Vol.2012-HPC-133 No.3
2012/3/26

CUDA プログラムにおけるメモリ参照効率を解析するための実行履歴生成手法

神田裕士[†1]　奥山倫弘[†1]
伊野文彦[†1]　萩原兼一[†1]

本稿では CUDA（Compute Unified Device Architecture）プログラムにおけるメモリ参照効率を解析するための実行履歴生成手法を提案する．メモリ参照命令の前後においてタイムスタンプを記録する単純な手法では，メモリ参照の完了前にタイムスタンプを記録してしまい，正確な実行履歴は得られない．そこで，提案手法はメモリ参照時間を正確に計測するために，メモリ参照を終えたのちにタイムスタンプを記録する．提案手法は対象とするメモリ領域に応じて 2 種類の方式を使い分ける．実験では提案手法および単純手法を用いて実アプリケーションにおけるメモリ参照時間を計測した．結果，提案手法は使用者に解析すべき箇所を正しく提示できた．

An Instrumentation Method for Analyzing Efficiency of Memory Access in CUDA Programs

Hiroto Kanda,[†1] Tomohiro Okuyama,[†1]
Fumihiko Ino[†1] and Kenichi Hagihara[†1]

This paper presents a log generation method for analyzing the efficiency of memory access in compute unified device architecture (CUDA) programs. A simple method that obtains time stamps before and after memory access fails to generate accurate logs for CUDA programs, because time stamps are recorded before the completion of memory access. The proposed method obtains time stamps after completing memory access to precisely measure memory access time. Our method uses two schemes according to the memory region to be analyzed. In experiments, we measured the memory access time of an application with the proposed method and the simple method. As a result, the proposed method gives users the correct point that should be analyzed for performance improvement.

1. はじめに

GPU（Graphics Processing Unit）[1] はグラフィクス処理用の並列プロセッサであり，CPU を上回る演算性能を持つ．近年，GPU 向け開発環境 CUDA（Compute Unified Device Architecture）[2] を用い，汎用計算を高速化する試みが盛んである．CUDA は SIMT（Single Instruction Multiple Thread）と呼ばれるアーキテクチャを持ち，数千個ものスレッドを順次切り替えながら，カーネルと呼ばれる関数を並列処理する．

並列プログラムの性能を解析するための手段として，実行履歴に基づく手法がある．ここで実行履歴とは，プログラム実行時に記録したタイムスタンプを時系列に並べたものである．実行履歴はプログラム実行時の挙動を提示でき，開発者を支援できる．例えば，スレッドにおける進捗のばらつきを示す，あるいは処理時間の内訳を示すことにより性能ボトルネックを特定できる．

CUDA プログラム向けの性能解析ツールとして，Compute Visual Profiler[3] がある．このツールでは GPU が実行した命令数や参照したデータ量などの統計情報を取得できる．しかし，実行履歴を取得する機能は提供していないため，カーネルにおける処理時間の内訳などは得られない．一般に CUDA は多数のスレッドを用いて，大量のデータを並列処理するため，メモリ参照が性能ボトルネックとなりやすい．したがって，実行履歴を用いてメモリ参照に要した時間を計測することは，性能ボトルネックを特定しメモリ参照効率を解析するために有用である．

単純な計測手法として，メモリ参照の前後でタイムスタンプを記録し，それらの差分をメモリ参照時間とする手法がある．しかし，CUDA はメモリ参照命令の完了を待たずにデータ依存のない演算命令を実行する．タイムスタンプの記録は計測対象の処理にデータ依存を持たないため，参照時間を正しく計測できない．

そこで本稿では，CUDA プログラムのメモリ参照効率を解析することを目的として，メモリ参照時間を正確に計測できる実行履歴生成手法を提案する．提案手法は，メモリ参照命令を完了させたのちにタイムスタンプを記録する．完了を保証する手法は計測対象とするメモリ領域に応じて 2 種類ある．GPU から読み書きできるメモリ領域に対しては CUDA が提供するメモリフェンス命令を用いる．一方，読み込みのみ可能なメモリ領域に対しては，メモリ参照命令とデータ依存のあるダミー命令を意図的に挿入する．また，メモリ参照時間

†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

を正確に計測するために，参照遅延の小さいオンチップメモリにタイムスタンプを一時的に記録しておき，プログラムの最後に遅延の大きいオフチップメモリに書き出す．

以降では，2 章で関連研究について述べ，3 章で CUDA の概要について説明する．次に，4 章で単純な計測手法について説明し，メモリ参照時間を計測する際の問題点を示す．5 章で提案手法について述べる．6 章で提案手法を評価し，メモリ参照効率の解析例を示す．最後に 7 章で本稿をまとめる．

2. 関 連 研 究

Farooqui ら[4] は CUDA プログラムの中間言語である PTX コード[5] に対して計測用の処理を挿入することによりタイムスタンプなどの情報を取得し，CUDA プログラムを解析する手法を提案している．得られた情報を基に，GPU 内のプロセッサ間における負荷分散や，プログラムにおいて頻繁に実行される処理などについて解析している．しかし，メモリ参照時間の計測やメモリ参照効率の解析を想定した手法ではない．

