情報処理学会論文誌　コンピューティングシステム　Vol.5　No.1　53–67 (Jan. 2012)

OpenCLにおけるタスク並列化支援のための実行時依存関係解析手法

佐藤 功人[1]　小松 一彦[2]　滝沢 寛之[1,3,a)]　小林 広明[2,3]

受付日 2011年5月11日, 採録日 2011年8月23日

概要：本論文では，OpenCL アプリケーションを対象とし，複数のアクセラレータを用いた並列処理に必要となるタスク並列性を見出すための，実行時情報を用いた依存関係解析手法を提案する．提案する解析手法では，メモリへの読み書き順序制約を表すデータ依存関係を解析し可視化を行う．また，API 関数の呼び出し順序制約を表すイベント依存関係を明らかにし，並列処理においてボトルネックになる同期処理を可視化する．提案手法に基づいて 54 種類のベンチマークプログラムを解析することにより，タスク並列性に基づいて並列化できる可能性のあるプログラムを特定することができた．また，潜在的なバグの発見にも，提案手法による解析が有用であることが示された．

キーワード：GPU コンピューティング，性能最適化支援ツール，OpenCL，タスク並列性，依存関係解析

A Runtime Dependency Analysis Method for Task Parallelization of OpenCL Programs

Katsuto Sato[1]　Kazuhiko Komatsu[2]
Hiroyuki Takizawa[1,3,a)]　Hiroaki Kobayashi[2,3]

Received: May 11, 2011, Accepted: August 23, 2011

Abstract: This paper proposes a runtime dependency analysis method to find task parallelism in an OpenCL application for use of multiple accelerators. The proposed method can visualize data dependencies among tasks that represent the constraints on memory access sequences, and event dependencies that show the constraints on API call sequences. As a result, the proposed method can help programmers to find unnecessary synchronization points that often become performance bottlenecks in task-parallel processing. We analyze 54 benchmarks to demonstrate that the proposed method can find programs with task parallelism. Besides, we show that the proposed method is also useful to detect potential bugs.

Keywords: GPU computing, performance tuning tool, OpenCL, task parallelism, dependency analysis

1. はじめに

汎用プロセッサ（CPU）に加えてアクセラレータを搭載している**複合型計算システム**は，様々なアプリケーションを高速化するための有力なシステムアーキテクチャとして注目されている．様々なアクセラレータの中でも，描画処理用プロセッサ（Graphics Processing Unit，GPU）は高い浮動小数点演算能力とメモリバンド幅を持ち，描画処理だけでなく汎用処理においても高い性能を得られる魅力的なアクセラレータである．アクセラレータを単独で用いる

[1] 東北大学大学院情報科学研究科
Graduate School of Information Sciences, Tohoku University, Sendai, Miyagi 980–8579, Japan
[2] 東北大学サイバーサイエンスセンター
Cyberscience Center, Tohoku University, Sendai, Miyagi 980–8578, Japan
[3] 科学技術振興機構戦略的創造研究推進事業
Japan Science and Technology Agency, Core Research for Evolutional Science and Technology, Chiyoda, Tokyo 102–0075, Japan
[a)] tacky@isc.tohoku.ac.jp

だけでも処理速度の高速化に効果的であるが，1つの複合型計算システムの中に複数台のアクセラレータが搭載されることも増えており，複合型計算システムの性能を最大限に引き出すためには複数のアクセラレータを効率的に利用する必要がある．

複合型計算システムの演算性能を活用するためには，アクセラレータにアクセスするための開発環境に含まれるApplication Programming Interface（API）を用いてプログラムを開発する必要がある．複合型計算システム用の開発環境の代表的なものとしてNVIDIA CUDA [1]，AMD APP [2]，OpenCL [3] などが存在する．これらの環境では，アクセラレータで実行される処理が**カーネル**と呼ばれる特殊な関数として定義されている．プログラマは，アクセラレータに様々な**コマンド**[*1]を送信することによって，カーネルの実行やデータ転送などを指示する．複数のアクセラレータを用いるためには，各アクセラレータへ適切にコマンドを送ることによってカーネルの実行を分散して割り当てる必要がある．

1つのアクセラレータを用いるように記述されたプログラムを，タスク並列性に基づいて効率的に複数のアクセラレータで並列動作できるように変更するためには，利用可能なアクセラレータの数に応じたカーネル実行コマンドの再割当てや，不必要な同期ポイントの消去などのチューニング（以下，**タスク並列化**とする）を行う必要がある．複数のアクセラレータの性能を最大限に利用するためのタスク並列化を行うには，プログラムから並列実行可能なカーネルを見つけると同時に，先行カーネルの計算結果を後続カーネルが利用する関係（以下，**データ依存関係**）にも注意を払わなければならない．データ依存関係が存在する場合には，データ依存関係を崩さないように実行の順序を考慮してカーネル実行コマンドを割り当てる必要がある．また，複数のアクセラレータを利用する場合には，データ転送や同期のオーバヘッドが発生するため，タスク並列化を行うことで高速化が期待できるタスクを選別する必要もある．タスク並列化を行うためのソースコードの解析は手間と時間がかかる作業であるため，タスク並列化の障害となっている．

本論文では，OpenCLを用いて開発されたプログラムを対象とし，複数のアクセラレータを用いて並列処理を行うことで性能向上を期待できる，タスク並列性を特定するための実行時依存関係解析手法を提案する．提案手法では，実行時情報から得られるメモリに対するカーネルからの読み書き状態から，カーネル実行コマンドの実行順序を制約するデータ依存関係を解析し，解析結果の可視化を行う．さらに，各API関数の呼び出しを記録・解析し，並列処理においてボトルネックとなる同期処理も可視化する．この依存関係解析結果を利用することで，プログラマのタスク並列化作業を支援することが可能となる．本論文では，得られたカーネル間の依存関係解析結果に基づいてタスク並列化を行い，複数のアクセラレータを利用可能な環境において得られる速度向上を定量的に評価する．

本論文の構成は以下のとおりである．2章では，OpenCLの概要やプログラムの依存関係解析手法の関連研究について述べる．3章では，OpenCLを用いて記述されたプログラムから実行時情報に基づいて依存関係解析を行う手法を提案する．4章では，3章で提案する解析手法を用いて得られた情報に基づいてプログラムを最適化する実例を示すとともに，複数のアクセラレータを用いて負荷分散を図ることで得られる性能向上について定量的に評価する．5章では，本論文のまとめを述べる．

2. 関連研究

2.1 OpenCL

OpenCL [3] は，多様なアクセラレータを統一的な手順で用いるためのプログラミングインタフェースである．OpenCLでは，1つのホストが複数の**デバイス**（Compute Device）と呼ばれるアクセラレータを管理するハードウェア構成を想定している．

OpenCLでは，デバイス上で実行するプログラムをカーネルとして定義し，カーネルを独立して実行するスレッドを多数生成して同時に処理を行うプログラミングモデルを持つ．ホストから各デバイスを制御するためには，**図1**に示すようにデバイスに対応づけられた**コマンドキュー**（Command Queue）と呼ばれる待ち行列にコマンドを投入する．コマンドキューにカーネル実行コマンドを投入することで，カーネルはホスト側とは非同期にデバイスで実行される．コマンドキューは互いに独立してコマンドを実行するため，複数のデバイスに対して並列実行可能なコマンドを割り当てる場合には，各デバイスに対応づけられたコマンドキューにカーネル実行コマンドを投入する．

コマンドキューにコマンドを投入するとき，他のコマンドに関連づけられた**イベントオブジェクト**を用いることで，コマンド間の順序制約を指定することが可能である．たとえば，カーネルK1の実行コマンドをコマンドキューに投入するときに，カーネルK1の実行というイベントに対応するイベントオブジェクトE1を取得する．次にカーネルK1に依存関係があるカーネルK2を実行するコマンドをコマンドキューに投入するときに，E1に依存していることを指定することで，2つのコマンド間の順序関係を明示的に指示できる．その結果，カーネルK1の実行完了後にカーネルK2を実行するという順序制約を与えることができる．

[*1] 本論文では，アクセラレータで処理を実行するために待ち行列に投入されるものをコマンドと定義する．カーネルが複数回実行される場合には，実行回数に応じてコマンドが生成される．また，データ転送API関数などでアクセラレータに対してデータ転送要求などを待ち行列に投入した場合もコマンドが生成される．

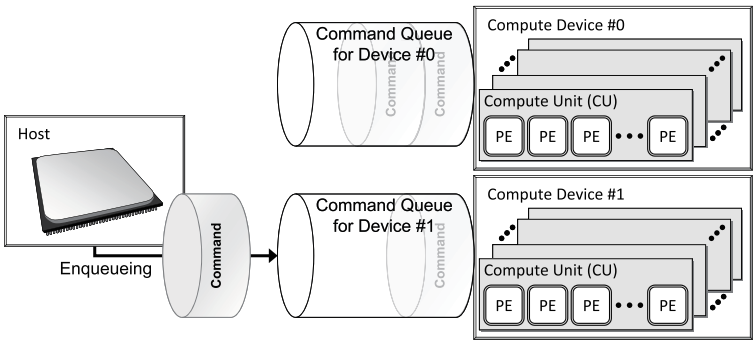


図 1　OpenCL のキューイングモデル
Fig. 1　The queueing model of OpenCL.

