味のない演算処理を m 回行う単純なデータ並列カーネルから構成される．今回評価に用いたシンセティックベンチマークは複数個の配列に対してこのような処理を連続的に行う．複数個の配列に処理を行う際に，グラフィクスメモリ上でデータの入れ替えが発生し，このデータ入れ替えにかかる転送遅延を隠蔽することが提案先読み機構の目標となる．配列のデータサイズ s，と要素ごとの演算処理の回数 m は一度のデータ転送量と転送データに対する処理負荷を調節するためのパラメータである．シンセティックアプリケーションは，OpenCL で記述されており，データ転送およびカーネル実行を含むすべての CPU 処理・GPU 処理がシーケンシャルに実行されるよう記述されている．

実験では，上記のシーケンシャルなシンセティックアプリケーション ($normal$) と，これに提案する先読み機構 ($prefetch$) を適用した場合における実行時間を比較した．実行時間は，5 回実行したときの平均値として計測している．表 2 に，今回使用した GPU コンピューティングプラットフォームの詳細を示す．なお，提案システムおよびシンセティックベンチマークのコンパイルには LLVM コンパイラ (version 3.0) を用い-O3 オプションにより最適化を行った．

また，システムメモリとグラフィクスメモリの間の転送最適化手法として，転送元のシステムメモリ領域にページロックメモリを利用する手法が広く利用されている．ページロックメモリを利用することで DMA 転送を利用し，高い転送性能を達成することができる一方，システムメモリ上で利用できるページロックメモリの総量は制限されている (環境に依存するが，数百 MB 程度)．このため，ページロックメモリを利用することが常に正しい解とは限らない．今回の実験では，アプリケーションがページロックメモリを利用している場合，利用していない場合 (通常のメモリを利用する場合)，2 つの場合を別々に実験している．

4.2 実験結果

図 7 から図 10 に，もとのアプリケーション性能に対する提案システム適用時の性能を示す．ページロックメモリを使用する場合，使用しない場合，およびデータ並列カーネルの計算量を調節するパラメータ m が 1 の場合と 7 の場合に対して，合計 $2x2 = 4$ 通りの実験結果を示している．縦軸は，提案システム適用時の実行時間をもとのアプリケーションの実行時間で割って算出した相対性能であり，横軸はシンセティックアプリケーションにおける配列サイズ s の値である．データ当たりの演算量 m の値は，図 9 では $m = 1$，図 10 では $m = 7$ となっている．今回用いたシンセティックアプリケーションで行われているストリーム処理は，提案システムに実装されているストライド予測器によって効果的に予測できるため，ほとんどの点において，先読みの成功率は 100% である．このため，この実験では実アプリケーションにおいて，CPU から GPU への方向のデータ転送予測が理想的に行えた場合に達成できる性能向上率の上限値を大まかに見積もっていると考えられる．なお，ページロックメモリを利用した場合にはシステムが提供できるページロックメモリ容量の制約から，データサイズを 32MB よりも大きくして評価を行うことはできなかった．

データサイズが小さい領域では，提案システムのオーバーヘッドの影響により先読みが成功しているにも関わらず，性能が低下している．一方で，配列サイズが 0.25〜1 MB以上になってくると先読みの効果がオーバーヘッドを越えるようになり，性能向上が確認できる．提案システムでは最大で 31% (ページロック，$m = 7$，16MB の点) の実行時間削減を達成しており，先読みによる性能向上を効果的に実現できている．しかし，さらにデータサイズを大きくした領域 (128MB および 256MB) では先読みによる性能向上効果が再び減少している．これは，今回用いたシンセティックアプリケーションではデータサイズが増加すると計算とデータ転送の比率が変化し，データ転送時間がアプリケーション全体に占める割合が小さくなるためである．

図 11 から図 14 では，CPU から GPU への転送，GPU 上での計算処理，および GPU から CPU への転送の各処理がアプリケーション実行時間全体のうちで占める割合をプロットしたものである．$cpu \rightarrow gpu$ は CPU から GPU

OpenCL Host	Intel Core i7 Processor
OpenCL Device	Nvidia GeForce GTX 570
CL Driver Version	280.13
OpenCL Platform Name	NVIDIA CUDA (OpenCL 1.1 CUDA 4.0.1)
System Memory	8GB (DDR3 PC3-10600 x2)
Graphics Memory	1280MB GDDR5
Host - Device Bus	PCIe 2.0 (x 16)
Operating System	Linux 2.6

表 2 評価に用いたプラットフォーム

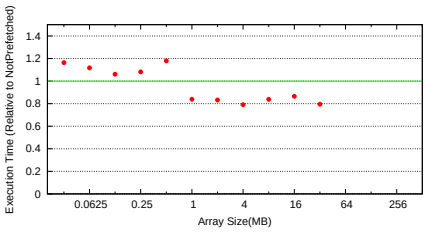

図 7 実行時間 (ページロックメモリ，$m = 1$)

