Vol.2017-HPC-160 No.4
2017/7/26

GPUサイクル共有を自動化するための タスク粒度推定手法

塚田 敬司[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，GPU（Graphics Processing Unit）サイクル共有システムにおけるゲストの労力を削減するために，タスク粒度を自動的に推定する手法を提案する．サイクル共有システムでは，計算資源提供者（ホスト）のフレームレート維持および科学計算投入者（ゲスト）の計算高速化を両立する必要がある．これらはゲストタスクの粒度を調節することで両立できる．しかし，既存の協調マルチタスク手法は手動による調節を必要とする．この手動の調節を回避するために，提案手法はホストの遊休時間を計測し，適切なタスク粒度を自動的に推定する．一般に，ホストプログラムを変更することなく遊休時間を計測することは難しい．そこで，ホストタスクの実行時間を間接的に計測することで，遊休時間を推定する．実験では，画像フィルタ（ホストタスク）のフレームレートを維持しつつ行列計算（ゲストタスク）の実効性能の低下を5%に抑え，適切なタスク粒度を自動的に推定できた．

キーワード：協調マルチタスク，ボランティア計算，遊休検出，GPU

A Method for Estimating Task Granularity for Automating GPU Cycle Sharing

Keishi Tsukada[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: In this paper, we propose a method for estimating task granularity to reduce guest's effort in graphics processing unit (GPU) cycle sharing systems. A cycle sharing system should maximize both the frame rate for resource donators (i.e., hosts) and the acceleration effect for scientific job submitters (i.e., guests). This can be realized by selecting the appropriate granularity for guest tasks. However, a previous cooperative multitasking method requires manual interactions to find the appropriate granularity. To avoid such interactions, the proposed method automatically estimates the appropriate task granularity by measuring the length of idle periods on the host. In general, it is not easy to measure the length of idle periods without changing the host program. Accordingly, we realize this estimation by indirectly measuring the execution time of host tasks. In experiments, the proposed method automatically estimated the appropriate task granularity, which not only maintained the frame rate for an image filter (i.e., host task) but also minimized the performance drop of matrix multiplication (i.e., guest task) within 5%.

Keywords: cooperative multitasking，volunteer computing，idle detection，GPU

1. はじめに

サイクル共有システムとは，科学計算の高速化を目的としてネットワーク上の計算資源を共有するボランティア型計算システムのことである．このシステムでは，計算資源を提供する側をホストと呼び，ホストの計算資源を用いて科学計算を高速化する側をゲストと呼ぶ．また，計算資源上で実行されるプログラムのインスタンスをタスクと呼び，ホスト側およびゲスト側のタスクをそれぞれホストタスクおよびゲストタスクと呼ぶ．計算資源上では，ホストタスクおよびゲストタスクが同時に実行されうる．

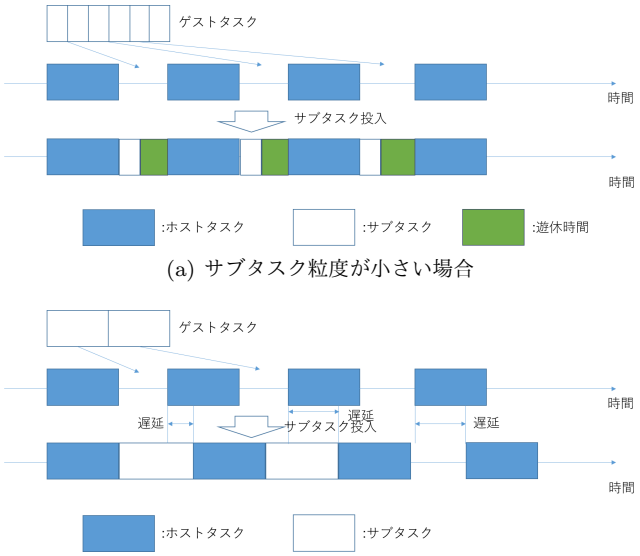
[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

