Vol.2012-HPC-136 No.15
2012/10/4

GPUにおいてパラメータスイープを高速化するための並列方式

重岡 謙太朗[1]　奥山 倫弘[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，GPU（Graphics Processing Unit）においてパラメータスイープ（PS）アプリケーションを高速化するための並列方式を提案する．提案方式では，パラメータごとの処理が互いにデータ依存を持たないことに着目し，複数パラメータの並列処理によりPSアプリケーションを高速化する．異なるパラメータに対するメモリ参照パターンが類似する可能性があるため，提案方式は入出力データをインタリーブ状に並び替える．これにより各パラメータにおける不連続なメモリ参照を一つにまとめることができ，メモリコアレッシングにより実効メモリバンド幅を向上する．さらに，パラメータ間の共通データを統合し，それらをオンチップメモリに格納することにより，オフチップメモリ参照量を削減する．実験では，4種の実アプリケーションに提案方式を適用した．結果，提案方式はグラフアプリケーションを8.5倍ほど高速化できた．一方，画像処理アプリケーションでは1.1倍の高速化に留まった．これらの結果に基づき，提案方式が高速となるアプリケーションの性質について述べる．

キーワード：パラメータスイープ，高速化，GPGPU，CUDA

A Parallel Method for Accelerating Parameter Sweep on the GPU

Kentarou Shigeoka[1]　Okuyama Tomohiro[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: This paper proposes a parallel method for accelerating parameter sweep (PS) applications on the graphics processing unit (GPU). Our method focuses on the data independence between computational tasks of parameters, accelerating PS applications by processing multiple parameters simultaneously. Our scheme arranges the input and output data in an interleaved manner, because memory access patterns from different parameters can be similar each other. This data arrangement increases the effective memory bandwidth by allowing noncontiguous memory accesses from each parameter to be coalesced into a single access. Furthermore, our method reduces the amount of off-chip memory accesses by unifying the common data for multiple parameters and by storing the data in on-chip memory. In experiments, we apply our method to four practical applications. As a result, our method achieves a speedup of 8.5 times for a graph application. In contrast, the speedup of an image processing application results in a factor of 1.1. According to the experimental results, we summarize the characteristics of applications that can be accelerated with our method.

Keywords: Parameter sweep, acceleration, GPGPU, CUDA

1. はじめに

PSは組み合わせ最適化問題を解くための手法としてよく知られている．パラメータ空間から解を得るために，PSアプリケーションは異なるパラメータに対して同一の処理を適用する．実際の応用では組み合わせの数が膨大であるため，PSアプリケーションは高性能システムにより高速化されている．

計算グリッドは高性能システムの1つであり，PS計算が内包する粗粒度の並列性を利用して高速化を図ってい

[1] 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

る [1], [2]．各パラメータの処理はデータ依存に関して独立であるため，PS アプリケーションはマスタ・ワーカ方式により効率よく並列化できる．この方式では，マスタは一連のパラメータを遊休ワーカに順に割り当てる．

もう一つの重要な高性能システムとして GPU が挙げられる．柔軟な開発環境として CUDA （Compute Unified Device Architecture）[3] が登場して以来，GPU はグラフィクスだけでなく，高いメモリバンド幅を必要とする汎用アプリケーションに対するアクセラレータとして注目されている．GPU は，数千個のスレッド上で細粒度の並列性を利用し，CPU に対して 10 倍の高速化を達成することが多い．

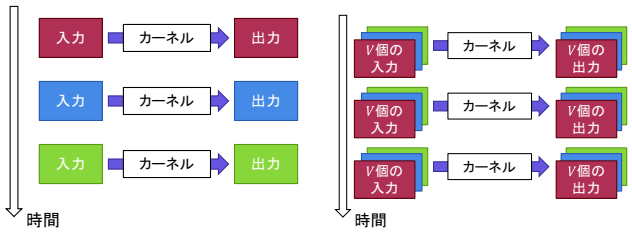
そこで，多くのグリッドシステムは GPU により高速化を図っている [4], [5]．例えば，Folding@home[4] では，たんぱく質の折り畳みシミュレーションを 2 万台の GPU を用いて高速化している．GPU 上のスレッドは，単一パラメータの計算が内包する細粒度のデータ並列性を利用している．GPU はシステムにおける計算ノードの 10%に過ぎないが，この高並列計算により全体に対して 70%の計算スループットを提供している．このように，GPU はグリッドシステムにおける貢献を増していて，PS アプリケーションを GPU 上で加速するための効率のよい並列方式が必要である．

そこで，本稿では CUDA 互換の GPU において PS アプリケーションを高速化するための並列方式を提案する．既存方式と同様に，提案方式は PS アプリケーションのデータ並列性に着目する．既存方式との違いは，単一の替わりに複数のパラメータを並列処理することである．このようなタスク構成により，パラメータ間の類似性を利用できる．例えば，単一パラメータの不規則なデータ参照を複数パラメータの規則的なデータ参照に変換し，GPU 上に効率よく実装できる．さらに，提案方式は複数のパラメータが共通して参照するデータを統合し，それらをオンチップメモリに格納することによりオフチップメモリのバンド幅を節約する．

