

CUDA カーネルの性能を解析するための 実行履歴生成ツール

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) プログラムの性能を解析するための実行履歴の生成手法を提案する。この手法は、統計的な履歴ではなく命令実行のタイミングに着目した履歴を生成する。履歴生成においては、本来の実行状況を適切に反映した履歴を得るために、外乱の発生を抑える必要がある。そこで、CUDA プログラムに適した履歴生成手法を明らかにすべく、記録先のメモリやその参照パターン等に注目して、複数の手法を実装した。実験では 8 つのアプリケーションを対象に、各手法適用時の実行時間の増加率を評価した。結果、共有メモリを用いる手法は増加率のばらつきが大きいことや分岐を用いてメモリ参照量を削減するよりも全スレッドで書き出す手法が有効な場合が多いことが確認できた。

A Log Generation Tool for Analyzing the Performance of CUDA Kernels

HIROTO KANDA,^{†1} TOMOHIRO OKUYAMA,^{†1}
FUMIHIKO INO^{†1} and KENICHI HAGIHARA^{†1}

This paper presents methods to generate logs for analyzing the performance of compute unified device architecture (CUDA) programs. Our methods generate logs which focus on the instruction execution timing in the CUDA programs. When we generate logs, we need to reduce the perturbation in order to get logs which properly reflect the original execution status. Then we implement multiple log generation methods to clarify the method which is best suited to CUDA programs. In experiments, we applied the methods to eight applications and measured the increase rate of execution time. As a result, the rate varies widely in the on-chip memory methods. We also find that the method which record events to the off-chip memory by all threads is often more effective than methods which use branch processes to reduce the off-chip memory access.

1. はじめに

近年、グラフィクス処理用のプロセッサである GPU (Graphics Processing Unit) を用いて汎用計算を高速化する研究が注目されている。GPU による汎用計算のための開発環境として CUDA¹⁾ (Compute Unified Device Architecture) がある。CUDA では GPU を SIMD²⁾ (Single Instruction Multiple Data) 型の並列計算機として扱い、カーネルと呼ばれる関数を多数のスレッドを生成して並列処理できる。ただし、GPU の性能を引き出すためにはメモリ参照パターンや実行スレッド数などに関するチューニングが必要となる。

プログラム内のチューニングすべき箇所を特定するためには性能解析が重要である。CUDA プログラムの性能を解析する既存のツールに CUDA プロファイラ³⁾ がある。このツールは、カーネル全体の実行時間やオフチップメモリの参照量などの統計的な情報を提供する。しかし、命令実行のタイミングに着目した情報は提供していない。したがって、開発者がカーネル内部の実行状況について得られる情報は限られており、チューニングすべき箇所の特定は容易でない。そのような開発者の負担を軽減するために、CUDA プログラムに対して実行履歴を利用した性能解析を導入することが有効である。例えば、性能ボトルネックとなっている処理の特定やスレッドの切替えによる命令遅延の隠蔽効果を確認するなどといった用途が想定される。タイミングに着目した履歴を生成する手法としては、対象プログラム中にイベントを記録する命令を挿入し実行する手法が一般的である。この手法では、記録命令の実行による実行時間の増加やスケジューリングの変化などの外乱が発生する。これらの外乱をできるだけ小さくしたい。

そこで、本研究では GPU プログラム上で外乱の小さい履歴生成を実現するための指針を明らかにすることを目的に、CUDA カーネルの実行履歴を生成する手法を提案する。提案手法として、イベントを記録するメモリ階層やその参照パターンおよび分岐の有無に関する特徴の組み合わせを変えた複数の手法を実装した。これらの特徴は、CUDA プログラムの性能を向上あるいは低下させる主な要因と考えられる点に着目したものである。提案手法を用いて実行履歴を生成するためには、まず履歴生成用のコードを対象の CUDA プログラム中に挿入する。そのプログラムを本研究で開発したライブラリとリンクさせて実行することで、実行履歴が得られる。生成した実行履歴は既存のツール Jumpshot⁴⁾ を用いて可視化で

^{†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

きる。

以降では、2章で関連研究を紹介し、3章でCUDAカーネルにおける実行履歴生成の概要を述べる。次に、4章で提案するイベント記録手法について述べ、5章で各提案手法を適用したアプリケーションを実行したときに発生する外乱について評価する。最後に、6章において本研究をまとめ、今後の課題を述べる。