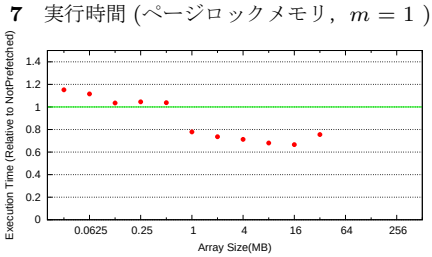

図 8 実行時間 (ページロックメモリ，m=7)

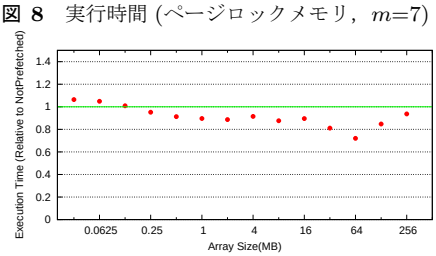

図 9 実行時間 (通常のメモリ，m=1

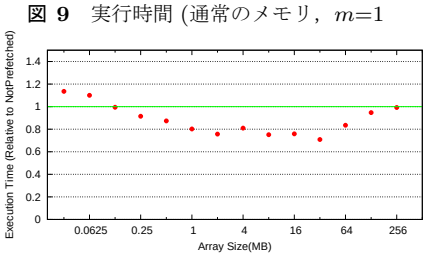

図 10 実行時間 (通常のメモリ ，m=7)

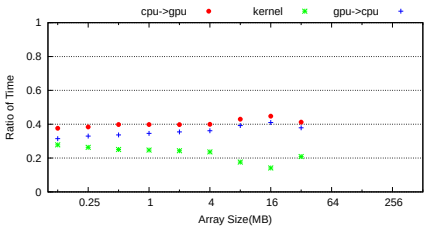

図 11 各処理が実行時間に占める割合 (ページロックメモリ，m=1)

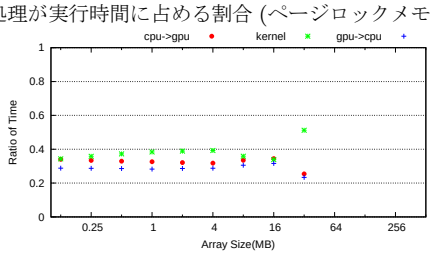

図 12 各処理が実行時間に占める割合 (ページロックメモリ，m=7)

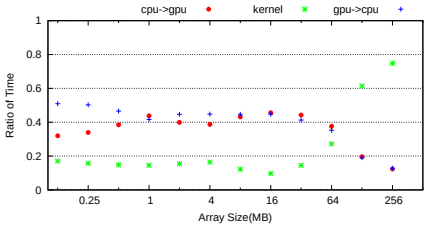

図 13 各処理が実行時間に占める割合 (通常のメモリ，m=1)

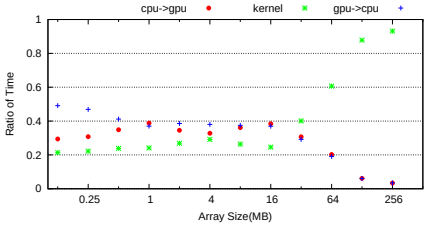

図 14 各処理が実行時間に占める割合 (通常のメモリ，m=7)

へ向かう方向の転送時間の割合，*kernel* はデータ並列処理時間の割合，$gpu \rightarrow cpu$ は GPU から CPU へ向かう方向の転送時間割合，をそれぞれ示している．値は全て先読み機構を適用しない状態で取得したものであるが，各処理単体の実行時間は，先読み機構を適用した場合でも大きく変化しないと考えられる．今回の実験では CPU から GPU への転送でのみ先読みを行っているため，計算と通信が並列実行可能な時間は図中 $cpu \rightarrow gpu$ と *kernel* のうち，より実行時間が短い方の値によって制限される．先読みによるオーバーヘッドが無視できるデータサイズ (0.25MB から 1MB 以上のデータサイズ) では，この計算と通信が並列実行可能な時間が大きい場合には，先読みによる実行時間短縮の効果が大きく (m=1，通常のメモリにおける 64MB 等)，逆に並列実行可能な時間が小さい場合には実行時間はあまり削減されない (m=7，通常のメモリにおける 256MB 等)．ダブルバッファリングに代表される通信遅延の手動最適化において重要になる，計算とデータ転送の間のロードバランスの問題がここでも生じていると考えることができる．

4.3 考察

実験結果は以下の 2 つにまとめることができる．

(1) データ転送サイズが十分大きければ (今回のプラットフォームでは，0.25〜1MB)，現実的なシステムオーバーヘッドで提案システムを既存のプラットフォームの上に構築できる．

(2) 先読みによる性能向上効果は，計算とデータ転送の間のロードバランスに強く依存する．

提案システムによる性能オーバーヘッドとして，3.5 節で説明したように先読みの失敗によるミスペナルティがある．今回の実験では，ストライド予測が効果的に働くストリーム処理を対象として実験を行ったため，ミスペナルティの影響を正確に評価できているとは言えない．しかし，GPU の加速となるデータ並列処理ではデータのアクセスパターンに強い規則性があるため，先読みの成功率は実アプリにおいてもかなり高く，ミスペナルティの影響は大きくなら