一方で，コマンドキューにコマンドを投入するための API 関数には，呼ばれたときにホストと非同期でコマンドを実行する**ノンブロッキング**関数と，ホストと同期してコマンドを実行し，コマンドが完了するまで API 関数からホストへ制御が戻らない**ブロッキング**関数が存在する．ブロッキング関数が呼ばれた場合，ホストは後続のコマンドを投入できないため，暗黙的な順序制約が発生する．

以上のように，コマンド間で実行の順序に制約がある場合，イベントオブジェクトを用いて明示的にその順序制約を指示するか，ブロッキング関数を用いてホストとデバイスを同期させて暗黙的に順序制約を指示する．後者の場合には，ブロッキング関数によって投入されたコマンド（**ブロッキングコマンド**）が完了するまで，後続のコマンドを投入することができない．したがって，ブロッキングコマンドと後続のすべてのコマンドとの間に順序制約が存在するといえる．一方，前者の場合には，本当に必要な順序制約のみを明示的に指示することが可能であり，この部分を特定できれば順序制約のない他の複数のコマンドを並列実行できる可能性が残されている．このため本論文では，並列実行可能なコマンドの発見を支援するために，コマンド間の順序制約を解析する．

また，カーネルからアクセスされるデータはすべてデバイス側のメモリに確保され，**メモリオブジェクト**として管理されている．メモリオブジェクトに対しては，データ転送 API 関数，およびカーネル内部からのみアクセス可能である．メモリオブジェクトの生成時に，カーネル内部からのアクセスモードとして読み込み専用（Read Only），書き込み専用（Write Only），読み書き可能（Read Write）のいずれかが指定される．そのため，カーネル内部での読み込みや書き込みの有無（以下，**読み書き関係**とする）は，生成時に指定されたアクセスモードを参照することで，ある程度推測することが可能である．しかし，複数のカーネルから参照されるメモリオブジェクトでは，生成時に “読み書き可能” が設定されていても，カーネルによっては読み込みのみや書き込みのみを行う場合もある．タスク並列性を抽出するためには，正確な読み書き関係を解析し，カーネル間のデータ依存関係を詳細に把握する必要がある．このため本論文では，各メモリオブジェクトに対するカーネルの読み書き関係も解析する．

2.2　プログラムの依存関係解析

依存関係解析は，プログラムの変換や並列化を行ううえで基礎的な情報を得る手段であり，様々な手法がこれまでに提案されている．効率的な機械語コードを生成するための基本的なコンパイラ技術である命令間のデータ依存解析 [4] は，並列実行可能な命令組を抽出するための重要な手法である．また，C 言語や FORTRAN 言語のような高級言語では，並列実行可能なブロックを検出することがプログラムの最適化にきわめて重要である．このため，高級言語における依存関係解析では，並列実行可能なコードブロックの検出と，それに基づく依存関係解析がこれまでに数多く研究されてきた．

Kasahara らは，FORTRAN 言語で記述されたプログラムから，並列処理可能な基本ブロック，繰返しブロック，サブルーチンブロックを抽出し，OpenMP のディレクティブを自動的に挿入する手法を提案している [5]．この手法では，プログラムを解析することで基本ブロック間の制御依存関係とデータ依存関係を表すマクロフローグラフを生成し，再早実行可能条件解析 [6] を行うことで並列実行可能な基本ブロックの集合を示すマクロタスクグラフを生成する手法を述べている．マクロタスクグラフを参照することで条件分岐に依存して変化する並列実行可能な基本ブロックの組合せを明らかにすることができ，OpenMP のディレクティブを付加した並列実行可能なコードを出力することが可能となる．

Diamos らは，複合型計算システムにおいて条件分岐を越えてカーネルを投機実行をすることで，カーネル間並列性を利用する手法を提案している [7]．この手法では投機実行を行うために，独自の Harmony フレームワーク [8] を用いて記述されたプログラムに対し，コンパイラによってプログラム全体の制御フロー解析とデータ依存関係解析を静的に行う．また，Diamos らはいくつかの基本的なプロ

グラムにおいて完全に制御依存関係が解消できた場合に得られるカーネルレベル並列性を示すとともに，投機実行した場合に得られる性能向上について報告している．

この手法では，プログラマが独自フレームワークで記述されたカーネルに入出力変数を明示する注釈（annotation）をつけることを前提として，コンパイラによる静的なデータ依存関係解析を行う．そのため，カーネルにおけるメモリオブジェクトの読み書き関係を正確に把握するためには，プログラマがプログラムを改変して正確に注釈をつける必要がある．また，既存のプログラムの中でアクセラレータで実行する部分だけを切り出して CUDA や OpenCL のプログラムに変換している場合では，ソースコード全体をフレームワークから参照することができないため，Harmony の依存関係解析手法をそのまま適用することは難しい．一方で，本論文で提案する手法は，カーネルプログラムの解析を実行時に行う．また，実行時にカーネルに渡されたメモリオブジェクトのポインタをもとに解析しているため，プログラムに追加の注釈をつける必要がなく，依存関係が動的に変化する場合でも把握可能である．さらに，実行時に得られる情報のみに基づいて解析を行うために，OpenCL をバックエンドとして用いるアノテーションベースのフレームワーク [9], [10] と同時に用いることも可能である．

投機実行によってカーネル間の並列性を引き出す手法は，投機失敗のオーバヘッドを考慮すると，すでに並列化されているプログラムに比べて実効性能が低下する恐れがある．そのため，並列性を持つ部分をプログラマが明示的に並列化しておくアプローチのほうが高い実効性能を期待できる．本論文では，カーネル間の並列性を引き出すことが可能なプログラムの作成を支援することを目的として，プログラマがプログラムを並列化するときに有用な情報を抽出する手法を提案する．

3. 実行時依存関係解析と依存関係グラフを利用したタスク並列化

本章では，OpenCL を用いて記述されたプログラムのタスク並列化を行うために，メモリに対するカーネルからの読み書き関係，コマンド間のデータ依存関係とコマンド間のイベント依存関係を実行時の情報から解析する手法を提案する．これらの依存関係グラフを生成することで，プログラマによるタスク並列化の支援に有用な以下の効果を期待できる．

- 並列実行できる可能性があるコマンド群の検出
- 不必要な順序制約や同期の検出
- メモリオブジェクト生成時の誤ったアクセスモード指定の検出

これらの効果によって，OpenCL プログラムのタスク並列化が容易になるとともに，高い並列化効率が期待できる．また，プログラム中の誤りを容易に検出することができる．

3.1 メモリアクセス関係の解析とデータ依存関係グラフの生成手法

同じメモリオブジェクトにアクセスする先行コマンドと後続コマンドの間には，**表 1** に示す 4 種類の依存関係が発生する．本論文では，データ依存関係グラフを，コマンドをノード（結節点），データ依存関係をノード間を結ぶ有向エッジ（矢印）として**図 2** のように表現する[*2]．図 2 では，依存関係の種類の違いがエッジの色の種類とラベル文字列（RaW，WaW，WaR，RaR）によって表現されている．

タスク並列化を行うためには，計算結果を保証するために**真のデータ依存関係**を維持した上で，名前依存による偽の依存関係をできる限り解消し，互いに依存関係がない並列実行可能なコマンドをより多く見つけ出すことが求められる．2 つのコマンドが Read after Write（RaW）の関係にある場合，それらは真のデータ依存関係であるため必ず維持しなければならない．2 つのコマンドが Write after Write（WaW）の関係や Write after Read（WaR）の関係にある場合，それらは偽の依存関係であり原則としてメモリオブジェクトを複製して別名を与えることで解消できる．ただし，WaW の依存関係では，先行コマンドと後続コマンドがそれぞれメモリオブジェクトの一部しか書き換えない場合，メモリオブジェクトは両者の書き込み結果を反映していなければならず，複製して別名を与えるだけでは解消できない場合がある．

本節では，まず，メモリオブジェクトに対してアクセスを行うコマンドを時系列順に記録し，メモリオブジェクトに対するアクセス状態を解析する手法を説明する．次に，解析されたメモリオブジェクトに対するアクセス状態をもとに，コマンド間のデータ依存関係を解析する手法について説明する．

OpenCL では，メモリ領域がメモリオブジェクト単位で管理されている．このため，カーネルや API 関数へ渡される引数から，各コマンドがアクセスするメモリオブジェクトを知ることができる．提案手法では，各コマンドのメモリオブジェクトに対する読み込みや書き込みの有無をすべて記録する．すべてのコマンドの実行完了後に，メモリオブジェクトに対するアクセス履歴を時系列順にたどることにより，メモリオブジェクトとコマンドの間の読み書き関係を表す**メモリアクセスグラフ**を得ることができる．

カーネルの呼び出しでは，引数として与えられたメモリオブジェクトに対する読み書き関係を明らかにするために，カーネルコードのビルド時（`clBuildProgram` 関数呼び出し時）にカーネルプログラムの解析を行う．メモリオブジェクト以外の引数は読み込み専用として扱い，メモリオブジェクトのポインタ引数については，カーネルプログ

[*2] 本論文ではグラフを可視化するために Graphviz [11] を使用している．

表 1　コマンド間に発生するデータ依存関係の種類
Table 1　Types of inter-command data dependencies.