以降では，まず 2 節で既存方式を含む予備知識についてまとめる．次に，3 節で提案方式を示す．4 節で評価実験の結果を示し，5 節で本稿をまとめる．

2. CUDA におけるパラメータスイープ

本稿では，PS アプリケーションが走査するパラメータの数を n とし，単一パラメータに対する処理をタスクと呼ぶ．パラメータ P_i $(1 \le i \le n)$ に対するタスクを $\mathcal{T}_i$ とおく．また，P_i に対する入力および出力データの集合をそれぞれ $\mathcal{I}_i$ および $\mathcal{O}_i$ $(1 \le i \le n)$ とおく．簡単のため，$\mathcal{I}_i$ および $\mathcal{O}_i$ は同数 m の要素を持つとする．このとき，i 番目の入力および出力データは以下のように表せる．



(a) 既存方式　(b) 提案方式

図 1 パラメータスイープの並列方式

$$\mathcal{I}_i = \{\, e_{i,j} \mid 1 \le j \le m \,\} \tag{1}$$

$$\mathcal{O}_i = \{\, f_{i,j} \mid 1 \le j \le m \,\} \tag{2}$$

ここで，$e_{i,j}$ および $f_{i,j}$ は，それぞれ i 番目の入力および出力データにおける j 番目の要素を表す．例えば，画像処理における $\mathcal{I}_i$ および $\mathcal{O}_i$ はそれぞれ入力画像および出力画像に対応し，$e_{i,j}$ および $f_{i,j}$ はそれらの画素に対応する．すべてのタスクが共通に参照する入力データの部分集合 $\mathcal{C}$ は次式で記述できる．

$$\mathcal{C} = \bigcap_{1 \le i \le n} \mathcal{I}_i \tag{3}$$

1 節で述べたように，マスタ・ワーカ方式は粗粒度の並列性を用い，ワーカに対して任意のタスクを割り当てる．この際，既存方式は GPU 上の SIMT（Single Instruction Multiple Thread）演算器を用いて単一タスクを並列処理する（**図 1**(a)）．この方式では，ワーカが入力データ $\mathcal{I}_i$ をメインメモリからビデオメモリに転送し，タスク $\mathcal{T}_i$ に対応するカーネル [3] を GPU 上で実行する．その後，出力データ $\mathcal{O}_i$ をメインメモリへ転送する．これらのステップを，i を 1 から n まで増やしながら反復処理する．

カーネルの性能を最大化するためには，以下に挙げる 3 つの設計点を実現する必要がある．

(1) メモリコアレッシングによるオフチップメモリ遅延の隠蔽．
(2) 小容量のオンチップメモリによるオフチップメモリ参照の削減．
(3) スレッドごとの資源節約による常駐スレッド [3] の増大．

3. 提案する並列方式

GPU 上に PS アプリケーションを効率よく実装するために，我々は PS が内包する 2 つの特徴に着目する．

(1) すべてのパラメータ $P_1, P_2, \ldots, P_n$ が同一の処理を適用すること．
(2) すべてのパラメータ $P_1, P_2, \ldots, P_n$ が入力データの共通部分集合 $\mathcal{C}$ を参照すること．

提案方式は特徴 (1) に基づいて複数のタスク $\mathcal{T}_i, \mathcal{T}_{i+1}, \ldots, \mathcal{T}_{i+V-1}$ を並列処理する．ここで，V は SIMT 演算器において同時に処理するパラメータの数を示す

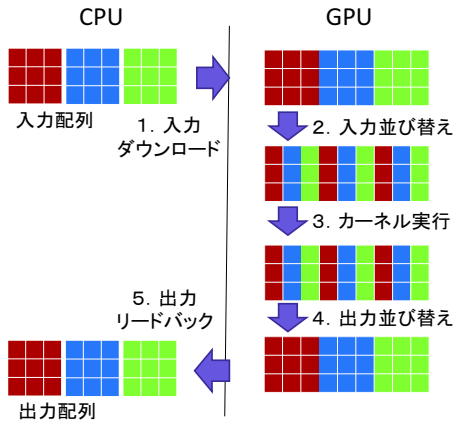


図 2 入出力データの並び替え

(図 1(b))．これにより GPU が持つ SIMT 演算器の利点を生かせる．現在の演算器はワープを処理の単位としているため，V の値としてワープサイズ 32 を用いる．さらに，提案方式は特徴 (2) にしたがい，V 個のタスク間で共通データ $\mathcal{C}$ を統合し，メモリ消費量を削減する．これにより統合データを共有メモリに格納できる可能性が高まる．

図 2 に，提案方式の概要を示す．単一パラメータを並列処理する既存方式とは異なり，提案方式は V 個のパラメータに対するメモリコアレッシングを実現するために，入力および出力データの並び替えを必要とする．この並び替えはオーバヘッドを伴う．ただし，CPU が並び替えを担当する場合，並び替えをカーネル実行とオーバラップでき，オーバヘッドを隠蔽できる．もう一つの違いは，呼び出し 1 回あたりのカーネル実行時間が長いことである．1 個のパラメータを並列処理する既存方式と比較して，我々のカーネル実行時間は V 倍ほど長くなる可能性がある．しかし，カーネル実行回数はおよそ $1/V$ に減少する．したがって，パラメータの数 n が十分に多ければ，この問題は無視できる．