サイクル共有システムを加速できる計算資源として Graphics Processing Unit （GPU）が着目されている [1]．GPU を汎用加速器として用いる場合，典型的なホストタスクは画面の描画処理であり，ゲストタスクは Compute Unified Device Architecture（CUDA）[2] により記述された科学計算である．ゲストタスクをホストの GPU 上で安易に実行した場合，GPU 向けプリエンプションを初めて実現した Pascal アーキテクチャ [3] においても，ゲストタスクが GPU を専有し，ホストタスクの実行を妨げてしまう．結果，ホストタスクの 1 秒間あたりの画面更新回数，すなわちフレームレート F が低下する．この低下の度合は，ゲストタスクの実行時間とともに大きくなり，最終的には 1 fps（frames per second）を下回る．したがって，F を維持できるゲストタスクの実行手法が必要である．

そのような手法として，既存手法 [4] はソフトウェア制御による協調マルチタスク手法 [5] に基づいて，1 回の実行を 10 ミリ秒程度で完了できるようにゲストタスクを分割している．以降，分割後のゲストタスクをサブタスクと呼ぶ．ゲストタスクの分割により，GPU は一定時間ごとにホストタスクを実行でき，フレームレート F を維持できる．ただし，ゲストタスクの実効性能とホストタスクのフレームレートはトレードオフの関係にあるため，ホストタスクに外乱を与えない範囲でゲストタスクの実効性能を最大化できる適切な分割数を探す必要がある．分割数が多い（サブタスク粒度が小さい）場合（**図 1(a)**），利用できる遊休時間に対してサブタスクが小さく，科学計算の高速化が十分でない．一方，分割数が少ない（サブタスク粒度が大きい）場合（**図 1(b)**），遊休時間内にサブタスクを完了できず，F を低下させてしまう．適切な分割数を探すためには，ゲストタスクの実行を繰り返す必要があり，ゲストの労力が大きい．

そこで，本論文ではゲストの労力を軽減するために，適切なサブタスク粒度を自動的に推定する手法を提案する．この実現のために，提案手法はホストタスクの実行時間 H を基に，ゲストタスクのために利用できる遊休時間の長さを推定する．一般に，サイクル共有システムでは，ホストのプログラムは入手できないため，ホストの立場から H を直接計測することは難しい．そこで，提案手法はゲストの立場から H を間接的に推定する．具体的には，実行直後に処理を完了するカーネル（空カーネル）を用い，小さなオーバヘッドで H を推定する．空カーネルはホストタスクが実行中でなければただちに処理を完了できるが，実行中であれば，その処理はホストタスクの完了まで遅延する．したがって，空カーネルを反復実行すれば，H を推定できる．

提案手法は，以下の 2 点を前提としている．(1) ホストのフレームレート F が専有実行時に一定であること，および (2) ゲストの科学計算が CUDA で記述されていること．



(a) サブタスク粒度が小さい場合

(b) サブタスク粒度が大きい場合

図 1 タスク分割による協調マルチタスク手法

以降では，2 章で既存手法 [4] をまとめる．その後，3 章で提案手法を示し，4 章で評価実験の結果を示す．5 章で関連研究を紹介する．最後に，6 章で本稿をまとめる．

2. 既存の協調マルチタスク手法

既存手法 [4] は，ホストタスクが一定のフレームレート F で画面を描画するものと仮定している．ホストタスクが 1 回の画面更新に費やす時間 H を用いれば，ゲストタスクが利用できる遊休時間 T は式 (1) のように表せる．

$$T = 1/F - H \tag{1}$$

ここで，サブタスクの実行時間を S とすると，フレームレートを維持するための条件は $S < T$ となる．

2.1 遊休資源の検出

遊休状態のホストを検出するために，既存手法は空カーネルを用いる．あらかじめ遊休時に空カーネルを実行し，専有時におけるその実行時間 K を計測しておく．具体的には，空カーネルを 1 秒間に渡って複数回実行し，それらの最大値を K とする．運用時には，サブタスクの実行直前に空カーネルを実行し，そのときの空カーネルの実行時間 E を K と比較することで，ホストの状態を判断する．R (> 1) 回連続して $K > E$ を満たした場合，ホストを遊休とみなし，そうでない場合，繁忙とみなす．ここで，R は検出の安定性を高めるためのパラメータであり，実験的に定める．$R = 1$ の場合，頻繁に遊休状態を検出する可能性があり，ゲストタスクの過度な実行が原因でホストに外乱を与えてしまう．