依存関係	先行コマンド	後続コマンド	備考
Read after Write (RaW)	書き込み	読み込み	真のデータ依存関係
Write after Write (WaW)	書き込み	書き込み	偽の依存関係
Write after Read (WaR)	読み込み	書き込み	偽の依存関係
Read after Read (RaR)	読み込み	読み込み	無視可能

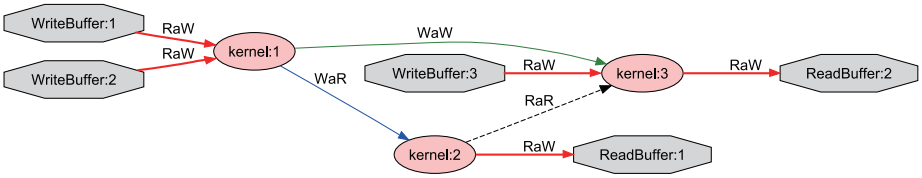


図 2　コマンド間のデータ依存関係の例
Fig. 2　Examples of inter-command data dependencies.

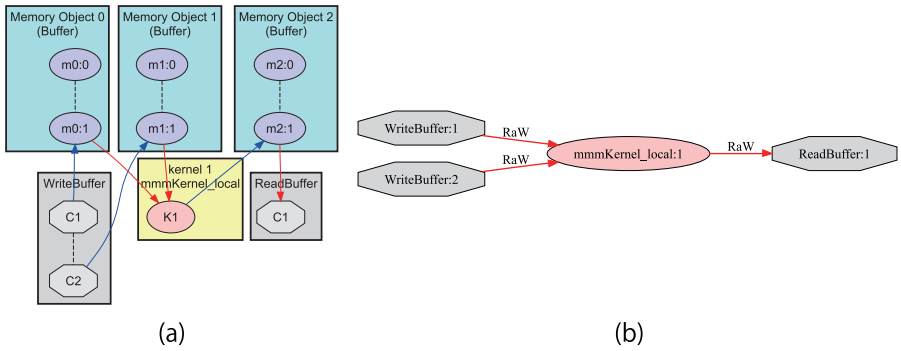


図 3　行列積プログラムにおける (a) メモリアクセスグラフと (b) データ依存関係グラフの例
Fig. 3　Examples of (a) an access relation graph and (b) a task dependency graph for matrix multiplication.

ラムの式を解析し，読み書き関係を把握する．メモリオブジェクトに対する読み書き関係の解析では，次の解析を式ごとに行う．

- 左辺にポインタ型変数を含む式の場合，右辺に含まれるポインタ型変数を関連ポインタとして左辺のポインタ型変数に関連づける．代入式の場合は以前の関連ポインタを破棄したうえで新しい関連ポインタを設定し，演算代入式の場合は新しい関連ポインタを追加する．
- ポインタ型変数へ参照演算子または配列インデックス式を用いるアクセスを検出した場合には，書き込みであればそのポインタ型変数に Write フラグを，読み込みであれば Read フラグを設定する．このとき，参照しているポインタ型に関連ポインタが設定されている場合には，その関連ポインタ変数に対しても同様にフラグを設定する．
- カーネル関数内で他の関数を呼び出している場合，呼び出し先の関数についても再帰的に解析を行い，子関数の引数読み書き関係を親関数の引数読み書き関係に統合する．

カーネル呼び出し時には，解析したカーネルの引数の読み書き状態と，引数として渡されたメモリオブジェクトのポインタの両方を参照することで，カーネル実行コマンドとメモリオブジェクトの間の読み書き関係を確定する．

本解析手法は，カーネルプログラムが解析可能なテキスト形式で与えられていることを仮定している．OpenCL ではカーネルをバイナリ形式で与えることも可能であるが，バイナリ形式で与えられた場合には本解析手法を用いることはできない．また，左辺にポインタ型変数をとる演算式の中で非ポインタ型からポインタ型への型変換が行われている場合には，関連するポインタの正確な追跡が困難となるため，そのような記述を検出した時点でカーネルの読み書き関係解析を中止する．このような読み書き関係解析に失敗した場合は，メモリオブジェクト生成時に設定されたアクセスモードフラグを用いて，コマンドからの読み書き関係を推定する．たとえば，Read/Write のアクセスモードが設定されているメモリオブジェクトに対しては，読み書き関係が解析不能のカーネル関数からのアクセスは Read/Write であると推測する．カーネルにおける引数の読み書き関係の解析が失敗した場合であっても依存関係解析は保守的に行われるため，偽の依存関係を生み出すことにはなるが，真の依存関係を見落とすことはない．

図 3 (a) は，提案するメモリアクセス解析手法に基づい

て生成されるメモリアクセスグラフである．プログラムには ATI Stream SDK 2.3 [12] に含まれている行列積プログラム（MatrixMultiplication）を用いている．メモリアクセスグラフでは，ノードがコマンドとメモリオブジェクトを表しており，メモリオブジェクトに対する書き込みが青色エッジで，読み込みが赤色エッジで示されている．また，コマンドノードは八角形で，メモリオブジェクトノードとカーネル実行コマンドノードは楕円形でそれぞれ示されている．図中に示すメモリオブジェクトのコロンの後ろは世代番号を示し，メモリオブジェクトが更新されるたびに世代番号が加算される．図 3 (a) の例では，WriteBuffer コマンドでメモリオブジェクト 0 とメモリオブジェクト 1 に対して書き込み操作を行い，mmmKernel_local というカーネルがメモリオブジェクト 0 とメモリオブジェクト 1 からデータを読み取り，計算結果をメモリオブジェクト 2 に書き込む．メモリオブジェクト 2 に書き込まれた結果は，ReadBuffer コマンドによって読み出されている．

次に，図 3 (a) を用いて得られたデータ依存関係グラフの作成手順を説明する．図 3 (a) において，メモリオブジェクト 0 に着目して世代番号順に探索すると，世代番号 1 では WriteBuffer コマンドが書き込みを行った後に mmmKernel_local というカーネル実行コマンドが読み込みを行う RaW 依存関係が存在している．これより，WriteBuffer コマンドと mmmKernel_local 実行コマンドの間には RaW 依存関係があると判断できるため，データ依存関係グラフの 1 つのエッジとして出力する．これをメモリオブジェクトごとに世代番号順に解析を繰り返し，最後にコマンド間で重複する依存関係を削除することでデータ依存関係グラフが完成する．図 3 (b) は MatrixMultiplication に対するデータ依存関係グラフの生成例である．このように，メモリオブジェクトを中心として読み書きの関係があるコマンドを解析することにより，データ依存関係グラフを生成することができる．

提案手法では，メモリアクセスグラフの各メモリオブジェクトに着目し，コマンドからの読み書きを時系列順にエッジとして出力することでデータ依存関係グラフが生成される．ただし，あるコマンドの実行によってメモリオブジェクトに書き込みが発生したとしても，メモリオブジェクト全体が書き換えられるとは限らないため，先行するコマンドの実行による書き込みの影響を無視することはできない．そのため，後続コマンドは同じメモリオブジェクトへの書き込みを行うすべての先行コマンドに対して偽の依存関係（WaW）を持つ可能性がある．しかし，すべての先行コマンドに対して偽の依存関係エッジを出力すると，エッジの数が多すぎてイベント依存関係が分かりにくくなる．そのため，本提案手法では次の原則に基づいて出力するエッジを削減する．

(1) **コマンド間に偽の依存関係と真のデータ依存関係が存在する場合，真のデータ依存関係のみを出力する．**維持しなければならない真のデータ依存関係を優先し，同じコマンド間に発生している偽の依存関係を無視する．

(2) **ある後続コマンドから真のデータ依存関係をたどって到達可能な先行コマンドから，その後続コマンドへ発生している偽の依存関係は出力しない．**複数のコマンドにまたがって真のデータ依存関係が連鎖している場合に，連鎖しているコマンド間に偽の依存関係があったとしても解消不能であるため無視する．

以上の依存関係削減手法を用いたうえで，本手法ではデータ依存関係グラフ構築時に出力する依存関係を選択できるようにしている．解消が不可能な真の依存関係のみが示されたデータ依存関係グラフを出力して参照することで，プログラマはそのプログラムの本質的な並列性を視認することができ，並列化をするために注力しなければならないカーネルを把握することができる．また，並列化の対象となるカーネルコマンドに直接依存関係がある部分的な依存関係グラフを出力することで，視認性の低下を抑えつつ，偽の依存関係を含むデータ依存関係グラフから最適化に必要な情報を得ることが可能である．ただし，以上の手法を用いてもプログラム規模の増大にともなって依存関係グラフの複雑性は増大する．しかし，多くのプログラムに含まれるループ文による繰返し処理については，2 回目以降に繰り返して実行されるコマンドについての依存関係グラフを省略して示すとともに，ループの内と外を分割し，階層化してグラフを示すことで視認性の低下を防ぐことが可能であると考えられる．このような視認性を高める手法の検討は，今後の課題である．

データ依存関係グラフは，カーネル間で維持すべき真のデータ依存関係を明らかにするとともに，タスク並列化のために解消しなければならない偽の依存関係の存在を提示することができる．これにより，タスク並列化を行う場合には，どの部分に着目すべきかが容易に判断できるため，並列化作業の効率化を図ることができる．

3.2　API 関数呼び出し履歴とイベントオブジェクトの解析によるイベント依存関係グラフの生成

本節では，呼び出された API 関数と順序制約を指定するイベントオブジェクトを実行時に記録し，それらを解析することによってコマンド間のイベント依存関係グラフを生成する手法を提案する．