提案方式におけるカーネルの記述方法を理解するために，以降ではメモリコアレッシングおよびデータ統合について説明する．

3.1 オフチップメモリ遅延を隠蔽するメモリコアレッシング

図 3 に示すように，提案方式は複数のメモリトランザクションを 1 つにまとめることができるよう，入力データ $\mathcal{I}_i$ および出力データ $\mathcal{O}_i$ を並び替え，カーネル性能の向上を図る．特に，各パラメータのメモリ参照パターンが不規則だが，それらが類似したメモリ参照ストライドを持つ場合，提案方式は有用である．そのような不規則なメモリ参照のコアレッシングは容易でないため，既存方式では低い性能となる．このように，PS 計算が内包する SIMT の特徴は複数パラメータのためのメモリコアレッシングに向く．

最新のアーキテクチャでは，カーネルが以下の条件を満たせば，メモリコアレッシングを実現できる．

(1) タスク割当に関する条件：同一ワープ内のスレッドが V 個のタスク $\mathcal{T}_i, \mathcal{T}_{i+1}, \ldots, \mathcal{T}_{i+V-1}$ を担当すること．

(2) データ構造に関する条件：タスク $\mathcal{T}_i, \mathcal{T}_{i+1}, \ldots, \mathcal{T}_{i+V-1}$ が参照する入出力データがインタリーブ状に格納されていること．

最初の条件を満たすために，我々は V 個の連続するタスク $\mathcal{T}_i, \mathcal{T}_{i+1}, \ldots, \mathcal{T}_{i+V-1}$ をワープに割り当てる．なお，ワープに単一タスクを割り当てる既存方式は，この条件を満たせない．既存方式と同様に，提案方式はそれぞれのタスクを数千個のワープを用いて並列処理する．

上記に加え，2 つ目の条件はメモリコアレッシングを実現するために必要である．仮に，タスク $\mathcal{T}_i$ が入力要素 $e_{i,j}$ $(1 \leq i \leq n, 1 \leq j \leq m)$ を参照する場合を考える．異なるパラメータに同一の処理を適用する PS アプリケーションでは，他のタスク $\mathcal{T}_{i+1}, \mathcal{T}_{i+2}, \ldots, \mathcal{T}_{i+V-1}$ が同一の相対番地 j に存在する入力要素 $e_{i+1,j}, e_{i+2,j}, \ldots, e_{i+V-1,j}$ を参照する可能性が高い (図 3(a))．したがって，図 3(b) に示すように，提案方式は入力および出力データをインタリーブ状に並び替える．つまり，元の並び $e_{i,1}, e_{i,2}, \ldots, e_{i,m}$ の代わりに，$e_{i,j}, e_{i+1,j}, \ldots, e_{i+V-1,j}$ を連続番地に格納する．これにより，異なるタスクが同一サイクルにおいて連続番地を参照できる．

なお，共通データ $\mathcal{C}$ の並び替えは不要であることに注意されたい．共通データに対しては，すべてのタスクがデータ統合 (後述) により同じメモリ領域を共有している．したがって，$\forall 1 \leq i \leq n,\ \mathcal{I}_i = \mathcal{C}$ である場合，入力データの並び替えは不要である．

高々 2 次元のスレッドブロックを用いるコードに対しては，スレッドブロックの次元を増やすこと (拡張) により，上記のタスク割り当てを実現できる．例えば，2 次元のスレッドブロックサイズ (X, Y) に対しては，ワープが V 個のタスクを処理できるように，3 次元のスレッドブロックサイズ (V, X, Y) に拡張する．スレッドブロックが上限の 3 次元に達している場合，あらかじめ次元を削減しておく必要がある．この削減は 3 次元から 2 次元への変換により実現できる．

ここで，スレッドブロックのサイズがアーキテクチャに制限されていることに注意されたい．変換後のサイズが上限値を超える場合，タスクあたりのスレッド数を削減し許容サイズに適合させる必要がある．スレッド間にデータ依存がない場合，この削減はスレッドブロックの分解により実現する．そうでない場合，スレッド集約，つまりスレッドあたりの仕事量を増やすことにより実現する．同様に，スレッド集約はスレッドブロックの数 (グリッドサイズ) を適合させるために使用できる．

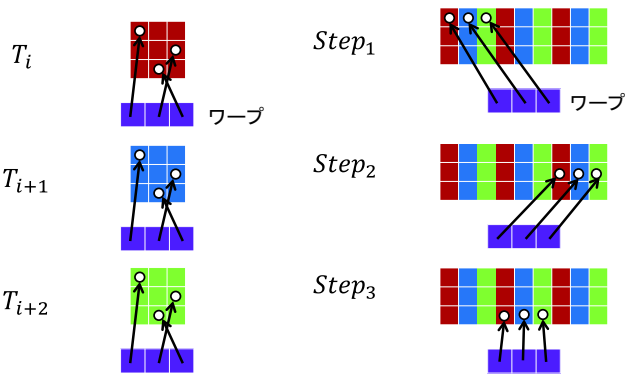


図 3 提案方式のスケーラビリティ

3.2 オフチップメモリ参照を削減するデータ統合

提案方式は V 個のパラメータを並列処理しているため，共通データ $\mathcal{C}$ を統合してメモリ使用量を節約できる．これにより，入力データサイズを $(V-1)|\mathcal{C}|$ だけ削減できるだけでなく，$\mathcal{C}$ が共有メモリ上に存在していればオフチップメモリの参照量を削減できる．このとき同一ワープ内のスレッドは $1/V$ のデータを参照すればよいため，オフチップメモリの参照量は少なくとも $1/V$ に削減できる．最良の場合，同一スレッドブロック内のスレッドが $\mathcal{C}$ を共有できるため，データ統合によりさらにオフチップメモリの参照量を削減できる．