ないと予想される．

一方で，データ転送遅延の隠蔽を考える場合に重要な計算とデータ転送の間でのロードバランスの問題は，今回提案した先読み機構だけでは解決できない．ループ分割やループ融合に類似した GPU カーネルの分割実行・融合実行技術を用いて計算と通信の間でロードバランスをとる，といった手法と協調した先読み機構を考えることができるが，これは今後の課題である．

5. 関連研究

GPU プログラミングのプログラマビリティ向上を目的として，システムメモリと GPU メモリの間のデータ転送を自動化するシステムがこれまでにも提案されている．これらは，コンパイラを用いるシステムと動的なメモリ管理機構を用いるものに大別できる．これらのは，2 章において紹介した [1] [3] [4] [5].

本稿ではシステムメモリとグラフィクスメモリの間のデータ転送に焦点を絞っているが，これに対してグラフィクスメモリと GPU のオンチップメモリ間でのデータ転送をターゲットとした先読み手法がこれまでに提案されている．この中で，GPGPU アプリケーションのメモリアクセスパターンの規則性は非常に高く，シンプルな予測アルゴリズムを用いて精度の高いプリフェッチが実現できることが示されている．本研究はこのような知見に基づき，先読みという枠組みによってより下位の階層において性能向上を達成しようとするものである．Lee らは文献 [4] において，GPU 特有のハードウェアプリフェッチ手法として自身とは異なるスレッドのデータをプリフェッチする手法をインタースレッドプリフェッチを提案している．一方，CPU と GPU が同一チップ上に存在し，システムメモリを共有するアーキテクチャにおいてアイドル中の異なるコアを利用してデータをオンチップ上にプリフェッチしようとする研究もなされている．Woo らは文献 [7] において，GPU 上で動作するスレッドを CPU 上で動作するスレッドのためのソフトウェアプリフェッチャとして用いることで CPU 上で動作するスレッドの性能向上を実現している．一方，Yi らは文献 [8] において，これとは逆に CPU 上で動作するスレッドが GPU のオンチップキャッシュへデータをプリフェッチすることで GPU 処理の性能向上を達成できることを示している．Yi らの研究は，よりコアに近いメモリ階層において本稿で提案した先読み機構と同様の機能を実現していると考えられる．

6. まとめと今後の課題

本稿では，GPU コンピューティングにおいてしばしば性能ボトルネックとなるシステムメモリとグラフィクスメモリ間のデータ転送を対象とした先読み機構を提案した．提案システムは，CUDA や OpenCL を用いて直接記述されたアプリケーションや既存の自動データ転送システムからのデータ転送リクエストを，そのセマンティックを守りつつ自動的に非同期通信に置き換えることで性能向上を達成する．既存のプラットフォーム上に構築したプロトタイプシステムを用いて，シンセティックなアプリケーションに対して最大で 31% の性能向上を達成し，提案システムの実現可能性を確認した．

今後の課題として，実アプリケーションにおける評価実験を通じて，ミスペナルティの定量的な評価を行い提案システムの有効性を検討することが挙げられる．また，GPU 上での処理が有するデータ並列性を利用したカーネルの分割実行・融合実行を通じて，計算と通信のロードバランスを自動で最適化する手法と協調し，より大きな性能向上を達成する先読み機構を検討していくことも課題である．

Acknowledgment

本研究は日本学術振興会特別研究員奨励費（２３・８０６２）の助成を受けて行われたものである．

参考文献

[1] Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N. and Hwu, W.-m. W.: An asymmetric distributed shared memory model for heterogeneous parallel systems, *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, New York, NY, USA, ACM, pp. 347–358 (2010).

[2] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture, Fifth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., Waltham, MA, USA (2012).

[3] Jablin, T. B., Prabhu, P., Jablin, J. A., Johnson, N. P., Beard, S. R. and August, D. I.: Automatic CPU-GPU communication management and optimization, *Proc. the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'11)*, New York, NY, USA, pp. 142–151 (2011).

[4] Lee, J., Lakshminarayana, N. B., Kim, H. and Vuduc, R.: Many-Thread Aware Prefetching Mechanisms for GPGPU Applications, *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, Washington, DC, USA, IEEE Computer Society, pp. 213–224 (2010).

[5] Lee, S., Min, S.-J. and Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization, *Proc. the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'09)*, New York, NY, USA, pp. 101–110 (2009).

[6] Nesbit, K. and Smith, J.: Data Cache Prefetching Using a Global History Buffer, *Software, IEE Proceedings-*, p. 96 (2004).

[7] Woo, D. H. and Lee, H.-H. S.: COMPASS: a programmable data prefetcher using idle GPU shaders, *SIGARCH Comput. Archit. News*, Vol. 38, No. 1, pp. 297–310 (2010).

[8] Yang, Y., Xiang, P., Mantor, M. and Zhou, H.: CPU-assisted GPGPU on fused CPU-GPU architectures, *High-Performance Computer Architecture, International Symposium on*, Vol. 0, pp. 1–12 (2012).