イベント依存関係は，コマンドをコマンドキューに投入する API 関数呼び出し間の依存関係であり，イベントオブジェクトを用いてプログラマから明示的に指定された**明示的な依存関係**と，ホストと同期することで同期以後のコマンドとの間に発生する**暗黙的な依存関係**が存在する．イベント依存関係グラフを生成するためには，コマンドを投入

する API 関数の呼び出し履歴とプログラマによって指定されたイベントオブジェクトの履歴を利用する．API 関数の呼び出し履歴には，投入したコマンドの種類と，そのコマンドがブロッキングコマンドであるかを記録する．イベント依存関係グラフ生成時には，時系列順に API 関数の呼び出し履歴をたどって各コマンドに対応するノードを出力し，ノードに関連するエッジを次の規則に従って出力する．

- ノードがブロッキングコマンドである場合，直前のブロッキングコマンドとの間に存在するすべてのノンブロッキングコマンドに対して明示的な依存関係を表すエッジを出力する．また，直前のブロッキングコマンドに対しては，暗黙的な依存関係を表すエッジを出力する．
- ノードがノンブロッキングコマンドである場合には，直前のブロッキングコマンドに対して暗黙的な依存関係を表すエッジを出力する．
- コマンドに対して明示的に依存関係が指定されている場合には，前述のエッジとは別に明示的な依存関係の指定に基づくエッジを出力する．同じコマンド間に明示的な依存関係と暗黙的な依存関係が両方存在する場合には，明示的な依存関係を表すエッジのみを出力する．
- 先行コマンドの完了を待つだけの API 関数（clWaitForEvent や clFinish）を，ノンブロッキング関数に変更することはできない．そのため，必ずブロッキングが発生する場所のつながりを示すために，直前の完了を待つだけのコマンドとの間に暗黙的な依存関係を出力する．

イベント依存関係グラフを**図 4** に示す．図 4 は `MatrixMultiplication` におけるイベント依存関係グラフの例であり，図 4 (a) は投入されたコマンドキューごとにイベントを分類した場合，図 4 (b) はコマンドキューに関係なく出力した結果を示している．図 4 では，コマンドをノード，コマンド間のイベント依存関係をエッジとして，明示的な依存関係は黒いエッジ，暗黙的な依存関係は青いエッジによって示されている．また，ブロッキングコマンドはノードの枠を赤で，ノンブロッキングコマンドはノードの枠を黒で示す．コマンドがカーネルや API 関数の何回目の呼び出しに対応しているかを，ノード名のコロンの後に示す．

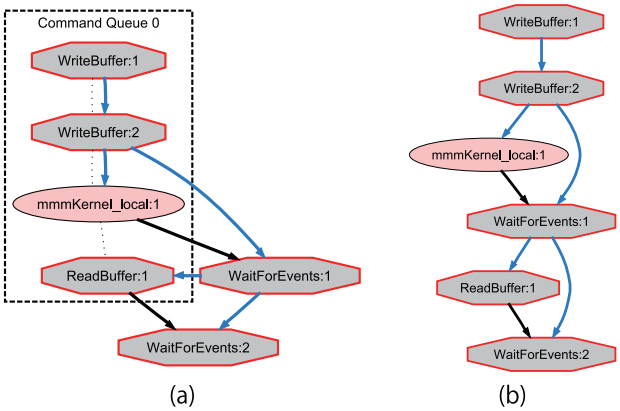


図 4　行列積プログラムにおけるイベント依存関係グラフ．(a) コマンドキューを考慮した場合，(b) コマンドキューを考慮しない場合

Fig. 4　Examples of event dependency graphs for matrix multiplication. (a) with considering a command queue. (b) without considering a command queue.

明示的な依存関係では，イベント依存関係が設定されたコマンドの間のみ順序制約が課される．一方で，暗黙的な依存関係では同期コマンドを越えて後続コマンドを先に実行することはできない．そのため，タスク並列化可能なコマンドを増やすためには，暗黙的な依存関係をできる限り明示的な依存関係に置き換える必要がある．本提案手法では，明示的な依存関係と暗黙的な依存関係を提示することで，どのコマンドに不必要な同期が存在するのかを明らかとし，タスク並列化を支援することができる．

4. 依存関係グラフに基づくタスク並列化支援の評価

本章では，提案手法を用いていくつかのプログラムを解析し，その解析結果に基づいてプログラムを最適化する．本評価には，ATI Stream SDK v2.3 および CUDA SDK 3.2 に含まれているベンチマークの中から 54 種類のベンチマークを用いる．評価環境には，**表 2** に示すように，描画用として NVIDIA GeForce GTX 480，演算用として NVIDIA Tesla C2070 という 2 種類の GPU が搭載されており，どちらもアクセラレータとして利用可能である．54 種類のベンチマークのうち，並列に実行可能なカーネルを持つベンチマークは 15 種類存在する．本評価では，依存関係解析の結果から本提案手法を議論するうえで特徴的な以下のベンチマークを用いて，提案手法の有用性を議論する．

- ATI Stream SDK: `MatrixMultiplication`
- ATI Stream SDK: `MontecarloAsian`
- ATI Stream SDK: `RadixSort`

上記のベンチマークを 1 台のデバイスで実行する場合，Tesla C2070 よりも GeForce GTX 480 を用いた方が高い性能を得られるため，タスク並列化されていないベンチマークを実行する際には，GeForce GTX 480 を用いる．

表 2　実験環境概要
Table 2　Experimental setup.

CPU	Intel Core i7 920 2.66 GHz
GPU	NVIDIA GeForce GTX 480 + Tesla C2070
OS	CentOS 5.5 (Linux 2.6.18)
Compiler	GCC 4.1.2 with -03
Video Driver	260.19.26

4.1　依存関係グラフに基づくタスク並列化

4.1.1　MatrixMultiplication

MatrixMultiplication のタスク並列化過程を通して，提案手法の有用性を議論する．提案手法により解析された MatrixMultiplication におけるデータ依存関係グラフは図 3 (b) に，イベント依存関係グラフは図 4 に示されている．データ依存関係グラフから，真のデータ依存関係によってコマンド群が直列につながっており，MatrixMultiplication には並列実行可能なカーネルが含まれていないことが分かる．一方で，データ依存関係グラフとイベント依存関係グラフをあわせて見ると，2 つの WriteBuffer コマンド間には真のデータ依存関係がないにもかかわらず，暗黙的なイベント依存関係によって順序制約が設定されていることが分かる．これは，WriteBuffer コマンドがブロッキング関数呼び出しを用いてコマンドキューに投入されているためである．また，ReadBuffer がブロッキングコマンドとなっているにもかかわらず，さらに WaitForEvents によってホストとデバイスが同期され，結果として不要な順序制約や同期が暗黙的に設定されている．

このような順序制約や同期は性能低下の潜在的な原因になりうるため，これらを削除することで性能向上が期待できる．プログラマは，不要な順序制約がプログラム中に存在することをグラフから読み取り，並列処理においてボトルネックとなっている API 呼び出しを重点的に最適化することができる．この例の場合，WriteBuffer:1 と WriteBuffer:2 をノンブロッキング関数呼び出しを用いてコマンドキューに投入し，データ依存関係グラフに従って WriteBuffer:1, WriteBuffer:2 および ReadBuffer:1 と mmKernel_local:1 との間の順序関係をイベントオブジェクトを用いて明示的に設定することができる．また，WaitForEvents を削除してもコマンド実行の順序が変化しないことがイベント依存関係グラフから読み取れるため，それらのコマンドを削除することで不要な同期を削減できる．以上の変更を加えると，イベント依存関係は**図 5** (a) から図 5 (b) のように整理され，不要な同期を削減できる．

このように，提案手法によって生成された依存関係グラフを用いることで，プログラマは不必要な順序制約や同期を容易に見出すことができ，並列処理に向けた最適化で注力すべき点を迅速に知ることができる．

4.1.2　MontecarloAsian

MontecarloAsian は，様々に条件を変えながらシミュレーションを実行し，それらの結果を積分する処理を行うプログラムである．

提案手法により解析された MontecarloAsian のデータ依存関係グラフを**図 6**，**図 7** に示す．図 6 のデータ依存関係グラフより，MontecarloAsian には多数の偽の依存関係が存在していることが分かる．真のデータ依存関係のみを示した図 7 より，偽の依存関係をすべて解消できた場合，10 個の並列実行可能なタスクがあることが分かる．

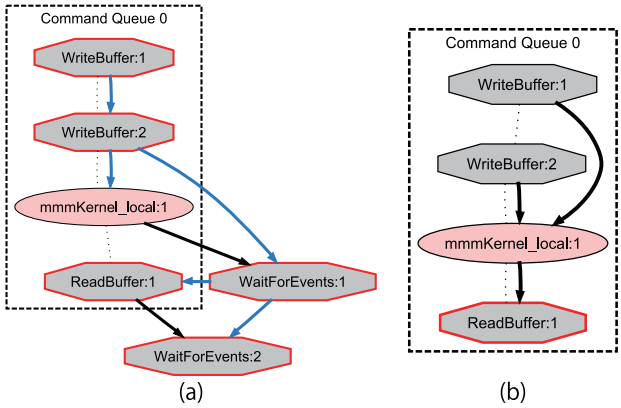


図 5　行列積プログラムにおける最適化前後のイベント依存関係グラフ．(a) 最適化前，(b) 最適化後

Fig. 5　A comparison result of event dependency graphs for matrix multiplication. (a) unoptimized. (b) optimized.