Malony ら[6] は TAU（Tuning and Analysis Utilities）[7] など並列計算環境向けのツールを用いて CPU および GPU をもつ異種な環境における性能を計測するための手法を提案している．CPU のスレッドに関する情報と複数の GPU における性能情報を組み合わせて提示する機能などを提供しているが，Compute Visual Profiler と同様に GPU における実行履歴を生成する機能は提供していない．

3. CUDA（Compute Unified Device Architecture）

GPU は内部に SM（Streaming Multiprocessor）と呼ばれるプロセッサを複数搭載している．カーネルを実行するスレッドは TB（Thread Brock）と呼ばれるグループに分割されており，各スレッドは TB ごとに資源が許す限り SM に割り当てられる．

SM は割り当てられた TB 内のスレッドを，ワープと呼ばれる 32 スレッドごとのグループに分ける．各ワープは 1 つの共通した命令を同一のタイミングで実行する．実行するワープは即時に切り替えられ，SM 内のワープスケジューラが命令を実行する準備ができているスレッドを持つワープを選択し，命令を実行する．したがって，あるワープにおけるメモリ参照遅延を，ワープの切り替えにより別の演算命令を実行することで隠蔽できる．

3.1 CUDA のメモリ階層

GPU のメモリ階層は共有メモリおよびデバイスメモリに分類できる．共有メモリは各 SM 内にあるオンチップメモリであり，同一 SM 内で実行するスレッドからのみ参照可能である．共有メモリは小容量だが，レジスタと同程度の遅延で参照できる．また，共有メモリは同一 SM 内で実行される TB が共有する資源であり，TB ごとの共有メモリの使用量が大きいほど同時に実行可能なスレッドの数は減少する．

一方，デバイスメモリはオフチップメモリであり，GPU 上で実行するすべてのスレッドが参照できる．デバイスメモリは CPU から参照可能である．デバイスメモリは大容量である一方，参照時の遅延は 400〜600 クロック程度と低速である．デバイスメモリはキャッシュ機構を備えており，それにより参照遅延を短縮できる．共有メモリをキャッシュの代わりとして使用することにより，グローバルメモリへの参照回数を削減することが重要となる．デバイスメモリはさらにグローバルメモリ，テクスチャメモリおよびコンスタントメモリと呼ばれる領域に分類できる．グローバルメモリはすべてのスレッドから読み書きが可能な領域である．その他のメモリ領域は読み込み専用の領域であり，CPU からのみ書き込み可能である．

3.2 実行履歴の生成に用いる CUDA の関数

本研究では実行履歴を生成するために，CUDA が提供するメモリフェンス関数およびクロック取得関数を使用する．

メモリフェンス関数は，実行したスレッドをそれ以前に実行したメモリ参照命令の結果が他のスレッドから利用可能となるまで待機させる．メモリフェンス関数には`__threadfence()` 関数や`__threadfence_block()` 関数など複数の種類があり，それぞれメモリ参照の結果が利用可能であることを判定する対象となるスレッドの範囲が異なる．前者は GPU 上のすべてのスレッド，後者は同一 TB 内のすべてのスレッドを対象とする．

クロック取得関数である `clock()` 関数は，実行することにより GPU のクロックごとにインクリメントされるカウンタの値を返す．同一ワープ内のスレッドは同一のタイミングで命令を実行することから，`clock()` 関数で取得できる値も同一である．

4. タイムスタンプを用いて処理時間を計測する手法

本章ではタイムスタンプを用いて，ある処理 P の処理時間 t_p，すなわち P が開始してから終了するまでに要する時間を計測する手法について述べる．また，その手法を用いてメモリ参照の所要時間を計測するときの問題点を示す．

単純な手法として，処理 P の前後にタイムスタンプ記録処理を挿入する手法がある．**図 1** に単純手法を用いた計測のコード例を示す．N 行のプログラム $\{S_1, S_2, \cdots, S_N\}$ を考える．ここで S_i は第 i 行目の文を意味する．プログラム内の処理 P が第 i 行目から始まる M 行

```
1: i := スレッド番号 ;
2: t1 := clock(); // 処理前のタイムスタンプ記録 R1
3: d = D[i]; // 計測対象の処理 P
4: t2 :=clock(); // 処理後のタイムスタンプ記録 R2
5: O[i] := d + 1; // 計算結果の出力
6: tp[i] := T2 − T1; // 処理時間
```

図 1 単純な計測手法のコード例

の文からなるとすれば，$P = \{S_i, S_{i+1}, \cdots, S_{i+M-1}\}$ と表せる．このとき P の実行開始前および終了後の時刻を得ることを目的に，S_{i-1} および S_i の間，S_{i+M-1} および S_{i+M} の間にそれぞれクロック取得命令を挿入する．挿入後のプログラムを実行することで P の前後におけるタイムスタンプを記録する．S_{i-1} および S_i の間で実行するタイムスタンプ記録処理を R_1，S_{i+M-1} および S_{i+M} の間に実行するタイムスタンプ記録処理を R_2 とする．そして，R_1 および R_2 で記録したタイムスタンプの値をそれぞれ t_1 および t_2 とするとき，単純手法を用いて計測した処理時間 $t_p{}' = t_2 - t_1$ となる．