2. 関連研究

Houら⁷⁾はGPU上でのデータの変更を追跡することを目的として、ソースコード中に履歴取得用のコードを挿入し、コンパイルおよび実行する手法を提案している。しかしながら、本研究はHouらと異なり、可能な限り外乱の小さい履歴生成手法を必要とする。

また、GPUをエミュレーションすることでプログラムの性能解析やデバッグを支援する試みも行われている。Collangeら⁸⁾はPTXを入力とするエミュレータを開発している。また、Bakhoda⁹⁾らもGPUのシミュレータを開発し、CUDAプログラムをシミュレータ上で実行することで、性能ボトルネックを解析している。

一方、並列環境向けの実行履歴生成ツールには、MPI (Message Passing Interface)⁵⁾ 向けのMPE (MPI Parallel Environment)⁶⁾がある。MPEはMPIプログラムを対象に、実行履歴の取得および履歴を解析する機能を提供する。MPEではソースコード中に履歴を取得するためのコードを埋め込み、実行履歴を生成する。ただし、MPEは一般的なCPU上で動作することを想定している。そのため、履歴の生成時にメモリの動的確保など、CUDAがGPU上での計算において提供していない機能を必要とする。

3. CUDAにおける実行履歴の生成

以下、ユーザが挿入した履歴生成用コードの実行により記録される情報をイベントと呼び、カーネル実行中に記録された情報全体の集合を指して実行履歴と呼ぶ。

3.1 履歴生成時に注意すべきCUDAの特徴

CUDAカーネルの実行履歴を生成する上で注意すべきCUDAの特徴について述べる。CUDAのアーキテクチャではGPUは内部に複数のマルチプロセッサ(MP)を持ち、各スレッドは複数のスレッドをまとめたスレッドブロック(TB)単位でMPに割り当てられ実行される。TBはさらにワープと呼ばれる単位に分割される。ワープとはCUDAにおけるスレッドの実行単位であり、現時点では32スレッドが1ワープを構成する。CUDAはSIMD型のアーキテクチャであり、同一ワープ内のスレッドは同じタイミングで同一命令を

実行する。分岐により同一ワープ内のスレッドが異なる処理をするとき、処理の効率が低下する。

次にメモリ階層について述べる。GPUのメモリ階層はおおまかにグローバルメモリおよび共有メモリに分類できる。グローバルメモリは全スレッドがアクセス可能であり大容量であるが、その参照命令は遅延が最も長い命令の1つである。一方、共有メモリはTBごとにデータを共有できるメモリ領域であり、参照遅延は短い小容量である。これらのメモリは参照パターンにより参照の効率が変動する。グローバルメモリでは、ワープの前半あるいは後半16スレッドが一定サイズの連続したメモリ領域を参照した場合、すべてのスレッドによる参照がまとめて処理され、離れた領域にあるデータを参照する場合と比べて効率が上がる。このようなパターンに従うメモリ参照をコアレスドアクセス(coalesced access)と呼ぶ。メモリ領域は32, 64および128バイトごとに分かれており、参照アドレスの範囲に合わせて自動的に最もサイズの小さい領域全体を使用しないデータも含めて参照する。共有メモリは内部が複数のバンクに分かれている。複数のスレッドが共有メモリを同時に参照する場合、それぞれが異なるバンクに含まれるアドレスを参照する場合は並列に処理できるが、同一バンク内のアドレスを参照する場合は各処理が逐次的になり、性能が低下する。この現象をバンクコンフリクトという。

また、共有メモリおよびレジスタはMP内で同時実行されるスレッドが共有する資源であり、スレッド当りの使用量が増加すると、MP上で同時実行可能なワープ数が減少しうる。同時実行が可能なワープ数の上限に対して、実際に同時実行されているワープ数の割合をMPの占有率(Occupancy)という。占有率が1に近いほど並列度が高いことを示す。

3.2 履歴生成に必要な処理

まず、本ツールにおけるCUDAカーネルの履歴生成の要件について述べる。本ツールではカーネルの実行中に履歴生成用のコードを実行した時間情報と各スレッドを実行しているMPの番号をイベントとして記録する。時間情報は履歴生成用コードを実行するたびに記録するが、MP番号はカーネルの開始時のみ記録する。ワープ内のスレッドは同一MP上で実行され、また同一のタイミングで命令を実行するため、イベントはワープ単位で記録する。実行履歴もワープごとに情報をまとめて記録するため、各イベントを記録したワープを識別できる必要がある。実行履歴はカーネル終了後にCPU側で生成する。