提案方式では，同一スレッドブロック内のスレッドがデータをオフチップメモリから共有メモリにコピーする．この際，冗長なメモリトランザクションを防ぐために，スレッドは協調してコピーする．この後，スレッドはオフチップメモリの代わりに共有メモリを参照して計算できる．最後に，カーネル実行を終えるまでに計算結果をオフチップメモリにコピーする必要がある．

3.3 並列方式選択のガイドライン

既存方式と比べて提案方式は利点および欠点を持つ．したがって，提案方式は既存方式よりも常に高速であるとは限らない．任意のアプリケーションに対して最善の方式を予測するのは難しいため，ここでは開発者のための選択ガイドラインを示す．このガイドラインは 2 節で示した 3 つの設計点を基にしている．

まず，メモリコアレッシングの効果は対象アルゴリズムが持つメモリ参照パターンに依存する．したがって，アルゴリズムのメモリ参照ストライドを調べる必要がある．アルゴリズムがタスク間で類似した参照パターン（参照ストライド）を持つ場合，提案方式はメモリコアレッシングを実現する．一方，既存方式はタスク内で連続したメモリ番地を参照する場合，メモリコアレッシングを実現する．したがって，メモリ参照の類似性と連続性がメモリコアレッシングの効果を決定する．もし対象アルゴリズムが類似性および連続性の双方を持つ場合，あるいは双方ともに持たない場合，残りの設計点を調べる必要がある．

次に，共有メモリのみならずデータ統合により，オフチップメモリ参照量を削減できる．3.2 節で述べたように，データ統合により共有メモリ上の共通データ $\mathcal{C}$ を V 個のタスク間で再利用できる．一方，既存方式はタスク内でデータを再利用するために共有メモリを用いる．提案方式は V 個のタスクを同時に処理するため，各タスクが専有できるメモリ空間は $1/V$ に限られる．したがって，タスク内およびタスク間のデータ再利用はトレードオフの関係にある．

最後に，提案方式は V 個のタスクを並列処理するため，既存方式よりも V 倍のメモリ空間を必要とする．共有メモリの容量は限られているため，提案方式の常駐スレッドは既存方式のものよりも減少する可能性がある．同様の問題はレジスタの枯渇により引き起こされる可能性がある．このように，提案方式は資源使用量に関して欠点を持つ．その対策としては，スレッド数の削減やカーネルの分割が挙げられる．ただし，V 個のタスクがオフチップメモリを枯渇させてしまう場合，提案方式をそのまま用いることはできない．

上記のガイドラインはカーネルの性能を決める．しかし，提案方式の欠点である並び替えのオーバヘッドを考慮する必要がある．特に，類似性および連続性の双方を持つ，あるいは双方とも持たない場合，そのオーバヘッドを調べる必要がある．

4. 評価実験

本節では，提案方式を性能向上の点から評価する．そのために提案方式を 4 つの実アプリケーションに適用し，提案方式および既存方式の実行時間を比較する．実験で用いた実アプリケーションは，ガウシアンフィルタ（GF），ニューラルネットワーク（NN），結合ヒストグラム（JH）および全点対最短経路問題（APSP）である．これらの詳細を**表 1** に示す．

ここで GF は，共有メモリのバンクコンフリクト [3] を回避するために，スレッドブロックサイズとして $(16, 8, 1)$ の替わりに $(32, 8, 1)$ を用いていることに注意されたい．

実験では，1.5 GB のオフチップメモリを持つ NVIDIA Geforce GTX580 および Core i7 2500K を搭載する Windows7 Professional 64bit の PC を用いた，ディスプレイドライバは 301.32 である．また，コンパイルには CUDA 4.2 および Visual Studio 2008 を用い，最適化オプションとして O2 を使用した．

4.1 提案方式の適用手順

まず，アプリケーションごとにメモリ参照の類似性および連続性の有無を調べた．**表 2** に示す結果に基づき，$V = 32$ 個のタスクを並列処理するようにプログラムを変

表 1 実験用アプリケーションの詳細

アプリケーション	NN[6]	APSP[7]	JH[8]	GF[9]
類似性	あり	あり	なし	あり
連続性	なし	なし	なし	あり
入力/出力 並び替え	あり/あり	なし/あり	あり/あり	あり/あり
詳細	ガウシアン関数の可分なフィルタを用いて画像を畳み込む．1 タスクは 1024 × 1024 画素の画像処理に対応する．	手書き文字を認識するためのニューラルネットワーク．1 タスクは数字を含む 29×29 画素の画像認識に対応する．	医用画像位置合わせのための結合ヒストグラムを作成する．1 タスクは 512 × 512 × 16 ボクセルからなる 1 組のボリュームデータの処理に対応する．	頂点数 32K の重み付きグラフの全点対最短経路を計算する．1 タスクはグラフにおける 1 つの単一始点最短経路の計算に対応する．