ただし，$t_p{}'$ は t_p の近似である．t_1 および t_2 を P の開始および終了と同一のタイミングでは記録できないためである．P の本来の開始時間および終了時間をそれぞれ t_{start} および t_{end} とすれば，以下の条件が成立する．このとき，処理時間を正確に計測できているという．**図 2(a)** に正確な計測ができる場合の t_p と $t_p{}'$ との時間的な関係を示す．

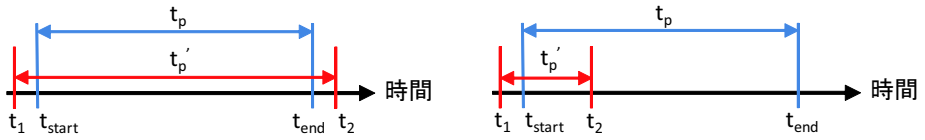
$$t_1 < t_{start} < t_{end} < t_2 \quad (1)$$

逆に，条件 (1) が成立していない場合，P が計測区間に含まれないため，その処理時間を正しく計測しているといえない．

CUDA プログラムにおいて P がメモリ参照処理である場合，単純な方法では正確な処理時間を計測できない．メモリ参照命令の実行後，その命令とデータ依存のない命令はメモリ参照の完了を待たずに実行される．タイムスタンプ記録は対象プログラムの処理に対してデータ依存のない処理である．したがって，R_2 はメモリ参照の完了前に実行されてしまうため，以下の式が成立する．**図 2(b)** に正確な計測ができない場合の t_p と $t_p{}'$ との時間的な関係を示す．

$$t_1 < t_{start} < t_2 < t_{end} \quad (2)$$

以上のことから，単純な手法ではメモリ参照が完了するまでの時間を正確に計測できない．



(a) 正確に計測できる場合　(b) 正確に計測できない場合

図 2 本来の処理時間 t_p と単純手法により計測した処理時間 $t_p{}'$ との関係

5. 提案手法

提案手法について述べる．まず，メモリ参照時間の正確な計測手法について述べる．その後タイムスタンプの記録方法について説明し，最後に CUDA プログラムにおいて実行履歴を生成するまでの全体の流れを示す．

5.1 メモリ参照時間の計測手法

メモリ参照の完了後にタイムスタンプを記録し，メモリ参照時間を正確に計測するための手法について述べる．以下では，メモリ参照処理 P_A の処理時間を計測することを目的とする．本手法は，計測する対象となるメモリ領域の種類により 2 種類の方式を使い分ける．グローバルメモリなど GPU 上のスレッドから読み書きが可能なメモリ領域に対してはメモリフェンス命令による方式を用いる．一方，テクスチャメモリなど GPU 上のスレッドからは読み込みのみ可能なメモリ領域に対してはデータ依存を生じさせる方式を用いる．以下でそれぞれの方式について説明する．

メモリフェンスによる方式は，CUDA が提供するメモリフェンス命令を用いてメモリ参照時間を計測する．**図 3** に本手法の適用例を示す．P_A の実行後に，まず `__threadfence_block()` 関数を実行し，その後にタイムスタンプの記録処理 R_2 を実行する．これにより R_2 を実行する時点で P_A が完了していることが保証されるため，P_A の処理時間を計測できる．なお，メモリフェンスとして `__threadfence_block()` 関数を用いる理由は，TB 内のスレッドから処理の結果が参照可能となった時点でメモリ参照は完了しているためである．また，待機する時間も他のメモリフェンス関数よりも短い．

次に，データ依存を生じさせる処理を挿入する方式について述べる．本方式は，タイムスタンプの記録処理とメモリ参照の結果にデータ依存がないために，正確な参照時間を計測できない，という点に着目する．P_A の実行後に，まず P_A とデータ依存のある処理 P_D を実

```
1: i := スレッド番号 ;
2: t_1 :=clock(); // 処理前のタイムスタンプ記録 R_1
3: d = D[i]; // 計測対象のメモリ参照処理 P_A
4: __threadfence_block(); // メモリフェンス関数
5: t_2 :=clock(); // 処理後のタイムスタンプ記録 R_2
6: O[i] := d + 1; // 計算結果の出力
```

図 3 メモリフェンス命令による方式のコード例

行する．このとき P_D を実行する時点で P_A は完了していなければならない．その後，R_2 を実行することにより R_2 を実行するタイミングは P_A が完了した後であることが保証される．以下に P_D が満たすべき条件を示す．