次に、これらの要件の実現に必要な処理を述べる。イベントに記録する時間情報はCUDAが提供するclock関数を用いて取得する。clock関数とはGPUの1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関数である。また、イベントを記録したワープはメモリ

上でのイベントの記録箇所から CPU 側で判断する。すなわち 1 つのワープに関するイベントは全て連続した領域、あるいは一定の間隔をおいた領域に記録することでイベントを記録したワープを識別できるようにする。そのため実行中にワープそのものを特定する情報は記録しない。記録したイベントはカーネル実行後に CPU 側に書き戻すため、GPU のグローバルメモリ上に記録する必要がある。そのため、対象とするアプリケーションの前提条件としてグローバルメモリ上に履歴を記録可能な未使用領域があるとす。

4. 提案手法

本章では、まず CUDA カーネル内でのイベント記録により発生する外乱を可能な限り小さくするために、イベント記録手法がとるべき方針を検討し、その方針を満たす手法を提案する。最後にその手法を用いた履歴生成の手順を述べる。

4.1 記録手法の指針

先に述べた CUDA の特徴を踏まえて、外乱を小さくするためにイベント記録手法が満たすべき方針として以下の 4 つを挙げる。

方針 1：グローバルメモリへの参照量の増加を抑制する グローバルメモリへの参照量が多い手法を用いると、参照遅延のためにイベント記録による外乱が大きくなる。なお以降では、メモリ参照量という言葉はグローバルメモリへの参照データ量を指す。

方針 2：MP 内の資源の消費の増加を抑制する 資源の消費量が増加すると占有率が低下し、並列度が下がることにより処理効率が低下する。そのため、資源の消費量が多い手法では、イベント記録による外乱が大きくなる。

方針 3：分岐処理を避ける CUDA は SIMD 型のアーキテクチャであるため、分岐処理を多用する手法はイベント記録による外乱を大きくする。

方針 4：共有メモリへの書き込みの衝突を避ける バンクコンフリクトが発生するため、共有メモリへの書き込みが衝突する場合はイベント記録で発生する外乱が大きくなる。

イベントを記録する手法は、4 つの方針をすべて満たすものが望ましい。しかし、これらの方針の間にはトレードオフの関係が成立するものがある。例えば、方針 1 を満たす手段として共有メモリを用いる手法が考えられるが、共有メモリの消費が増えることは方針 2 に反する。そのため、方針の一部のみを満たす手法を実装し、どの方針をより優先すべきかを検証する。

4.2 記録手法の分類

表 1 に本稿で提案するイベント記録手法の一覧を示す。これらの手法はいずれも 4.1 節で

表 1 各手法の特徴

	C1 (イベントの一時的な記録先)	C2 (イベントを記録するスレッド)	C3 (メモリのアクセスパターン)
手法 A	グローバルメモリ	ワープ内の全スレッド	連続領域
手法 B	グローバルメモリ	ワープ内の全スレッド	同一アドレス
手法 C	グローバルメモリ	ワープ内の 1 スレッドのみ	—
手法 D	共有メモリ	ワープ内の全スレッド	連続領域
手法 E	共有メモリ	ワープ内の全スレッド	同一アドレス
手法 F	共有メモリ	ワープ内の 1 スレッドのみ	—

挙げた 4 つの方針の一部のみを満たす。各手法を以下の特徴 C1～C3 について分類する。

- C1：イベントの一時的な記録先
- C2：イベントを記録するスレッド
- C3：メモリのアクセスパターン

以降では各特徴による分類と 4 つの方針との関係を述べ、6 つの手法がそれぞれ満たす方針および満たさない方針をまとめる。

まず、特徴 C1 に着目すると、各手法はグローバルメモリに記録する手法および共有メモリに記録する手法に分類できる。前者はイベントを記録するたびにグローバルメモリへ書き込む手法であり、メモリ参照量が増加するため方針 1 に反する。後者はイベントを取得した時点では共有メモリに一時的に記録しておき、最後にグローバルメモリにまとめて書き出す手法である。この手法はメモリ参照量を削減するが、共有メモリを消費するため方針 2 に反する。