表 2 カーネルの詳細

アプリケーション	カーネル	変数	属性	類似性	連続性	統合	サイズ（B）	領域
NN	FirstLayer	Layer1_Neurons	$\mathcal{I}$	あり	なし	なし	3.6 K	G
		Layer1_Weights	$\mathcal{C}$	あり	あり	あり	0.6 K	G
		Layer2_Neurons	$\mathcal{O}$	あり	あり	なし	4.0 K	G
	SecondLayer	Layer2_Neurons	$\mathcal{I}$	あり	なし	なし	4.0 K	G
		Layer2_Weights	$\mathcal{C}$	あり	あり	あり	30.5 K	G
		Layer3_Neurons	$\mathcal{O}$	あり	あり	なし	4.9 K	G
	ThirdLayer	Layer3_Neurons	$\mathcal{I}$	あり	なし	なし	4.9 K	G
		Layer3_Weights	$\mathcal{C}$	あり	なし	あり	488.7 K	G
		Layer4_Neurons	$\mathcal{O}$	あり	なし	なし	0.4 K	G
	ForthLayer	Layer4_Neurons	$\mathcal{I}$	あり	なし	なし	0.4 K	G
		Layer4_Weights	$\mathcal{C}$	あり	なし	あり	4.0 K	G
		Layer5_Neurons	$\mathcal{O}$	あり	なし	なし	40.0	G
APSP	SSSPKernel1	Va	$\mathcal{C}$	あり	あり	あり	128 K	G
		Ea	$\mathcal{C}$	あり	なし	あり	512 K	G
		Wa	$\mathcal{C}$	あり	なし	あり	512 K	G
		Ma	$\mathcal{O}$	あり	あり	なし	128 K	G
		Ca	$\mathcal{O}$	あり	あり	なし	128 K	G
		Ua	$\mathcal{O}$	あり	なし	なし	128 K	G
	SSSPKernel2	Ma	$\mathcal{I}$	あり	あり	あり	128 K	G
		Ca	$\mathcal{I}$	あり	あり	なし	128 K	G
		Ua	$\mathcal{I}$	あり	あり	なし	128 K	G
		F	$\mathcal{O}$	あり	あり	あり	4	G
JH	MakeHistogram	fdat	$\mathcal{I}$	あり	あり	なし	8 M	G
		rdat	$\mathcal{C}$	あり	あり	あり	8 M	G
		hist2d	$\mathcal{O}$	なし	なし	なし	256 K	G
GF	Rows	d_Dst	$\mathcal{I}$	あり	あり	なし	4 M	G+S
		c_Kernel	$\mathcal{C}$	あり	あり	あり	68	C
		d_Src	$\mathcal{O}$	あり	あり	なし	4 M	G
	Columns	d_Dst	$\mathcal{I}$	あり	あり	なし	4 M	G+S
		c_Kernel	$\mathcal{C}$	あり	あり	あり	68	C
		d_Src	$\mathcal{O}$	あり	あり	なし	4 M	G

更した．

NN は，人工ニューラルネットワークに基づき手書き文字を認識する．このプログラムは 4 つのカーネルを持ち，各々はネットワークの各層を担当する．すべてのタスクが同じニューラルネットワークを共有するため，その重み係数は $\mathcal{C}$ とみなせる．すべてのカーネルは 3.1 節で述べたスレッドブロック拡張により提案方式を適用できる．しかし，FirstLayer ではスレッドブロックサイズの上限を超えてしまうため，その次元を削減した．結果，スレッドブロックのサイズ $(13, 13, 1)$ はスレッド集約により $(13, 1, 1)$ となり，スレッドブロック拡張により $(32, 13, 1)$ となった．

表 2 に示すように，APSP は 2 つのカーネルを持つ．

SSSPKernel1 カーネルは共通データ $\mathcal{C}$ を入力にしているため，提案方式ではそれらを共有メモリに格納した．スレッドブロックのサイズ $(256, 1, 1)$ は，スレッドブロック分解により $(4, 1, 1)$ とし，スレッドブロック拡張により $(32, 4, 1)$ とした．同様の手順に基づき，SSSPKernel2 カーネルではサイズを $(256, 1, 1)$ から $(64, 4, 1)$ に置換した．SSSPKernel1 と比べて 32 の替わりに 64 を選択した理由は，パーティションキャンピングを回避するためである．スレッドブロックのサイズが $(32, 4, 1)$ である場合，最大で 7 倍の性能低下が起こり得る [10].

JH は 1 組の参照データおよび浮動データに対し，それらの結合ヒストグラムを計算する．計算時において，参照データは 1 つに固定されている一方，浮動データはパラメータとみなして置き換えられる．したがって，参照データは共通データ $\mathcal{C}$ とみなせ，データ統合を施せる．スレッドブロックのサイズ $(32, 32, 1)$ は，スレッドブロック分解により $(32, 1, 1)$ とし，スレッドブロック拡張により $(32, 32, 1)$ とした．

最後に，GF はガウシアン関数に基づき，低次元分解可能な畳み込み演算を 2 次元信号に適用する．フィルタは 2 つのカーネルからなり，それぞれ行および列方向の 1 次元畳み込み演算を担当する．両カーネルは定数メモリにガウシアン係数を保持し，タスク間で再利用する．カーネルはタイリングにより高速化されており，共有メモリにより作業用変数を共有できる．スレッドブロックのサイズ $(32, 8, 1)$ はスレッドブロック分解により $(8, 1, 1)$ とし，スレッドブロック拡張により $(32, 8, 1)$ とした．