条件 C_1　P_A の処理結果を使用する処理であること

P_A および P_D の間にデータ依存を生じさせるためにこの条件が必要である．

条件 C_2　対象プログラムにおける出力結果と依存がある処理であること．出力結果とはグローバルメモリに書き込む値のことであり，タイムスタンプそのものも含む．

P_D は本来の対象プログラムが出力結果を得る過程において不要な処理である．そのような処理は，コンパイラが最適化時に削除する可能性が高い．コンパイラによる P_D の削除を防ぐために，この条件が必要である．

条件 C_3　挿入済プログラムの出力結果が元のプログラムの出力結果と同一であること

条件 C_2 を満たす処理をプログラムに挿入した場合，挿入の前後でプログラムの出力結果が異なる可能性がある．出力結果が異なる場合，プログラムが正しく動作していることを判断できないことから，実行履歴から得られる実行状況が元の実行状況に対して正確であることも保証できない．したがって，この条件を満たすべきである．

本方式では，条件 C_1〜C_3 を満たし，かつ P_D の処理時間 t_d をできる限り短くすることを目指す．この方式により得られるメモリ参照時間は，t_d を含む．したがって，t_d の長い P_D の挿入により正確なメモリ参照時間が得られなくなる．さらに，そのような P_D の挿入により，対象プログラムの挙動が変化し，性能を正しく解析できない可能性がある．

以上のことを踏まえて，P_A の結果を使用するダミー処理を P_d として挿入することを提案する．**図 4** にダミー処理を挿入する方式の例を示す．4，5 行目がダミー処理にあたり，ここで変数 $dummy$ はカーネルの引数として与えられ，呼び出し時に 0 が代入されているものとする．このダミー処理は条件 C_1 および C_2 を満たす処理が，ある条件下でのみ実行されるよう，条件分岐を用いて記述したものである．この条件分岐に与える条件式は実行時でないと評価できず，かつ評価結果は常に偽となるものとする．図 4 の例においては，$dummy$ はコンパイル時にその値を判断できず，かつ実行時には必ず 0 であるため評価結果は偽となる．結果として，挿入した処理は条件判定および分岐命令のみ実行される．この処理は条件 C_1 および C_2 を満たしており，かつプログラムの出力結果は維持できることから条件 C_3 を満たす．また条件分岐の本体である処理の内容に関わらず，条件判定および分岐命令のみが実行されるため，t_d の発生を抑制できる．以上より，P_D としてふさわしいと言える．

```
1: i := スレッド番号 ;
2: t_1 :=clock(); // 処理前のタイムスタンプ記録 R_1
3: d = D[i]; // 計測対象のメモリ参照処理 P_A
4: if dummy < 0 then // ダミー処理
5:     d := d + 1; // ダミー処理
6: t_2 :=clock(); // 処理後のタイムスタンプ記録 R_2
7: O[i] := d + 1; // 計算結果の出力
```

図 4 ダミー処理を使用する方式のコード例

5.2 タイムスタンプの記録手法

CUDA カーネル内部でタイムスタンプを記録する方法について述べる．まず，タイムスタンプとして記録する値は CUDA が提供する `clock()` 関数を実行して得られる値とする．

実行履歴は GPU プログラムの実行終了後に CPU 側でファイルとして出力することから，タイムスタンプはグローバルメモリに記録する必要がある．しかし，グローバルメモリは参照時の遅延が大きいことから，タイムスタンプを記録するたびにグローバルメモリを参照する場合，記録のオーバヘッドが大きくなる．

そこで，参照遅延の小さい共有メモリを一時的に使用してタイムスタンプを保持することによりオーバヘッドを削減する．ただし，共有メモリの使用量が増加したことを原因として，同時に実行可能なスレッド数が減少することは避けたい．同時に実行可能なスレッド数が異なれば実行状況も異なり，性能も変化してしまうためである．共有メモリの使用量を削減することを目的として，同一ワープ内のスレッドによる同一箇所におけるタイムスタンプの記録は，同一のアドレスに書き込むこととする．このとき，ワープ内のいずれか 1 スレッドが書き込んだ内容が記録され，それ以外のデータは残らない．しかし，先に述べたとおり同一ワープ内のスレッドが書き込むデータはすべて同一であり，ワープごとのタイムスタンプを記録するという目的は達成できる．

この手法を使用する場合でも，記録するタイムスタンプの数とともに共有メモリの使用量

が多くなれば，同時に実行可能なスレッド数が減少しうる．その場合は同時に実行可能なスレッド数を維持することを優先し，共有メモリを使用せずにグローバルメモリに記録する．

5.3 実行履歴生成までの流れ

カーネルを実行する前に，タイムスタンプを記録するための領域をグローバルメモリ上に確保する．確保する領域は TB の個数，TB 内のスレッド数，および記録するタイムスタンプ数により決定する．カーネル実行時には，5.2 節で述べた手法によりグローバルメモリにタイムスタンプを記録する．カーネルの実行終了後，タイムスタンプをグローバルメモリから CPU のメインメモリに転送し，それを基にして CPU 側で実行履歴を生成する．