次に、特徴 C2 に着目すると、各手法はワープ内の全スレッドがイベントを記録する手法およびワープ内の 1 スレッドのみがイベントを記録する手法の 2 つに分類できる。前者は分岐処理を用いないため方針 3 を満たす。しかし、メモリを参照するスレッド数が多いことにより、メモリ参照量あるいは共有メモリの消費量が増加する。一方で、後者の手法は分岐処理を必要とするため、方針 3 に反する。しかし、メモリを参照するスレッド数が少ないため、メモリ参照量あるいは共有メモリの消費量を抑えられる。

特徴 C3 に着目すると、ワープ内の全スレッドがイベントを記録する手法は、連続領域へ書き込む手法と同一アドレスへ書き込む手法に分類できる。ワープ内の 1 スレッドのみが記録する手法はアクセスパターンによる分類はない。連続領域へ書き込む手法は、ワープ内の全スレッドが重複なく連続した領域にイベントを記録する。一方、同一アドレスへ書き込む手法は、ワープ内の全スレッドが 1 つのイベントを同一のアドレスに記録する。複数

スレッドがメモリの同一アドレスに書き込みをする場合、少なくとも1つのスレッドによる書き込みの成功は保証されている。その他のスレッドによる書き込み内容は失われるが、イベントはワープ単位で記録するため、ワープ内のいずれかのスレッドがイベントを記録できればよい。メモリのアクセスパターンが性能に与える影響はグローバルメモリおよび共有メモリで異なるため、以下でそれぞれについて説明する。

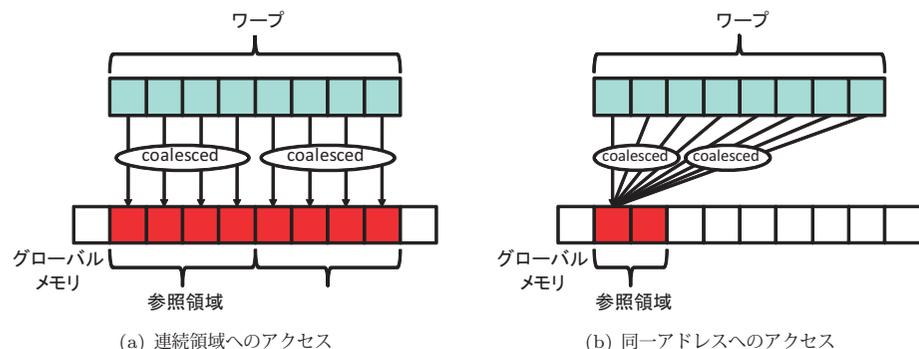


図1 グローバルメモリへのアクセスの様子

グローバルメモリに記録する手法では、連続領域および同一アドレスどちらに書き出す手法もコアレスドアクセスの条件を満たしている。図1にグローバルメモリの連続領域および同一アドレスへの参照の様子を示す。同一アドレスに書き出す方がより狭い領域へ参照が集中するため、メモリ参照量を抑えられる。

共有メモリに記録する手法では、連続領域に書き出すと共有メモリの消費量が増えるため、占有率が低下しやすい。同一アドレスに書き出すと共有メモリの消費量は抑えられるが、書き込みの衝突が発生するため方針4に反する。なお、共有メモリからグローバルメモリへのイベントの書き出しは、分岐処理を避けるためにイベント数に関わらずワープ内の全スレッドで行う。

最後に各手法と4つの方針の対応関係についてまとめる。まず手法Aはメモリ参照量が全手法中最も多く方針1に反する。しかし、共有メモリおよび分岐処理を使用しないため方針2, 3, 4を満たす。手法Bは手法Aのメモリ参照量を削減した手法といえる。その他の特徴は手法Aと同様である。手法Cは手法Bよりもさらにメモリ参照量を削減する代わり