4.2 アプリケーションの性質および速度向上率

提案方式および既存方式による実行時間を比較し，3.3 節において述べたアプリケーションの性質が，提案方式の性能向上の度合を決めることを示す．提案方式および既存方式について，実行時間の内訳を**図 4(a)** から**図 4(d)** に示す．これらのアプリケーションでは，提案方式により既存方式の高速化を達成した．特に APSP および NN ではそれぞれ，8.5 倍，7.7 倍の高速化となった．これは，表 1 に示すように，アプリケーションが連続性を持たず類似性を持ち，複数パラメータの参照によりメモリコアレッシングを実現できるためである．

NN 内の 4 つのカーネルは提案方式により高速化できる．まず，FirstLayer カーネルでは，プロファイラ結果において，occupancy[3] が 0.12 から 0.26 となり，2.2 倍増加している．これは，より多くのメモリ参照のレイテンシを隠蔽できたことを示す．また，SecondLayer についても FirstLayer と同様の理由から高速化できている．プロファイラ結果では，occupancy は 0.08 から 0.48 となり，6.0 倍増加している．

ThirdLayer の高速化は，提案方式によりワープ内の全スレッドが処理するように変更したことに起因する．このカーネルのスレッドブロックサイズは $(1, 1, 1)$ であるため，ワープ内の 1 スレッドのみが処理をする．提案方式により $(32, 1, 1)$ に拡張でき，ワープ内の全スレッドが処理するよう変更できる．プロファイラ結果では，分岐数が 273K から 8.5K へと 1/32 に削減されている．ForthLayer は ThirdLayer と同様の理由により高速化される．スレッドブロックサイズ $(1, 1, 1)$ を $(32, 1, 1)$ に拡張することにより，分岐数を 3264 から 102 に削減できる．

次に APSP 内の 2 つのカーネルは，ダイバージェントブランチの削減により，それぞれ 7.4 倍，7.7 倍の高速化を達成している．プロファイラ結果によると，SSSPKernel1 および SSSPKernel2 では，ダイバージェントブランチ数をそれぞれ 90M から 25M，6M から 2M まで削減できる．なぜならば，並列処理するパラメータ数と共有メモリのバンク数を一致させ，各バンクに別々のパラメータのみを配置するようにしたためである．ワープ内の各スレッドはそれぞれ 1 つのパラメータを処理するため，異なるバンクを参照するようになり，ダイバージェントブランチがなくなる．ただし，共通データ $\mathcal{C}$ を共有メモリに格納した場合は，全タスクでデータを共有するため，ダイバージェントブランチが発生する可能性がある．

また，NN および APSP ではリードバックの実行時間がそれぞれ 25 倍，14.8 倍高速化されている．これは，リードバックする出力データのサイズが小さいため，データ転送時間が遅延時間に支配されるためである．つまり，NN の出力 Layer5_Neurons および APSP の出力 F は，それぞれ 10 B，4 B と小さく，提案方式により $V = 32$ 倍のサイズを転送しても，リードバック時間が大きく増加しない．

一方，JH および GF は高速化がわずかであり，それぞれ 1.09 倍および 1.01 倍である．JH については，連続性を持たず，類似性も持たないプログラムであり，どちらの方式を用いてもメモリコアレッシングを実現できない．しかし，カーネルが 1.13 倍の高速化がなされたのは，参照データの参照を統合することにより，メモリ参照遅延を削減できるためである．プロファイラでは，L1 キャッシュヒット率が 0%から 73.4%まで増加している．さらに，提案方式では，各タスクのパラメータがメモリ領域を共有しないため，アトミックな演算によるメモリ参照の逐次化が発生しない．そのためプロファイラ結果では，グローバルメモリのキャッシュミスによる命令繰り返しの割合を示す，Global Memory Cache Replay Overhead が 2.57%から 0.36%に減少している．

次に，GF は連続性と類似性をともに持つプログラムであり，どちらの方式においてもメモリコアレッシングを実現できる．GF の 2 つのカーネル RowsKernel および ColumnsKernel は，それぞれ 1.54 倍，1.29 倍の高速化がなされている． これは，L2 キャッシュヒット回数の増加

が原因である．Columns カーネルのプロファイラ結果では，L2 キャッシュヒット率が 2.3 倍，Rows カーネルでは 1.4 倍に増加している．

4.3 スケーラビリティ

最後に，提案方式のスケーラビリティを評価する．**図 5** に，APSP および JH において問題サイズを変化させたときの既存方式に対する提案方式の速度向上率を示す．APSP ではグラフの頂点数を 1K から 2M まで変化させ，JH については 512×512 画素を含む 2 次元画像（スライス）の数を 1 から 64 まで変化させた．

APSP では，頂点数が増大するにつれて，速度向上率が 21 倍から 2.6 倍に低下している．この理由は，頂点数が少ない場合，既存方式において常駐スレッドが不足するためである．頂点数が十分に多い場合，既存方式はメモリ遅延を隠蔽するための常駐スレッド数を確保でき効率が向上する．一方，$V = 32$ 個のタスクを並列処理する提案方式では，頂点数が少ない場合においても高い効率を実現できている．