実行履歴は CSV（Comma Separated Values）形式と CLOG 形式[8] の 2 種類を生成する．CSV 形式は汎用性が高く，表計算ソフトなどで内容を確認できるが，単純な数字の羅列であるため可読性は低い．一方，CLOG 形式は MPE 向けの可視化ツールである Jumpshot を用いてガントチャート形式で表示できる．

6. 評価実験

本章では提案手法を用いて計測したメモリ参照時間が対象アプリケーションの特性を正しく示せることを評価する．また，得られた履歴を用いて，実アプリケーションのメモリ参照効率を解析した結果を述べる．

6.1 実験内容

実験に用いるアプリケーションは CUDA SDK[9] の VR（Volume Rendering）である．VR はメモリ参照が性能ボトルネックである．提案手法の有効性を検証するために，提案手法および単純手法を用いて計測したメモリ参照時間を比較する．サイズが $1024 \times 1024 \times 1024$ ボクセルのボリュームを対象とした．VR におけるテクスチャメモリ参照を計測対象とし，各ワープの参照 1 回あたりの平均所要時間を算出する．さらに計算など，メモリ参照以外の処理時間も同様に正規化し，メモリ参照とそれ以外の処理のいずれが性能ボトルネックであるかを確認する．

また，VR は実行時のパラメータに依存して異なる性能が得られる．そのパラメータとは視点位置および TB の形状である．ここで性能を決定する主な要因はテクスチャメモリのキャッシュヒット率であることが知られている．テクスチャメモリのキャッシュヒット率を以下では TC ヒット率と呼ぶ．メモリ参照時間を正確に計測できていれば，VR の性能および TC ヒット率とメモリ参照時間には相関があるはずである．そこで，相関の有無を確認するために，同一視点において複数の TB 形状で実行し，履歴を生成した．VR の性能を示す指標としてはフレームレート（FR：Frame Rate）を用いる．視点位置は，ボリュームを x 軸方向に α 度，y 軸方向に β 度，z 軸方向に γ 度回転させたとき，(α, β, γ) と表す．また，TB 形状については，TB の x 方向のスレッド数を a，y 方向のスレッド数を b とするとき，$a \times b$ と表す．TB あたりのスレッド数は 128 に固定する．

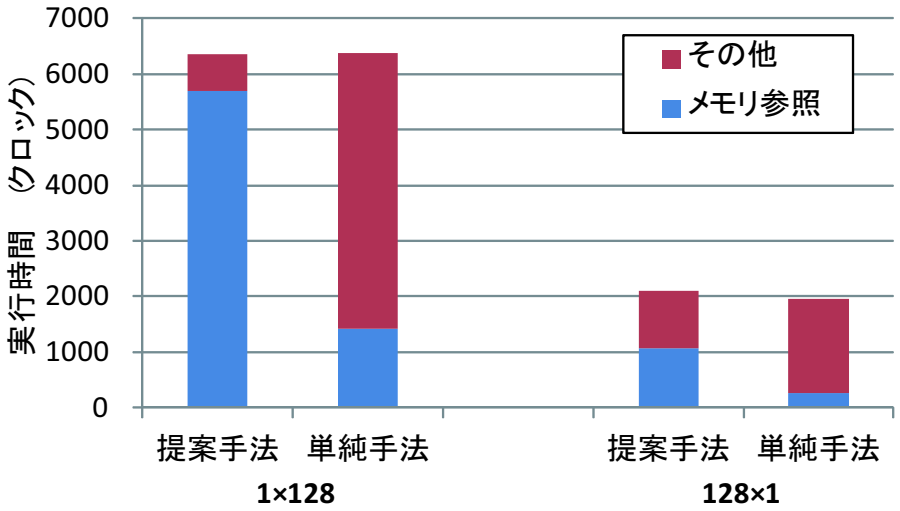


図 5 提案手法と単純手法の比較

実験環境は CPU が Intel Xeon E5440（2.83GHz），GPU が NVIDIA GeForce GTX 480，CUDA バージョン 4.0 を用いた．

6.2 性能ボトルネックの特定

図 5 に単純手法および提案手法を用いてメモリ参照とその他の処理時間を計測した結果を示す．視点 (0,0,0)，TB 形状 1×128 および 128×1 において実験した．この視点においては TB 形状 1×128 が VR の性能が低い条件であり，128×1 が性能の高い条件である．

単純手法を用いた場合，メモリ参照時間が全体の処理時間に占める割合は TB の形状 128×1 においては約 13%，1×128 においては約 20%である．このデータからは VR の性能ボトルネックはメモリ参照以外の処理であり，TB 形状の変更による性能の変化はそれらの処理が原因であると認識できる．結果として，使用者は解析すべき箇所の判断を誤ることとなる．

一方，提案手法を用いた場合，メモリ参照時間が全体の処理時間に占める割合は TB の形状が 1×128 のとき約 50%，1×128 のとき約 80%である．この結果からは VR の性能ボトルネックがメモリ参照であると認識できる．