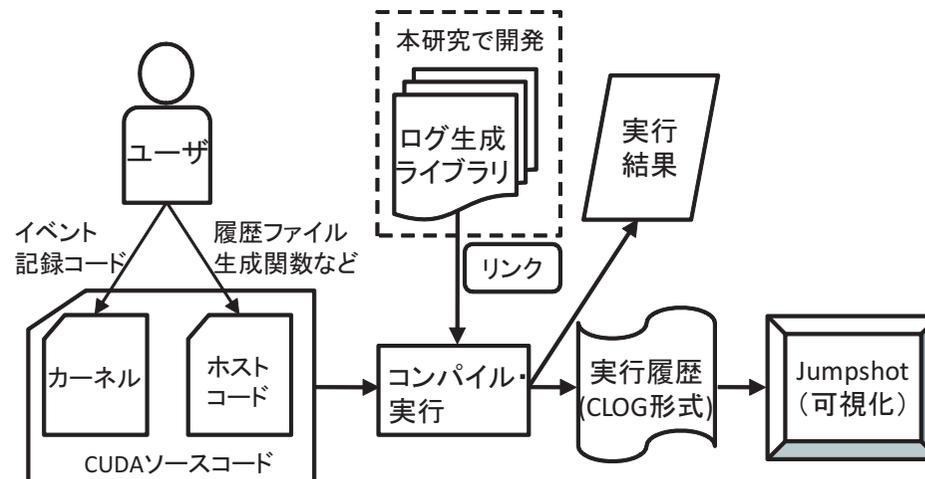


図2 履歴生成から可視化までの処理の流れ

に分岐処理を用いるため方針3に反する。手法Dは共有メモリを使用するため方針2に反するが、大幅にメモリ参照量を削減している。ただし、共有メモリからグローバルメモリへの書き出しは全スレッドで行うためイベント数が少ない場合は冗長な書き込みが発生する。また、共有メモリの消費量が他の共有メモリを用いる手法より多く、占有率が低下しやすい。手法Eは手法Dより共有メモリの消費量を削減するが、バンクコンフリクトが発生するため方針4に反する。手法Fは共有メモリの消費量は手法Eと同等だが、分岐処理を用いるため方針3に反する。

以上のように前節で挙げた全ての方針を満たす、あるいは他の手法より明らかに優れているといえる手法は今回提案する手法の中には存在せず、それぞれの手法が有効となる状況は異なると考えられる。

4.3 実行履歴を生成するまでの流れ

提案手法を用いて実行履歴を生成し、その実行履歴を確認するまでの手順について述べる。図2にその概要を示す。ユーザはCUDAカーネルの開始時と終了時およびその他カーネル中でイベントを記録したい任意の箇所にイベント記録用のコードを挿入する。そしてホストコードではカーネルの実行よりも前に対象となるCUDAカーネルの実行TB数、TBサイズ、記録イベント数をツールに対して指示する関数を呼び出す。また履歴の記録領域を

確保する関数および履歴のファイルを生成する関数を呼び出す。カーネルには引数として履歴を記録するメモリ領域を指す変数を追加する。その上で、本研究で開発したライブラリとリンクさせて実行することで履歴のファイルを得る。履歴のファイルは CLOG 形式⁴⁾であり、既存のツール Jumpshot を用い、ガントチャート形式で可視化できる。このチャートは縦軸がワープを、横軸が時間を示しており、どの時刻にどのワープがカーネルを実行しているかを確認できる。

次に、ツールが実行履歴のファイルを生成する処理について説明する。ユーザが指定した TB 数、TB サイズおよびイベント数を基にして、履歴を記録するためのメモリ領域をデバイスメモリ上に確保する。カーネル内では、ユーザがコードを追加した箇所で、前節で述べた各手法を用いてイベントを記録する。カーネルの開始時と終了時に記録するイベントは、記録回数が少ないことおよび記録位置が処理の途中ではないことから発生する外乱は小さいと考え、直接グローバルメモリに記録する。履歴のファイルを生成する関数は、デバイスメモリ上に記録された履歴をホストメモリにコピーし、それを基にして履歴のファイルを生成する。

5. 評価実験

本章では、提案した各手法を履歴生成時の外乱の大きさに関して評価する。そのためにアプリケーションのカーネル内に各手法を適用し、イベントを記録した。ここでは履歴生成による実行時間の増加が大きいほど外乱が大きいとみなす。すなわち、外乱 δ は以下のように定義する。

$$\delta = |T_1 - T_2| \quad (1)$$

ここで T_1 はイベントを記録しない場合の実行時間であり、 T_2 はイベントを記録する場合の実行時間である。

本章ではまず、各アプリケーションの特徴について述べ、各々に提案手法を適用した結果について述べる。最後に各手法の有用性について考察する。

5.1 実験方法

実験の対象として、8つのアプリケーションを用いた。その一覧を表2に示す。行列積、行列転置および FWT (Fast Walsh Transform) は CUDA SDK¹⁰⁾ に含まれているアプリケーションである。そして HS (Hot Spot), SRAD (Speckle Reducing Anisotropic Diffusion), BP (Back Propagation), KM (K-means), CFD (Computational Fluid Dynamics) の5つは Rodinia ベンチマーク¹¹⁾ に含まれている。また表2に各アプリケー