同様の傾向は，JH においても確認できる，つまり，1 個のスライスに対しては速度向上率が 1.26 倍であるのに対し，41 個のスライスに対しては 1.08 倍まで低下している．以上から，提案方式は単一タスクにおいて常駐スレッドが不足する応用に有用である．

なお，図 5(b) では 42 個以上のスライスに対して速度向上率を計測できていない．この理由は，オフチップメモリの容量不足が原因で提案方式の実行に失敗するためである．このように，$V = 32$ 個のタスクを並列処理する提案方式は，計算資源の不足を引き起こす可能性がある．今回の実験環境では，提案方式は既存方式と比較して $1/V$ のスライス数を処理できる．一方，APSP に関しては 1/6 の頂点数を並列処理できる．APSP では，大部分の入力データが共通データ $\mathcal{C}$ であり，それらのメモリ領域を統合できるため，より大きな問題サイズを扱える．このように，入出力データサイズおよびカーネルのメモリ使用量から，提案方式が適用できるかを判断しなければならない．

5. まとめ

本稿では，GPU 上においてパラメータスイープを高速化するための並列方式を提案した．提案方式は，プログラムに単純な変更をすることにより，一つのカーネルにおいて，単一のタスクではなく複数のタスクを並列処理する．これにより，並列処理するスレッド間においてメモリ参照が類似する．そのため，提案方式では入出力データをインタリーブ状に並び替えることにより，メモリコアレッシングを実現し，スループットを向上する．さらに，複数のパラメータ間で共通して参照するデータをオンチップメモリ上に格納して共有することにより，タスク処理に必要なオフチップメモリのバンド幅を削減している．

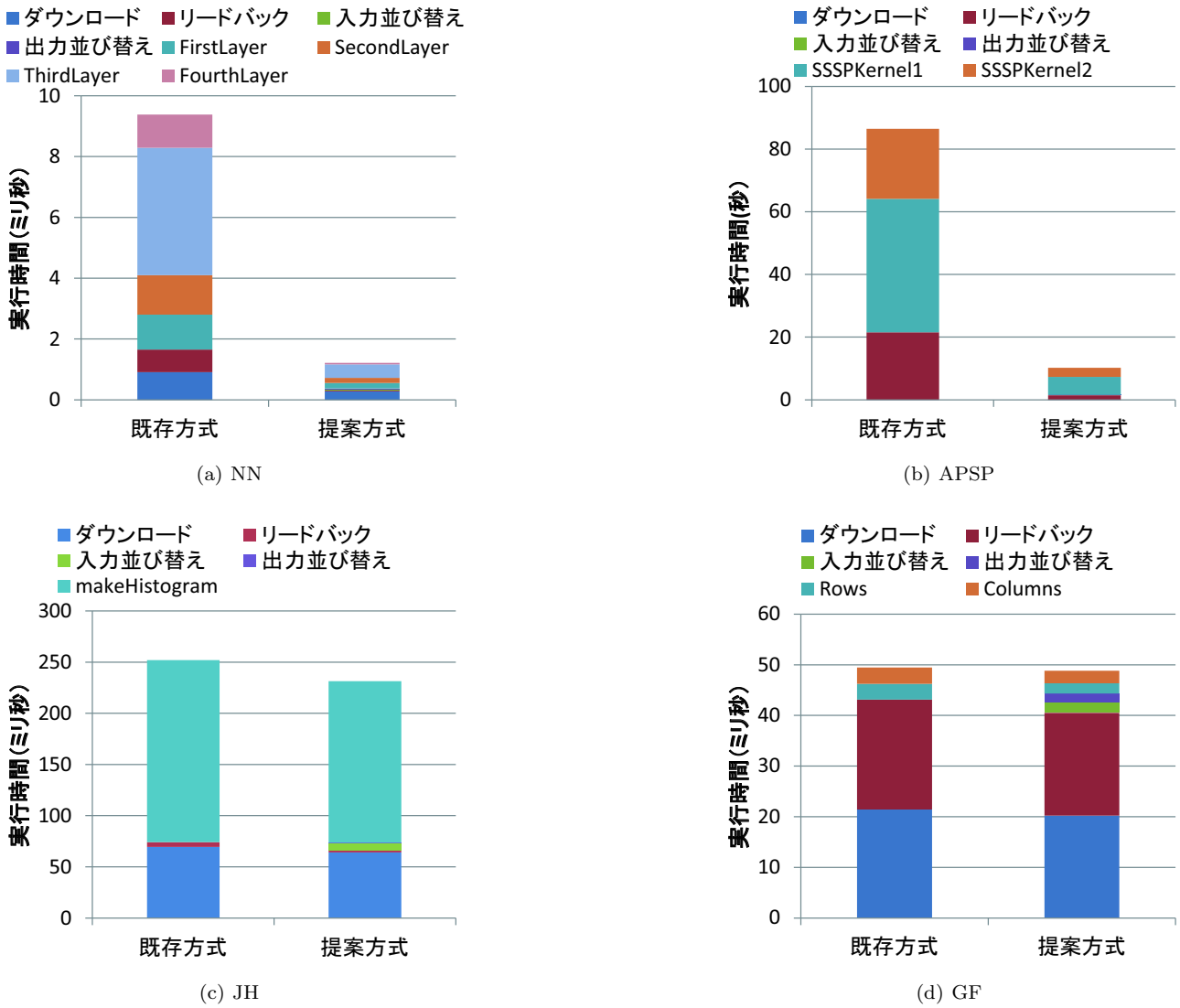
評価実験では，複数の実アプリケーションに対して提案方式を適用した．結果，グラフアプリケーションにおいて最大 8.5 倍の実行時間の速度向上率を得た．一方，画像処理アプリケーションにおいては，1.08 倍の速度向上率しか得られなかった．これは，データインテンシブなアプリケーションであり，入出力の並び替えの実行時間がボトルネックとなるためである．さらに，データサイズを増やした場合でも我々の方式が有効であることを確認した．