以上のことから提案手法を使用することにより，使用者がプログラム中の解析すべき箇所を正しく示せることを確認できた．

6.3 メモリ参照時間と性能およびキャッシュヒット率の相関

単純手法および提案手法のそれぞれを用いて，視点 (0,0,0) における各 TB 形状でのメモリ参照時間と TC ヒット率および FR との相関を調べた．ここではメモリ参照時間の逆数をとったメモリ参照頻度という尺度を考える．メモリ参照頻度が高いほどメモリ参照効率は高いと言え，TC ヒット率および FR とは正の相関が見られるはずである．

図 6 に単純手法および提案手法で計測したメモリ参照頻度と TC ヒット率を比較した結果を示す．**図 6(b)** より単純手法を用いて計測した場合，x 方向の長さが 8 以上である TB の形状において TC ヒット率とメモリ参照頻度の傾向が一致していない．TC ヒット率がより高い形状においてメモリ参照頻度が低く，TC ヒット率がより低い場合にメモリ参照頻度が高いという負の相関が見られる．

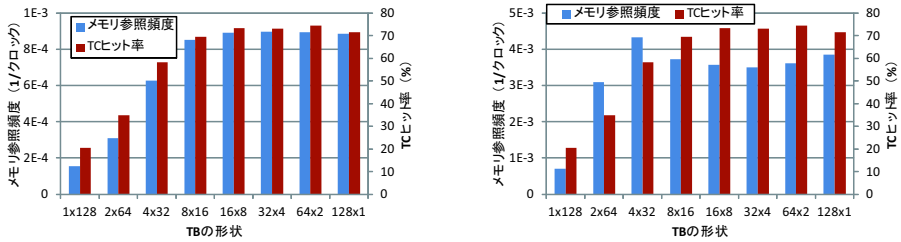
一方，**図 6(a)** より提案手法を用いた場合は，すべての TB 形状において TC ヒット率とメモリ参照頻度が増減する傾向が概ね一致しており，両者に強い正の相関があるとわかる．

次に**図 7** に単純手法および提案手法で計測したメモリ参照頻度と FR を比較した結果を示す．これらの結果からも TC ヒット率と比較した場合と同様の傾向が見られる．すなわち単純手法を用いた場合，TB の x 方向の長さが 8 以上であるときにメモリ参照頻度と FR に負の相関がある．一方，提案手法を用いた場合，TB の形状に関わらず TC ヒット率とメモリ参照頻度の間に強い正の相関がある．

以上のことから，提案手法を用いた計測の方がより対象アプリケーションの特徴を正しくとらえることができているといえる．

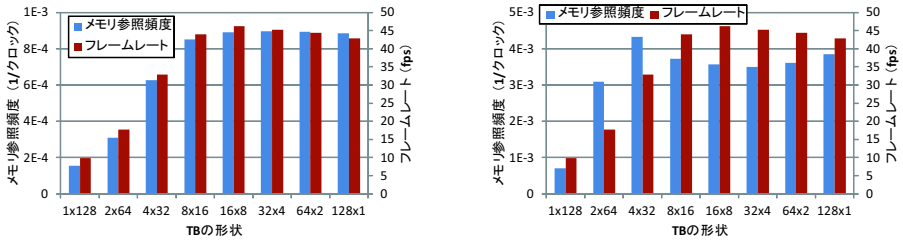
ただし，TC ヒット率については必ずしもメモリ参照頻度との相関があるといえない場合もある．視点 (0,0,0) と同様の実験を視点 (90,0,90) においても実施した．その結果を**図 8** および**図 9** に示す．

まず**図 8(b)** より単純手法を適用した場合の TC ヒット率とメモリ参照頻度を比較する．TB の形状を x 方向が長くなるよう変更していくことを考えた場合，8×16 および 16×8 の場合に TC ヒット率と比較してメモリ参照頻度の値が大きく減少しており，メモリ参照頻度の高さと TC ヒット率の高さが必ずしも対応しているとは言えない．また**図 9(b)** より FR