表2 実験対象アプリケーションの一覧

	グリッド サイズ	TB サイズ	レジスタ 消費率	共有メモリ 消費率	占有率	命令数 (1 ワープ)	記録 イベント数
行列積 ¹⁰⁾	(64,64,1)	(16,16,1)	0.875	0.625	1	3204	192
行列転置 ¹⁰⁾	(64,64,1)	(16,16,1)	0.5	0.375	1	49	3
FWT ¹⁰⁾	(8192,1,1)	(512,1,1)	0.25	0.531	0.5	295	16
HS ¹¹⁾	(43,43,1)	(16,16,1)	1	0.438	0.5	172	10
BP ¹¹⁾	(1,4096,1)	(16,16,1)	0.875	0.375	1	310	17
SRAD ¹¹⁾	(32,32,1)	(16,16,1)	0.625	0.813	0.5	270	15
KM ¹¹⁾	(29,29,1)	(256,1,1)	0.75	0.625	1	3800	200
CFD ¹¹⁾	(506,1,1)	(192,1,1)	0.938	0.0625	0.375	783	44

ションの持つ特徴を示す。レジスタ消費率 R_r および共有メモリ消費率 R_s は $R_r = r/r_{max}$ および $R_s = s/s_{max}$ である。ここで r_{max} および s_{max} はそれぞれ MP が持つレジスタ数および共有メモリの容量を示し、 r および s はそれぞれ同時実行されているスレッドが実際に使用しているレジスタ数および共有メモリ容量の割合を示す。

これらのアプリケーションのカーネル内で T_1 および各手法を適用したときの T_2 をそれぞれ計測し、イベントの記録による実行時間の増加率 $R = 100\delta/T_1$ を調べた δ の大きさと δ が各カーネルの実行状況に与える影響の大きさは必ずしも対応しないと考えられる。例えば、 δ の値が等しくても、元のカーネルの実行時間が異なれば δ の意味合いも変化する。そこで、 R を用いて各カーネルに δ が与える影響の大きさを評価する。また、この実験では各手法間の優劣の評価が主な目的であるため、各カーネルの処理に対してイベント記録の処理が占める割合を均一にしたいと考えた。そこで、イベント数は各カーネルの命令数におよそ比例する値に設定した。ただしここで言うイベント数には開始時と終了時のイベントは含んでいない。実験環境は CPU が Intel Xeon W3250 (2.67GHz), GPU が NVIDIA GeForce GTX 285, CUDA バージョン 2.3 を用いた。

5.2 履歴生成により発生する外乱の評価

図3に各アプリケーションにおけるイベント記録時の増加率 R を示す。

また、表3に各手法を適用したときの占有率を示す。HS, BP, CFDにおける占有率の低下はレジスタ使用数の増加による。その他のアプリケーションにおける低下は共有メモリ消費量の増加に起因する。

各アプリケーションにおける手法間の優劣について考察する。まず全体の傾向として多くのアプリケーションで手法 E の R が最も大きい。BP では R が次に大きい手法 C と比較