今後の課題は，連続性を持ち類似性を持たないアプリケーションに対する提案方式の適用を評価すること，および自動的に我々の並列方式を適用するツールを作成することである．

謝辞 本研究の一部は，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」，科研費 23300007，および 23700057 の補助による．

参考文献

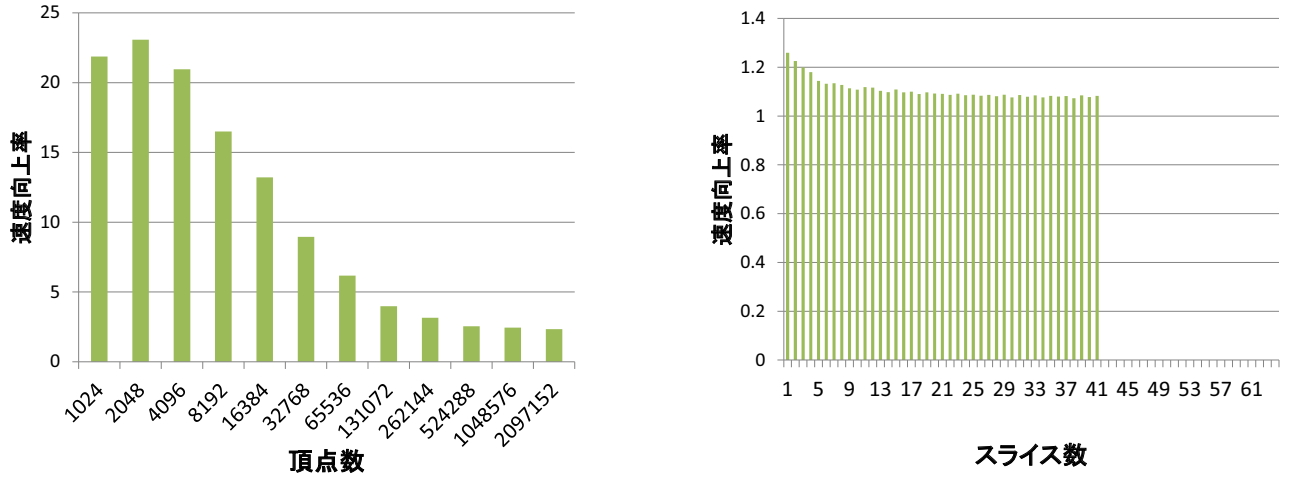
[1] Olabarriaga, S. D., Nederveen, A. J. and Nualláin, B. O.: Parameter Sweeps for Functional MRI Research in the “Virtual Laboratory for e-Science” Project, *Proc. 17th IEEE Int’l Symp. Cluster Computing and the Grid (CCGrid’07)*, pp. 685–690 (2007).
[2] Youn, C. and Kaiser, T.: Management of a parameter sweep for scientific applications on cluster environments, *Concurrency and Computation: Practice and Experience*, Vol. 22, No. 18, pp. 2381–2400 (2010).
[3] NVIDIA Corporation: CUDA Programming Guide Version 4.2 (2012).
[4] The Folding@Home Project: Folding@Home Distributed Computing (2010). `http://folding.stanford.edu/`.
[5] Ino, F., Munekawa, Y. and Hagihara, K.: Sequence Homology Search using Fine-Grained Cycle Sharing of Idle GPUs, *IEEE Trans. Parallel and Distributed Systems*, Vol. 23, No. 4, pp. 751–759 (2012).
[6] NVIDIA Corporation: GPU Computing SDK (2012).
[7] billconan and kavinguy: A Neural Network on GPU (2008). `http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU`.
[8] Ikeda, K., Ino, F. and Hagihara, K.: Accelerating Joint Histogram Computation for Image Registration on the GPU, *Proc. Computer Assisted Radiology and Surgery: 26th Int’l Congress and Exhibition (CARS’12)*, pp. S72–S73 (2012).
[9] Harish, P. and Narayanan, P. J.: Accelerating Large Graph Algorithms on the GPU using CUDA, *Proc. 14th Int’l Conf. High Performance Computing (HiPC’07)*, pp. 197–208 (2007).
[10] Aji, A. M., Daga, M. and chun Feng, W.: Bounding the Effect of Partition Camping in GPU Kernels, *Proc. 8th Int’l Conf. Computing Frontiers (CF’11)* (2011). 10 pages.



(a) NN (b) APSP

(c) JH (d) GF

図 4 アプリケーションの実行時間



(a) APSP (b) JH

図 5 提案方式のスケーラビリティ