(a) 提案手法　　(b) 単純手法

図 6 視点 (0,0,0) におけるメモリ参照時間と TC ヒット率との関係

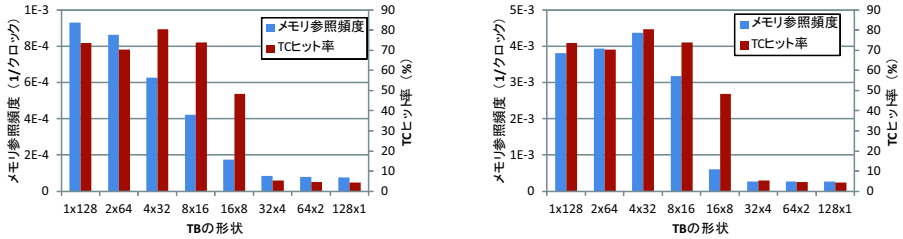


(a) 提案手法　　(b) 単純手法

図 7 視点 (0,0,0) におけるメモリ参照時間とフレームレートとの関係

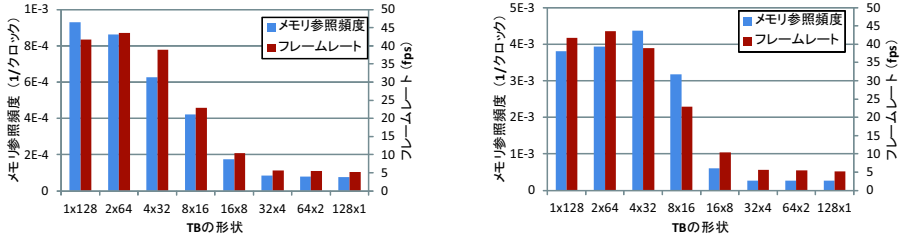
とメモリ参照頻度を比較した場合，まず TB の形状を 2×64 から 4×32 に変更した場合に，FR が減少する一方メモリ参照頻度は増加するという逆の関係が生じている．また，8×16 から 16×8 に変更した場合に，FR の減少幅と比較してメモリ参照頻度の減少幅が大きく，両者の傾向が一致しているとは言いにくい．

図 8(a) より提案手法を適用した場合，TB の形状が 4×32 の場合に，2×64 と比較してメモリ参照頻度は減少する一方 TC ヒット率は増加するという傾向の違いが見られる．また 4×32 から 8×16，16×8 へと変化させる場合には，いずれも減少しているものの，4×32 におけるずれは解消されておらず，全体として見ると TC ヒット率とメモリ参照頻度は必ずしも相関があるとは言えない．一方，**図 9(a)** より，128×1 から 2 × 64 への変化において



(a) 提案手法　　(b) 単純手法

図 8　視点 (90,0,90) におけるメモリ参照時間と TC ヒット率との関係



(a) 提案手法　　(b) 単純手法

図 9　視点 (90,0,90) におけるメモリ参照時間とフレームレートとの関係

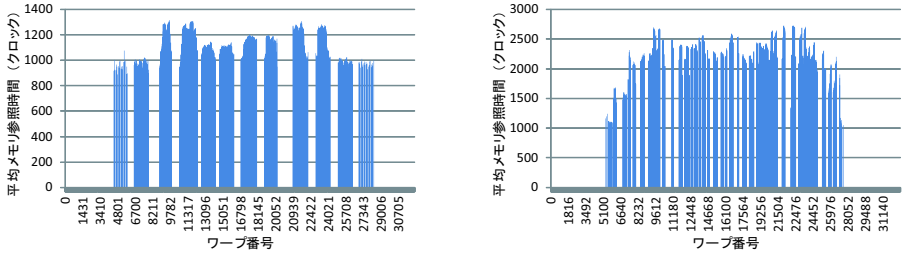
傾向の違いは見られるもののメモリ参照頻度と FR には概ね相関があると言える．

以上のことから，TC ヒット率については提案手法を適用した場合でもメモリ参照頻度と相関がないケースがあることがわかる．

6.4　メモリ参照時間の分布と性能の関係についての解析

6.3 節で述べた，視点 (90,0,90) においてメモリ参照時間と TC ヒット率の間で一部相関がない原因を特定するため，実行履歴を用いて VR のメモリ参照効率に関する解析をする．今回は 2×64 および 8×16 に着目する．図 8(a) より，これらの形状を比較した場合，TC ヒット率は同等である一方，2×64 においてメモリ参照がより高速であることがわかる．

また，TC ヒット率が同等でメモリ参照時間に差が出る原因は，キャッシュヒットが発生



(a) TB 形状 2×64　　(b) TB 形状 8×16

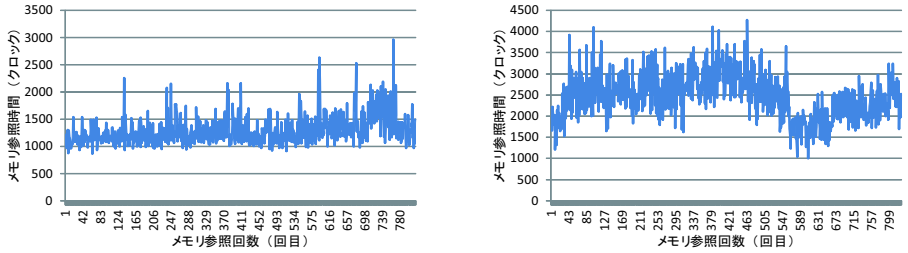
図 10　視点 (90,0,90) における各ワープの平均メモリ参照時間

するタイミングのばらつきが異なるためではないか，という推測に基づいて解析をすることとした．そこでまず実行全体において，2 つの TB 形状におけるメモリ参照時間の分布傾向に差があるかを調べた．

図 10 に，視点 (90,0,90)，TB 形状 2×64 および 8×16 で実行したとき，各ワープにおいて実行されたメモリ参照時間の平均値を示す．ワープの実行順序はワープ番号に概ね対応しており，メモリ参照時間の短いワープ，すなわちキャッシュヒットが多く発生しているワープの分布が確認できると考えたためである．なお，単一の SM における計測結果のみを示している．また，棒グラフが表示されていないワープは，メモリ参照をしていないことを示す．