タスク間に偽の依存関係を生じさせる原因となるメモリオブジェクトは，提案手法により解析された**図 8** に示すメモリアクセスグラフを参照することで特定可能である．図 8 を参照することで，メモリオブジェクト 1 およびメモリオブジェクト 2 が複数回用いられていることが偽の依存関係（WaW）の原因となっていることが把握できる．よって，偽の依存関係を解消するためにはこれらのメモリオブジェクトの複製を行う必要があることが分かる．

提案手法によって解析されたイベント依存関係グラフを**図 9** に示す．図 9 (a) と図 7 から，並列に実行可能なカーネル実行コマンドが同期コマンドの間に挟まれ，暗黙的な順序制約によって逐次実行されている様子が分かる．さらに，カーネル実行のたびに ReadBuffer コマンドによるデータ転送とホスト側での演算が行われるために，タスク並列性が十分に活用できていないことも分かる．

プログラマは，解析結果グラフに示された再利用されているメモリオブジェクトについて，複製して別名をつけることで偽の依存関係を容易に解消することができる．後続コマンドによる書き込みがメモリオブジェクト中のすべての値を更新するかを確認し，偽の依存関係（WaW）を生じさせているこれらのメモリオブジェクトを複製することで，カーネル実行コマンドの並列実行を阻害する偽の依存関係を取り除くことができる．

さらに，データ依存関係グラフに示されている真の依存関係に基づいてコマンド間の明示的な依存関係をイベントオブジェクトを用いて指定し，ブロッキングコマンドをノンブロッキングコマンドに変更して不要な依存関係を解消することができる．同時に，演算の中心になっているループを変形してコマンドの投入順序についても調整する．最適化前と最適化後のループ部分の擬似コードを**図 10** に示す．図 10 上段に示している最適化前プログラムでは解消できないブロッキングイベントとノンブロッキングイベン

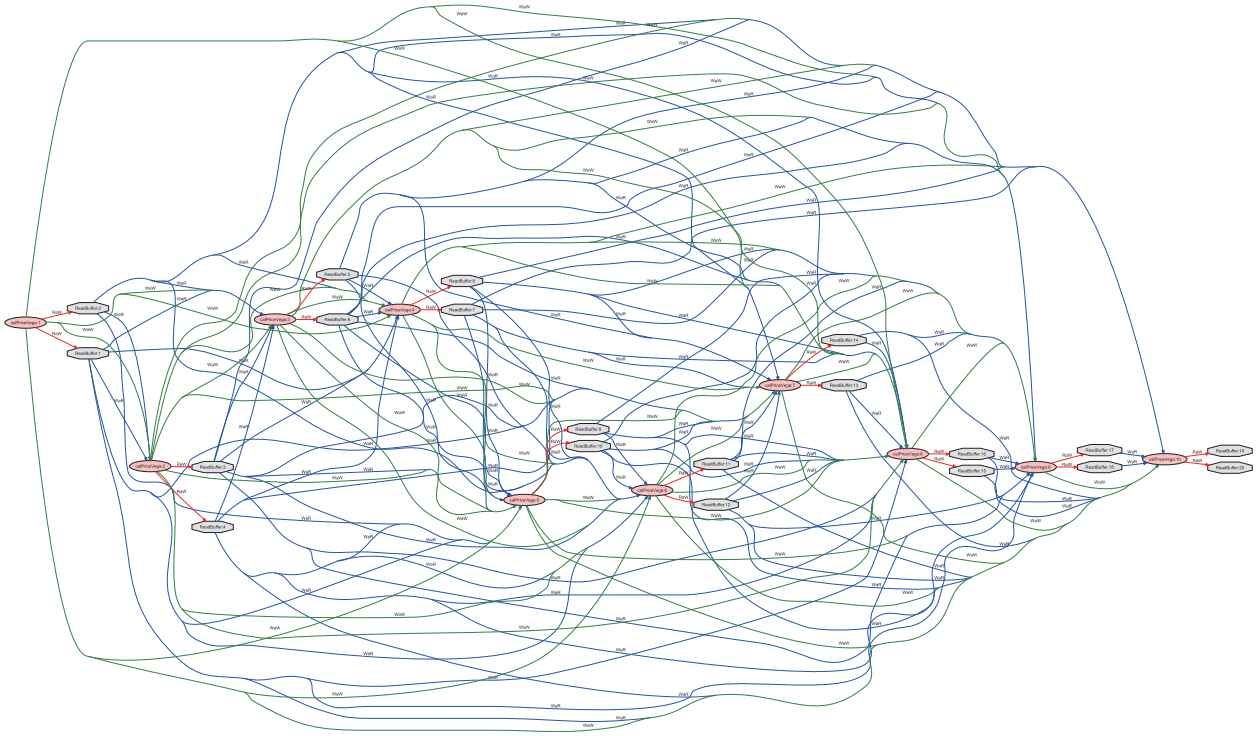


図 6　MontecarloAsian におけるデータ依存関係グラフ（偽の依存関係を含む全依存関係）
Fig. 6　The task dependency graph of the MontecarloAsian (All dependencies including RaW, WaW, WaR).

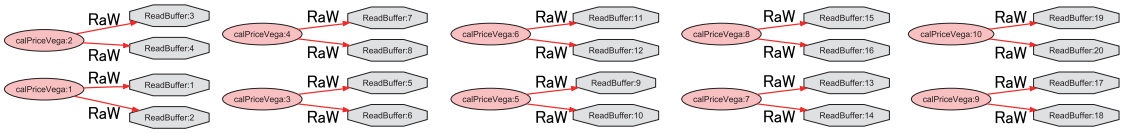


図 7　MontecarloAsian におけるデータ依存関係グラフ（真のデータ依存関係のみ）
Fig. 7　The task dependency graph of the MontecarloAsian (Only true dependencies).

トが交互に呼び出されていたが，図 10 下段に示すように互いに依存関係を持たない独立なノンブロッキングイベントをコマンドキューに投入するループを別に作ることで，これらのイベントの並列処理が可能となる．以上の手順で最適化した結果のイベント依存関係グラフは図 9 (b) のようになる．最適化されたイベント依存関係では，10 個の calPriceVega カーネル実行コマンドがどこにも依存していない状態となり，並列実行可能なタスクとして実行できるようになる．

4.1.3　RadixSort

RadixSort は，histgram と permute という 2 種類のカーネルをループで繰り返し実行する．**図 11** に示すメモリアクセスグラフから，RadixSort では，**表 3** に示すように生成時には書き込み専用に設定されていたメモリオブジェクト 2 が，permute カーネル内でリードアクセスされていることが分かる．これは，本評価環境下では正常動作しているが，OpenCL の実装によっては不具合を引き起こす恐れのある潜在的バグである．このように，提案手法による依存関係解析はメモリオブジェクトに対するアクセスモード設定の間違いというバグの検出にも有用である．

ただし，この事例では提案手法によって提示できる情報の限界も示している．**図 12**，**図 13** に示す RadixSort の真のデータ依存関係グラフからは，それぞれのカーネル実行はすべて独立に行うことができるように読み取ることができる．

データ依存関係グラフに基づき，メモリオブジェクトの複製により図 12 に存在している偽の依存関係を解消することで，カーネルを並列に実行するように最適化できるように見える．しかし，複製対象となるメモリオブジェクト周辺のホスト側のコードを解析すると，RadixSort ではカーネルの実行結果をホスト側に一度転送し，他のメモリオブジェクトへ書き込んでいるため，OpenCL のメモリオブジェクトを介さない真のデータ依存関係が存在している (図 13 (b))．このように，本提案手法で並列化ができる可能性を持つカーネルを示した場合でも，ホスト側の処理においてデータ依存関係が発生している場合には，並列化できない場合が存在する．しかし，このような事例でも本提案手法による解析結果を利用することにより，プログラマ

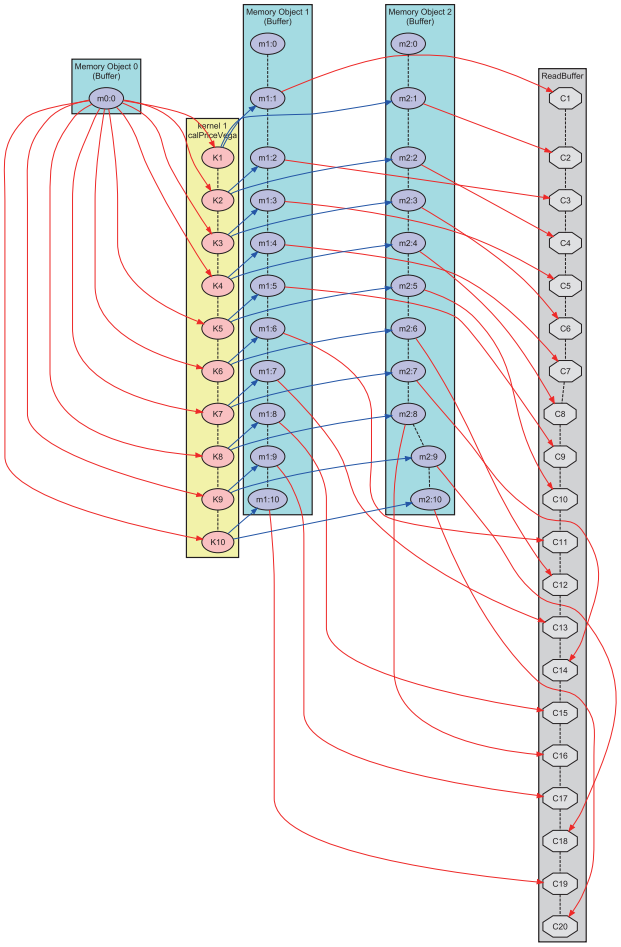


図 8　MontecarloAsian におけるメモリアクセスグラフ
Fig. 8　The memory access graph of MontecarloAsian.