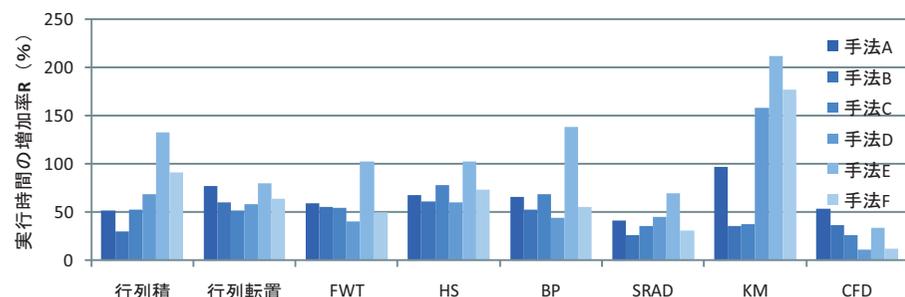


図 3 各アプリケーションにツールを適用した場合のオーバーヘッド

表 3 ツール適用後の占有率

	手法 A	手法 B	手法 C	手法 D	手法 E	手法 F
行列積	1	1	1	0.25	0.25	0.25
行列転置	1	1	1	1	1	1
FWT	0.5	0.5	0.5	0.5	0.5	0.5
HS	0.25	0.25	0.25	0.25	0.25	0.25
BP	0.75	1	1	0.75	0.75	1
SRAD	0.5	0.5	0.5	0.25	0.5	0.5
KM	1	1	1	0.25	0.25	0.25
CFD	0.188	0.188	0.188	0.188	0.188	0.188

して約 2 倍の値を示している。このことからバンクコンフリクトが発生する手法では性能の低下が著しいとわかる。また、手法 A も R が大きく、少なくとも同じグローバルメモリに記録する手法 B および C の両方に対して優位なケースはない。これはメモリ参照量が他手法よりも多いことによると考えられる。

次にイベントを共有メモリに記録する手法の効果について確認する。行列積や KM では共有メモリを用いる手法の R がいずれも大きい。これはイベント数が多いために共有メモリ消費量が増大し、占有率が 1 から 0.25 へ低下していることが原因と考えられる。また SRAD では手法 D の R が手法 E に次いで大きく、これも占有率が 0.25 へ低下したことによると考えられる。これらのアプリケーションにおいてこの数字は TB が同時に 1 つしか実行されていない状態を示し、TB 内同期の遅延が隠蔽されないなど性能が低下しやすい。一方、BP では手法 D で唯一占有率が低下しているが、 R が最小という結果を示している。これ

は行列積などと異なり低下後も 0.75 と十分な同時実行ワーブ数を確保できているため、性能低下を抑えられていると考えられる。FWT、HS および CFD も手法 D の R が最も小さい。これらのアプリケーションでは、共有メモリ消費量の増加による占有率の低下がない。手法 D は、占有率が低下しやすい代わりにメモリ参照量および分岐処理を削減した手法である。そのため、この場合は手法 D の利点のみが活かされる。結果、イベント記録による性能の低下を抑えられたものと思われる。CFD は手法 D だけではなく、共有メモリを用いる手法が全体的に優れた結果を残している。これはイベント数が 44 と FWT や HS と比較して多いことにより、メモリ参照量削減の効果が高まったものと思われる。行列転置も FWT などと同様に全手法で占有率は等しいが、手法 C の R が最小である。行列転置はイベント数がワーブあたり 3 と最も少ない。そのため共有メモリの使用によるメモリ参照量削減の効果が得られず、グローバルメモリに記録する手法ではメモリ参照量の最も少ない手法 C が優位になったと思われる。

最後に、分岐を用いる手法の有用性を確認する。行列積、HS、BP、SRAD および KM では手法 B が手法 C よりも R が小さい。これらのアプリケーションではメモリ参照量を削減した効果よりも分岐を用いたことによる性能の低下が大きいといえる。逆にこれら以外のアプリケーションではメモリ参照量を削減した効果が大きいと考えられる。このようにアプリケーションによる優劣の違いが生じる原因は特定できていない。

5.3 適切な手法の選択方法についての考察

実験結果から、対象とするアプリケーションにより有効な手法が異なるため、動的に外乱の少ないイベント記録手法を選択する機構を検討する。

まず、ほぼすべてのアプリケーションにおいて手法 E の R が最大であることから方針 4 を優先して満たすべきであるといえる。また同様に手法 A も R が大きく、今回の実験ではこれを選択すべきケースはなかった。これらのことから、この 2 手法は積極的に選択しない。

以降は、手法 B、C、D、F の選択基準について考察する。まず、占有率が低下した手法はそうでない手法と比較して、メモリ参照量の大小あるいは分岐処理の有無に関わらず R が大きい傾向がある。例外として BP が挙げられるが、このような例外が生じる条件は明らかになっていない。そのためここでは占有率が他手法より低下した手法は選択しないこととする。