まずワープ番号順に見たときに，いずれの TB 形状においてもメモリを参照しているワープが分布している領域の両端においてメモリ参照時間が短くなる傾向にあることがわかる．これは，ボリュームの境界においてメモリ参照をするワープとしないワープが混在し，全体のメモリ参照データ量が少ないためであると推測できる．その両端においては，いずれの TB 形状においてもメモリ参照時間は約 1000 クロック程度でほぼ同等である．しかし，その他の範囲では 2×64 の方が 8×16 と比較して全体的にメモリ参照時間が短い．さらに，8×16 の方が各ワープにおけるメモリ参照時間の分散が大きい傾向にある．メモリ参照時間の長いワープと短いワープで参照時間の差が大きく，かつそれらがばらばらに分布している．一方，2×64 では，メモリ参照時間がとる値の幅が小さく，またワープ番号が近いワープではメモリ参照時間も近い傾向が見られる．

以上のことから，TC ヒット率が同一でもメモリ参照効率の低い条件下では，各ワープのメモリ参照時間の分散が大きくなっていることがわかる．ただし，そのような傾向が見られ



(a) TB 形状 2 × 64　　(b) TB 形状 8 × 16

図 11　視点 (90,0,90) の単一ワープにおけるメモリ参照時間の推移

る原因は特定できていない．

次に，より詳細なレベルでの時系列に沿ったメモリ参照時間の分布を知るために，ある単一のワープに着目してメモリ参照時間の推移を調べた．着目するワープとしては，メモリ参照回数が多く，かつ平均のメモリ参照時間が長いものを選択した．メモリ参照時間のサンプル数が多く，かつメモリ参照効率が悪いと思われるワープの傾向を知るためである．

図 11(a) および**図 11(b)** に TB 形状 2×64 および 8×16 の単一ワープにおけるメモリ参照時間の推移を示す．横軸が何回目のメモリ参照であるかを示し，縦軸がそのときのメモリ参照時間を示している．これらを比較すると，単一ワープのレベルにおいても 8×16 の方がメモリ参照時間の分散が大きい様子が確認できる．

7. まとめと今後の課題

本研究では CUDA プログラムにおけるメモリ参照効率を解析することを目的に，メモリ参照時間を正確に計測できる実行履歴の生成手法を提案した．提案手法はメモリ参照を完了させたのちに，タイムスタンプを記録する．メモリ参照の完了を保証する手法としては，計測対象とするメモリ領域に応じてメモリフェンス命令を挿入する方式と，メモリ参照との間にデータ依存のあるダミー命令を挿入する方式を使い分ける．

評価実験として，メモリ参照が性能ボトルネックであるボリュームレンダリング（VR）を対象に実行履歴を生成し，メモリ参照時間を計測した．提案手法および単純手法で計測したメモリ参照時間を比較した結果，単純手法ではメモリ参照以外の処理が性能ボトルネックであると使用者に誤って判断させていたが，提案手法によりメモリ参照が性能ボトルネックであると正しく判断できた．また，VR のフレームレートとメモリ参照時間との間に強い相関があることを確認でき，単純手法を用いた計測よりもアプリケーションの特性を正しく捉えられることを確認できた．

今後の課題は，実行履歴に基づいてメモリ参照効率の低い点を検出する手法を考案することである．また，実行履歴を使用者が理解しやすく提示するための手法を提案することも必要である．

謝辞　本研究は，科学技術振興機構の戦略的創造研究推進事業 CREST における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」によるものである．

参 考 文 献

1) Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E. and Phillips, J.C.: GPU Computing, *Proceedings of the IEEE*, Vol.96, No.5, pp.879–899 (2008).
2) NVIDIA Corporation: CUDA Programming Guide Version 4.0 (2011).
3) NVIDIA Corporation: Compute Visual Profiler User Guide (2011).
4) Farooqui, N., Kerr, A., Diamos, G., Yalamanchili, S. and Schwan, K.: A framework for dynamically instrumenting GPU compute applications within GPU Ocelot, *Proc. 4th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)* (2011).
5) NVIDIA Corporation: PTX: Parallel Thread Execution ISA Version 2.1 (2010).
6) Malony, A.D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., Dietrich, R., Poole, D. and Lamb, C.: Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs, *Int'l Conf, Parallel Processing (ICPP 2011)* (2011).
7) Shende, S.S. and Malony, A.D.: The TAU Parallel Performance System, *Int. J. High Performance Computing Applications*, Vol.20, No.2, pp.287–311 (2006).
8) Zaki, O., Lusk, E., Gropp, W. and Swider, D.: Toward Scalable Performance Visualization with Jumpshot, *Int. J. High Performance Computing Applications*, Vol.13, No.2, pp.277–288 (online),
available from ⟨http://www-unix.mcs.anl.gov/perfvis/software/viewers/⟩ (1999).
9) NVIDIA Corporation: GPU Computing SDK (2012).