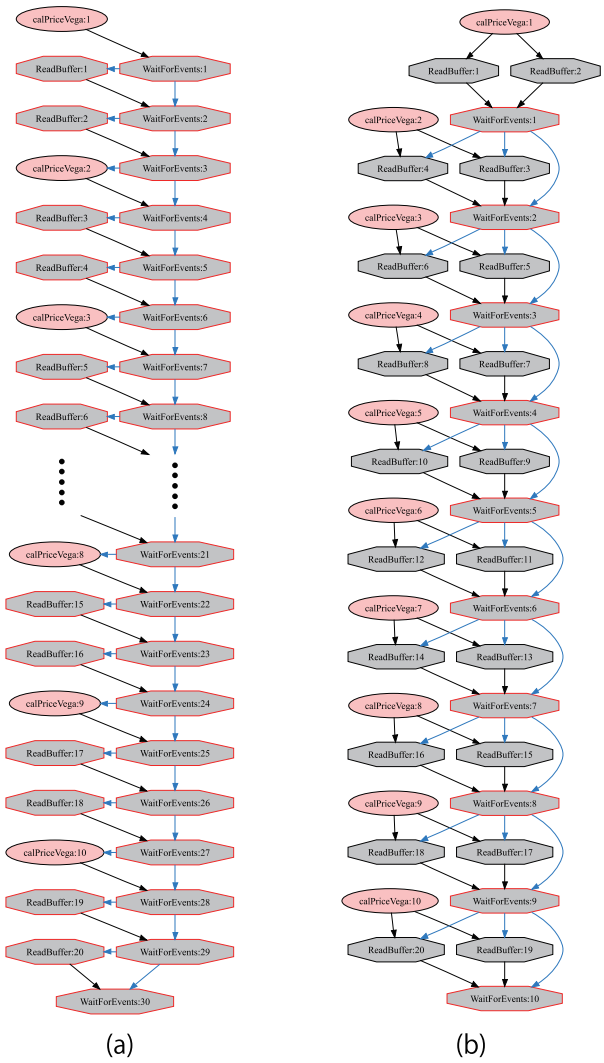


図 9　MontecarloAsian におけるイベント依存関係グラフ
Fig. 9　The event dependency graph of MontecarloAsian.

は並列化のために変更しなければならない箇所を絞り込むことができ，提案手法はタスク並列化作業の負担軽減に有効であるといえる．

4.2　タスク並列化による性能向上の評価

提案手法によって生成された依存関係グラフに基づき，プログラムを最適化した場合に得られる高速化効果を定量的に評価する．本評価では，MatrixMultiplication と MontecarloAsian について解析結果に基づいて最適化したプログラムを，1 台のデバイス（GeForce GTX 480）を用いて実行する場合（single）と，2 台のデバイス（GeForce GTX 480 + Tesla C2070）を用いて実行する場合（double）の実行時間を最適化前のプログラムを 1 台のデバイス（GeForce GTX 480）で実行する場合（original）の実行時間で正規化して比較する．また，プログラム全体の実行時間内訳では最適化前のプログラムにおける実行時間の内訳についても示す．MatrixMultiplication は，8,192 × 8,192 の正方行列間の乗算を行う．また，MontecarloAsian の計算ステップ数は，既定値である 10 回に設定されている．各ベンチマークにおける実行時間は，そのベンチマークを 10 回実行して計測された実測値の平均から求められる．プログラム全体の総実行時間と，カーネルコンパイルなどの初期化処理の時間を除いた演算実行時間の 2 種類を性能指標として用いる．MatrixMultiplication ではタスク並列性がないために，複数のデバイスを用いることによる高速化を期待できないため，1 台のデバイスで実行する場合のみを評価する．

評価結果について，演算実行時間を**図 14** (a) に，総実行時間とその内訳を図 14 (b) に示す．MontecarloAsian ではデバイスを 1 台だけ用いた場合でも演算実行時間が約 28%減少し，デバイスを 2 台用いた場合では約 40%減少している．MontecarloAsian の主要なカーネルである calPriceVega カーネルの呼び出し 1 回あたりの実行時間は，GeForce GTX 480 で約 16 ms，Tesla C2070 で約 21 ms となるため，GeForce GTX 480 を 1 としたときの Tesla C2070 の相対実効演算性能は 0.76 である．このため，GeForce GTX 480 のみを用いる場合に対して，GeForce GTX 480 と Tesla C2070 の 2 台のデバイスを用いることによって期待される最大速度向上率は 1.76 倍である．本

```
1  // 最適化前
2  steps = 10;
3  for(i = 0; i < steps; i++){
4      (set parameters to kernel);
5      clEnqueueNDRangeKernel(calPriceVega); // カーネル実行（ノンブロッキング）
6      clWaitForEvent(...); // カーネル実行完了待ち（ブロッキング）
7      clEnqueueReadBuffer(memory obj 1); // データ転送（ブロッキング）
8      clWaitForEvent(...); // データ転送完了待ち（ブロッキング）
9      clEnqueueReadBuffer(memory obj 2); // データ転送（ブロッキング）
10     clWaitForEvent(...); // データ転送完了待ち（ブロッキング）
11     (some computation);
12 }
```

```
1  // 最適化後
2  steps = 10;
3  for(i = 0; i < steps; i++){
4      (set parameters to kernel);
5      clEnqueueNDRangeKernel(calPriceVega); // カーネル実行（ノンブロッキング）
6  }
7  for(i = 0; i < steps; i++){
8      clEnqueueReadBuffer(memory obj 1); // データ転送（ノンブロッキング）
9      clEnqueueReadBuffer(memory obj 2); // データ転送（ノンブロッキング）
10     clWaitForEvent(...); // データ転送完了待ち（ブロッキング）
11     (some computation);
12 }
```

図 10　MontecarloAsian におけるコマンド投入順の最適化
Fig. 10　Command queueing optimization for MontecarloAsian.

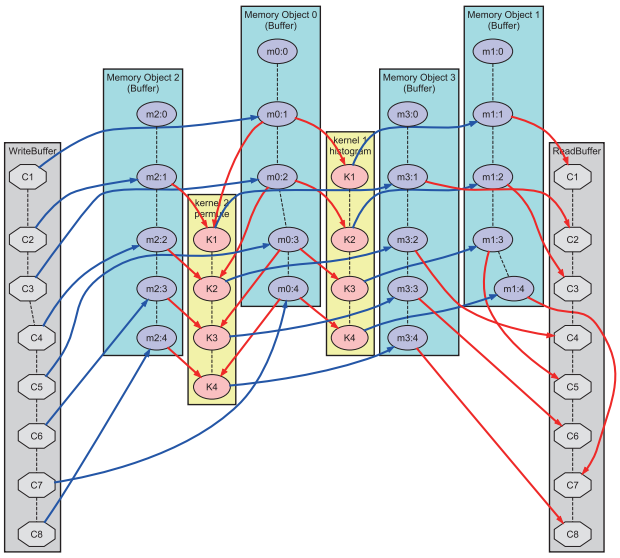


図 11　RadixSort におけるメモリアクセスグラフ
Fig. 11　The memory access graph of RadixSort.

評価環境では，最適化を適用したプログラムを 2 台のデバイスを用いて実行することで，最適化をしていないプログラム（original）を 1 台のデバイスのみを用いて実行した場合に対して約 1.68 倍，最適化を適用したプログラムを 1 台のデバイスのみを用いて実行した場合に対して約 1.21 倍の速度向上が得られた．MontecarloAsian ではデバイスでの計算結果を用いてホスト側で積分計算を行うために，不要な同期を取り除いて依存関係を整理することで，デバイスとホストの演算を並列に実行できる．その結果，デバイスを 1 台しか利用できない環境においても，高速化が達成されている．また，2 台のデバイスを利用可能な場合には，双方にカーネル実行コマンドを送ることで負荷を分散することができ，その結果として演算実行時間がさらに短縮されている．