各手法で占有率が等しい場合は手法 D を選択する。ただしイベント数はある閾値 E_D 以上大きい場合に限る。 E_D の具体的な値については検討できていないが、メモリ参照量の差や実験に基づく事前予測は可能と考えられる。 E_D を下回った場合は手法 B あるいは C を

選択する。この2手法の優劣が決定する条件は現在のところ明らかでないため、厳密には両方実行して確認する必要がある。しかし、今回の実験で手法Bが優位である場合が多かったため、厳密さを求めないならば手法Bを選択すればよいと考える。手法Fはこの場合、分岐処理のコストがあり、手法Dよりも常に性能の低下が大きいと考えられるため、選択されない。しかし、手法Fも手法D同様イベント数がある閾値 E_F 以上であれば手法BおよびCよりも優位になると考えられる。そのため、手法Dで占有率が低下している場合などは手法Fを選択することもありうる。ただし、分岐処理のコストがあるため、 $E_F > E_D$ と考えられる。

まとめると、占有率が等しくイベント数がある程度多い場合は共有メモリを用いる手法、中でも手法Dを優先して選択し、イベント数が少ない場合は手法BあるいはCを選択する。ただし、占有率が他手法より低い手法については選択しない。以上が今回の実験結果に基づくイベント記録手法の選択方針である。

6. まとめ

本稿では、CUDAプログラムの性能を解析するために、CUDAカーネル内でイベントを記録し実行履歴を生成するツールを提案した。イベントの記録により発生する外乱を可能な限り小さくするため、記録するメモリ階層、アクセスパターンおよび分岐の有無という3点に着目して6つの手法を検討した。

評価実験では、8つのアプリケーションに各手法を適用してイベントを記録し、対象アプリケーションによる各手法の優劣を評価した。結果として共有メモリ消費率が低い場合は共有メモリを用いる手法が有効であるが、イベント数が少ない場合はその利点が生かせないことがわかった。イベント数が非常に多い場合には占有率の低下により大幅に実行時間が増加するなど性能のばらつきが大きかった。また、メモリ参照量が増加しても、分岐を用いる手法よりワープ内の全スレッドでイベントを記録する手法が有効な場合が多い。さらに共有メモリへの書き込みの衝突は優先して避けるべきであることも確認した。これらの結果を踏まえて対象アプリケーションに合わせて適切な手法を選択する方針を考察した。

今後の課題は複数手法の存在をユーザに対しては隠蔽しつつ、適切な手法を動的に選択する機構を実装することである。

謝辞 本研究の一部は、科学研究費補助金基盤研究(A)(2024002)および大阪大学グローバルCOEプログラム「予測医学基盤」の補助による。

参考文献

- 1) NVIDIA Corporation: CUDA Programming Guide Version 2.3 (2009). <http://developer.nvidia.com/cuda/>.
- 2) Grama, A., Gupta, A., Karypis, G. and Kumar, V.: *Introduction to Parallel Computing*, Addison-Wesley, Reading, MA, second edition (2003).
- 3) NVIDIA Corporation: CUDA Visual Profiler User Manual (2009).
- 4) 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 (1999).
- 5) Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, *Int'l J. Supercomputer Applications and High Performance Computing*, Vol. 8, No.3/4, pp.159–416 (1994).
- 6) Chan, A., Gropp, W., and Lusk, E.: User's Guide for MPE: Extensions for MPI Programs. <http://www.mcs.anl.gov/research/projects/mpi/mpich1/docs/mpeman/mpeman.htm>.
- 7) Hou, Q., Zhou, K. and Guo, B.: Debugging GPU Stream Programs Through Automatic Dataflow Recording and Visualization, *Proc. SIGGRAPH Asia 2009*, pp. 1–11 (2009).
- 8) Collange, S., Defour, D. and Zhang, Y.: Dynamic detection of uniform and affine vectors in GPGPU computations, *Europar 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, pp.1–10 (2009).
- 9) Bakhoda, A., Yuan, G.L., Fung, W. W.L., Wong, H. and Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator, *IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS 2009)*, pp.163–174 (2009).
- 10) NVIDIA Corporation: NVIDIA CUDA SDK Code Samples (2010). http://developer.nvidia.com/object/cuda_sdk_samples.html.
- 11) Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H. and Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing, *IEEE Int'l Symp. Workload Characterization (IISWC 2009)*, pp.44–54 (2009).