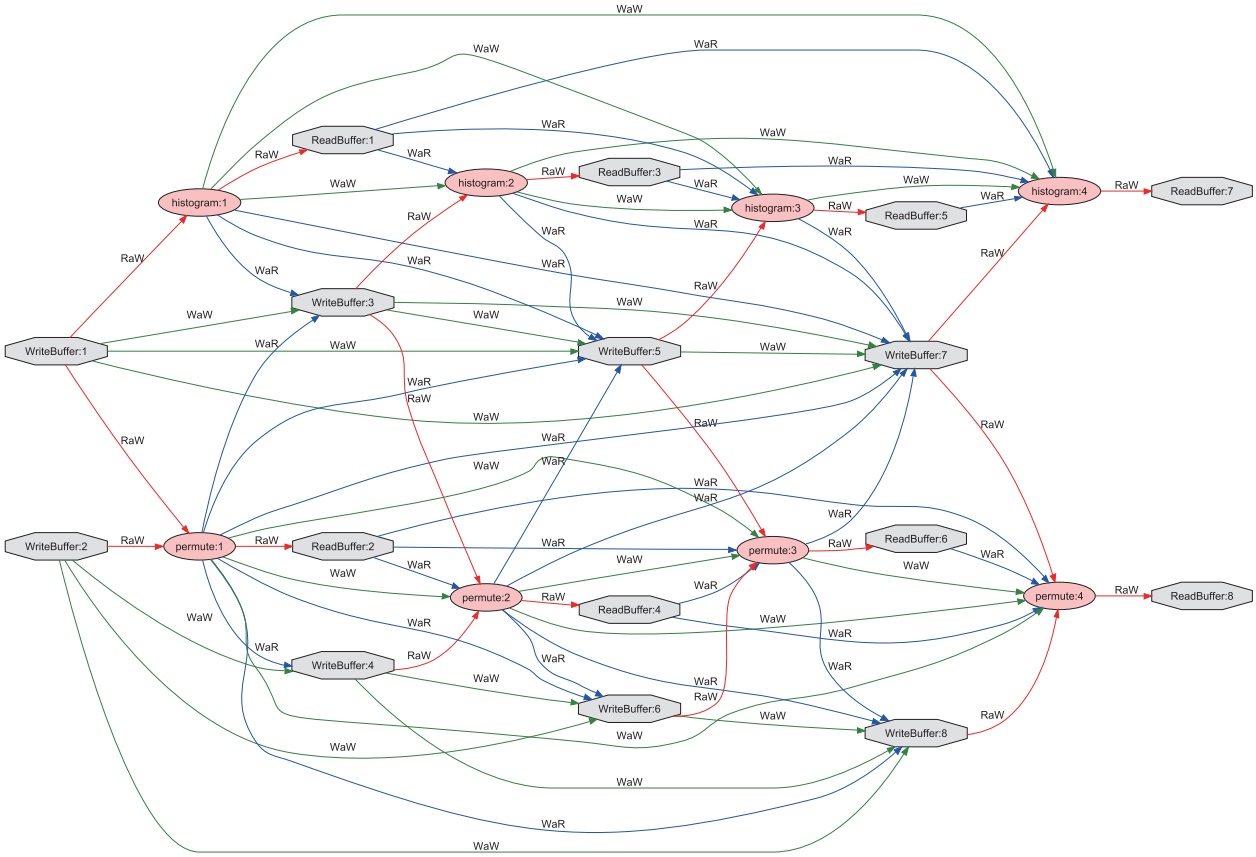
表 3　RadixSort におけるメモリオブジェクト生成時のアクセスモード
Table 3　Access modes of memory objects in RadixSort.

メモリオブジェクト	アクセスフラグ
Memory Object 0	Read Only
Memory Object 1	Write Only
Memory Object 2	Write Only
Memory Object 3	Write Only

一方で，最適化では独立して実行可能なカーネルコマンドを同じ数だけそれぞれのデバイスに割り振るようにしたため，デバイス間で演算負荷の不均衡が発生し，並列化効率の低下が見られた．10 回実行されるカーネルコマンドのすべてを GeForce GTX 480 で実行した場合，実行時間は 16 ms × 10 = 160 ms となる．GeForce GTX 480 と Tesla C2070 でカーネルの実行を 5 回ずつ均等に負荷分散をした場合は，

$$\text{GeForce GTX 480}：16\,\text{ms} \times 5 = 80\,\text{ms} \quad (1)$$

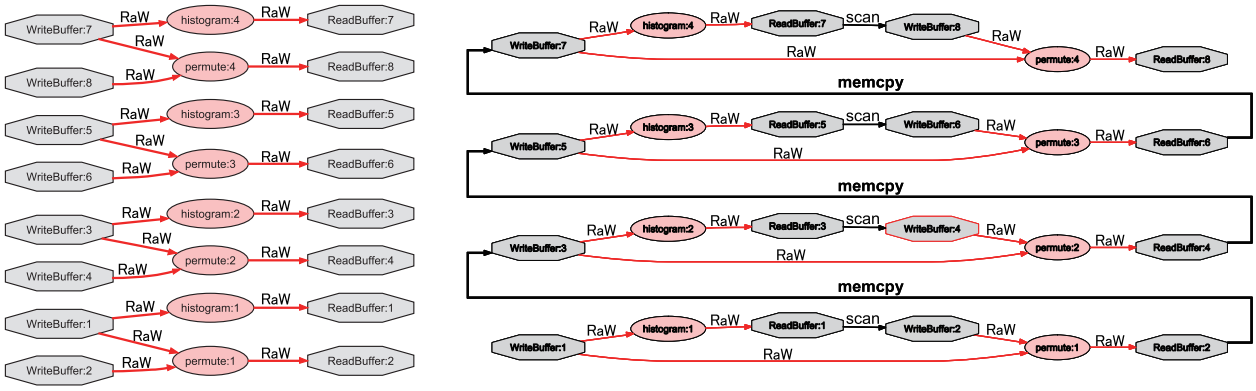
$$\text{Tesla C2070}：21\,\text{ms} \times 5 = 105\,\text{ms} \quad (2)$$



(a)

図 12　RadixSort におけるデータ依存関係グラフ（偽の依存関係を含む全依存関係）

Fig. 12　The task dependency graph of the RadixSort (All dependencies including RaW, WaW, WaR).



(a)　(b)

図 13　RadixSort におけるデータ依存関係グラフ．(a) 真のデータ依存関係のみ，(b) ホスト側の処理を考慮に加えた場合

Fig. 13　The task dependency graph of the RadixSort. (a) Only true dependencies. (b) considering with host-side processing.

となり，Tesla C2070 の実行時間に律速されるため，速度向上率の上限は $\frac{160\,\mathrm{ms}}{105\,\mathrm{ms}} \fallingdotseq 1.52$ となる．理想的な負荷分散を実現するためには，カーネルコマンドの実行数比率を GeForce GTX 480 : Tesla C2070 = 6 : 4 とする必要がある．この比率で負荷分散を行った場合，

$$\text{GeForce GTX 480}：16\,\mathrm{ms} \times 6 = 96\,\mathrm{ms} \tag{3}$$

$$\text{Tesla C2070}：21\,\mathrm{ms} \times 4 = 84\,\mathrm{ms} \tag{4}$$

となり，GeForce GTX 480 の実行時間に律速されるようになるため，最大速度向上率は $\frac{160\,\mathrm{ms}}{96\,\mathrm{ms}} \fallingdotseq 1.67$ となる．理想的な負荷分散比率はデバイスと処理内容の組合せによっ

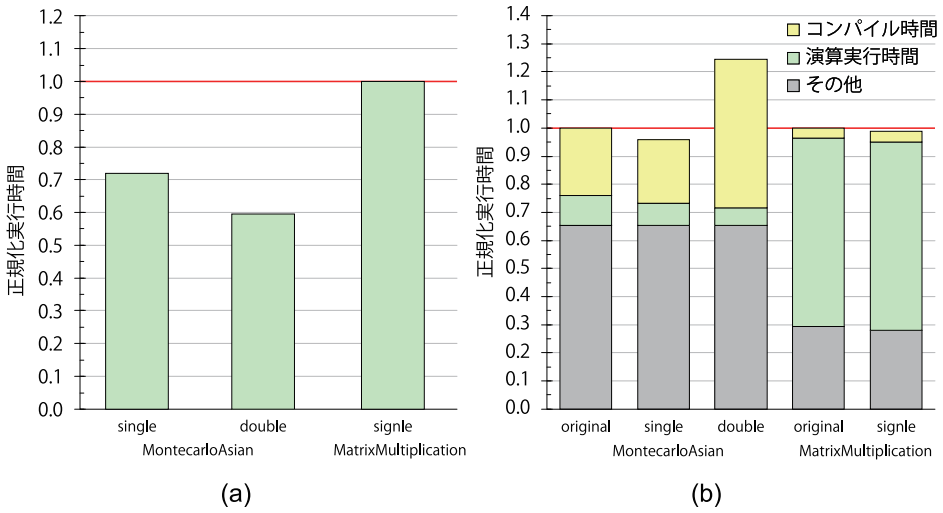


図 14　依存関係グラフに基づく最適化による高速化効果．(a) 演算実行時間，(b) 総実行時間とその内訳

Fig. 14　Evaluation results of optimization based on dependency analysis graphs. (a) kernel execution times. (b) breakdowns of total execution times.

て変化するため，プログラマが実行時に使用するデバイスに合わせて負荷分散割合をそのつど変更することは煩雑な作業となる．そのため，並列化したカーネルコマンドの負荷割当てをコマンドの実行時間を予測して [13] 自動化する手法の開発が今後の課題である．

`MontecarloAsian` では，真のデータ依存関係が存在せず，独立して実行可能なタスクがステップ数に応じて増加するが，タスク間の偽の依存関係を解消するため，タスクの数だけメモリオブジェクトを複製する必要がある．本評価では並列実行可能な 10 個のタスクをすべて並列化したために，メモリ使用量は 8.4 倍に増加した．メモリ使用量を抑制するためには，タスク並列化作業において，タスクをデバイスの台数と同じ数のグループに統合し，各デバイスにタスクグループを 1 つずつ割り当てる方法がある．この場合，グループ内ではメモリオブジェクトを再利用してタスクを逐次実行することができるため，デバイスあたりのメモリ使用量を増加させずに，デバイス間の負荷分散による速度向上を期待することができる．

また図 14 の `MontecarloAsian` の single と double を比較すると，デバイスが 1 台の場合には総実行時間が減少しているのに対して，2 台の場合には逆に増加していることが分かる．デバイスが 1 台の場合，演算時間が削減されたために総実行時間も削減されている．しかし，デバイスが 2 台の場合には，各デバイス向けにカーネルをコンパイルする必要があり，コンパイル時間が 2 倍に増えているため，演算実行時間が削減されているにもかかわらず総実行時間が増えている．`MontecarloAsian` では，演算実行時間が全体に占める割合が低くコンパイル時間が大きな割合を占めるために，複数のデバイスを用いたことによる演算実行時間の減少量をコンパイル時間の増加量が上回ったことが総実行時間の増加として現れている．しかし，実用的な OpenCL アプリケーションにおいては，演算実行時間が総実行時間の大半を占め，カーネルコンパイル時間の割合は無視できるほど小さいことが期待できる．そのため，実用的なアプリケーションにおいては，演算実行時間の短縮分だけタスク並列化による高速化を期待でき，カーネルコンパイル時間の増加は実用上問題とならないと考えられる．

`MatrixMultiplication` では，WriteBuffer コマンドをノンブロッキングコマンドに変更したことにより，デバイスへのデータ転送とホスト側での初期値設定処理がオーバラップされ，総実行時間は約 1%減少することが確認できた．このように，データ依存関係が単純で，タスク並列性が存在しないプログラムであっても，不要な同期を省くことによってホスト側とデバイス側で処理をオーバラップ実行することが可能となり，性能が向上する可能性がある．特に，データ転送オーバヘッドが比較的大きい場合に，ホスト側とオーバラップして実行できるということは，Virtual OpenCL [14] のようにネットワーク越しに他の計算システムのデバイスを利用する場合に，データ転送遅延時間の隠蔽に大きな効果を発揮することが予想される．

以上より，提案する解析手法によって生成された依存関係グラフを用いてプログラムの依存関係を整理・削減することで，実行時間を短縮可能であることが分かり，提案手法がタスク並列化作業の支援ツールとして有用であることが示された．

5. まとめ

本論文では，複数のアクセラレータを用いることで性能向上を期待できるタスク並列性を見出すための，実行時情報を用いた依存関係解析手法を提案した．提案手法では，

メモリオブジェクトへの読み書きを介してカーネルの実行順序を制約するデータ依存関係を可視化する．また，API関数の呼び出しとプログラマから与えられる明示的な依存関係を解析することでイベント依存関係を明らかとし，並列処理を阻害する同期を可視化する．これらの解析結果を用いることで，プログラマにタスク並列化を支援するための有用な情報を提示できることを示した．また，解析結果に基づいてタスク並列化を行うことで，`MontecarloAsian`では2台のアクセラレータを用いて演算時間を約40%削減可能であることを実証した．

今後の課題として，並列実行可能なカーネルを検出した場合に，実行時環境が自動的に複数のアクセラレータへカーネル実行コマンドを割り当てる機構の実現があげられる．現状では，提案手法が並列実行可能なカーネルをプログラマに提示し，プログラマが複数のコマンドキューを管理して並列実行を行うが，使用するアクセラレータの性能差に合わせて自動的に負荷分散割合を決定する手法を確立することにより，静的な割当てに比べて実効演算性能を向上させることができる．また，支援ツールとして大規模なアプリケーションに適用する場合に対応するために，複雑化する依存関係グラフの視認性を向上させる手法についても検討が必要である．

謝辞　多くの有用なコメントをくださいました査読者の方々に深く感謝いたします．

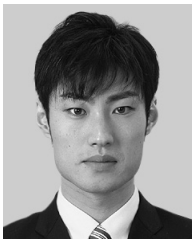
本研究は東北大学グローバルCOEプログラム流動ダイナミクス知の融合教育研究世界拠点の支援を受けています．本研究の一部は科研費若手研究（B）23700028，科研費基盤研究（B）21300007，および中山隼雄科学技術文化財団の助成によるものです．また，共著者の滝沢寛之は科学技術振興機構戦略的創造研究推進事業（JST CREST）「進化的アプローチによる超並列複合システム向け開発環境の創出」，小林広明は同事業「自己修復機能を有する3次元VLSIシステムの創製」の支援を受けています．

参考文献

[1] NVIDIA Corporation: NVIDIA CUDA C Programming Guide Version 3.2 (2010).
[2] AMD Corporation: Accelerated Parallel Processing SDK version 2.4 (2011).
[3] Khronos OpenCL Working Group: The OpenCL Specification version 1.1 (2010).
[4] Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986).
[5] Kasahara, H., Obata, M. and Ishizaka, K.: Automatic Coarse Grain Task Parallel Processing on SMP Using OpenMP, *Languages and Compilers for Parallel Computing*, Midkiff, S., Moreira, J., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J., Pugh, W. and Tseng, C.-W. (Eds.), Lecture Notes in Computer Science, Vol.2017, Springer Berlin/Heidelberg, pp.189–207 (online) (2001), available from ⟨http://dx.doi.org/10.1007/3-540-45574-4_13⟩.
[6] 稲石大祐，木村啓二，藤本謙作，尾形　航，岡本雅巳，笠原博徳：最早実行可能条件解析を用いたキャッシュ利用の最適化，情報処理学会研究報告，計算機アーキテクチャ研究会報告，Vol.98, No.70, pp.31–36（オンライン）(1998)，入手先 ⟨http://ci.nii.ac.jp/naid/110002775529/⟩.
[7] Diamos, G. and Yalamanchili, S.: Speculative execution on multi-GPU systems, *2010 IEEE International Symposium on Parallel Distributed Processing* (*IPDPS*), pp.1–12 (online), DOI: 10.1109/IPDPS.2010.5470427 (2010).
[8] Diamos, G.F. and Yalamanchili, S.: Harmony: An execution model and runtime for heterogeneous many core systems, *HPDC '08: Proc. 17th International Symposium on High Performance Distributed Computing*, pp.197–200, ACM, New York, NY, USA (online) (2008), available from ⟨http://doi.acm.org/10.1145/1383422.1383447⟩.
[9] Dolbeau, R., Bihan, S. and Bodin, F.: HMPP: A hybrid multi-core parallel programming environment (2007).
[10] The Portland Group: PGI Fortran & C Accelerator Programming Model white paper ver.1.3 (2010).
[11] Ellson, J., Gansner, E., Koutsofios, L., North, S. and Woodhull, G.: Graphviz – Open Source Graph Drawing Tools, *Graph Drawing*, Mutzel, P., Junger, M. and Leipert, S. (Eds.), Lecture Notes in Computer Science, Vol.2265, pp.594–597, Springer Berlin/Heidelberg (online) (2002), available from ⟨http://dx.doi.org/10.1007/3-540-45848-4_57⟩.
[12] AMD Corporation: ATI STREAM Computing User Guide version 2.3 (2010).
[13] Sato, K., Komatsu, K., Takizawa, H. and Kobayashi, H.: A History-based Performance Prediction Model with Profile Data Classification for Automatic Task Allocation in Heterogeneous Computing Systems, *The 9th IEEE International Symposium on Parallel and Distributed Processing with Applications* (2011).
[14] Barak, A. and Shiloh, A.: The MOSIX Management System for Linux Clusters, Multi-Clusters, GPU Clusters and Clouds (2011).

佐藤 功人　（学生会員）

平成19年東北大学工学部機械知能工学科卒業．平成21年東北大学大学院情報科学研究科修士課程修了．平成21年より東北大学大学院情報科学研究科博士後期課程に在籍．描画処理用プロセッサ（GPU）による汎用演算処理に関する研究に従事．IEEE CS会員．



小松 一彦

平成 20 年東北大学大学院情報科学研究科博士後期課程修了．博士（情報科学）．同年東北大学サイバーサイエンスセンター産学連携研究員．並列計算，大規模計算，コンピュータグラフィックスとその応用に関する研究に従事．IEEE CS 会員．

滝沢 寛之　（正会員）

平成 11 年東北大学大学院情報科学研究科博士課程修了．博士（情報科学）．同年新潟大学総合情報処理センター助手，平成 15 年東北大学情報シナジーセンター助手，平成 16 年同大学大学院情報科学研究科講師を経て，平成 21 年より同研究科准教授．高性能計算システム，コンピュータアーキテクチャとその応用に関する研究に従事．平成 16 年 ISPA04 最優秀論文賞，平成 18 年船井情報科学奨励賞，平成 20 年情報処理学会東北支部野口研究奨励賞，平成 21 年石田記念財団研究奨励賞受賞．電子情報通信学会，IEEE CS 各会員．

小林 広明　（正会員）

昭和 63 年東北大学大学院博士課程修了．同年東北大学助手．平成 3 年東北大学講師．平成 5 年東北大学助教授．平成 13 年東北大学教授（平成 20 年 4 月よりサイバーサイエンスセンター長兼任）．平成 18 年より NII 客員教授併任．コンピュータアーキテクチャ，並列処理システムとその応用に関する研究に従事．工学博士．IEEE Senior Member，ACM，電子情報通信学会各会員．