2.2 ゲストタスクの分割

一般に，スレッド間のデータ依存が原因で，並列に動作す

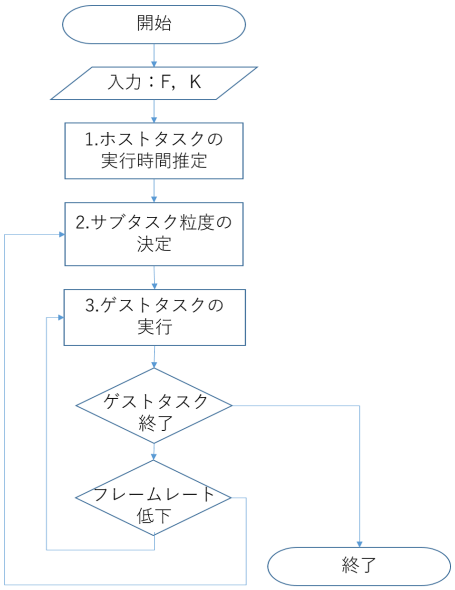


図 2　提案手法における実行の流れ

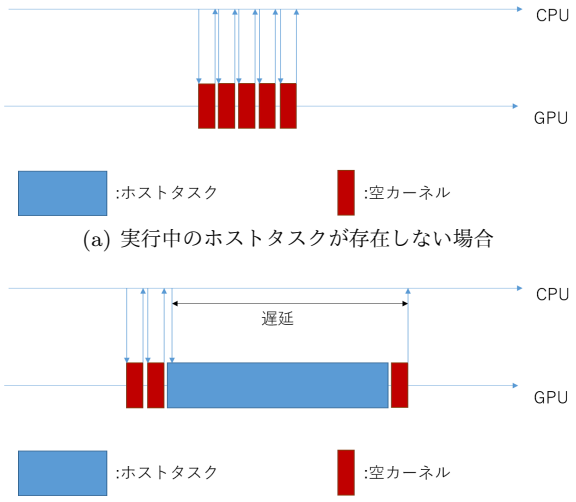


図 3　空カーネルによるホストタスクの実行時間推定

るタスクを分割することは容易ではない．しかし，CUDA ではデータ依存に関してスレッド間に制約があり，異なるブロックに所属するスレッド間にデータ依存を許さない．したがって，ブロック単位で自由にスレッドを間引けば，1 回あたりの実行時間を制御できる．そこで，既存手法は 1 度に生成するスレッドブロックの数を削減することで，ゲストタスクの分割を実現している．ただし，この削減は GPU プログラムの呼び出し回数だけでなく，メモリ参照量も増大させるため，分割に伴うオーバヘッドが生じる．

2.3　ゲストタスクの実行

既存手法は，遊休状態だけでなく繁忙状態の計算資源においてもゲストタスクを実行する．ただし，前者と後者で 2 種類の実行モードを使い分けている．ホストが繁忙状態ならば間欠実行モードを用い，遊休状態ならば連続実行モードを用いる．間欠実行モードは，サブタスクを周期 F で間欠実行し，ホストへの負荷を軽減する．一方，連続実行モードはサブタスクを連続実行し，ゲストが投入する科学計算の実効性能を向上する．

3.　提案手法

提案手法は，n 個のスレッドブロックからなるゲストタスク $G = \{t_1, t_2, ..., t_n\}$ およびフレームレート F が与えられたときに，F を維持できる範囲でサブタスク $A \subseteq G$ の粒度 $|A|$ を最大化する．ここで，F はホストがあらかじめ申告する値である．

図 2 に，提案手法における実行の流れを示す．提案手法はまず（1）ホストタスクの実行時間 H を推定し，与えられた F に対する遊休時間 $T = 1/F - H$ を得る．その後，（2）時間 T 内に完了できる最大のサブタスク粒度 $|A|$ を推定し，（3）$|A|$ 個のスレッドブロックでサブタスクを実行する．

3.1　ホストタスクの実行時間推定

提案手法は，空カーネルを連続的に実行してホストタスクの実行時間 H を推定する（**図 3**）．実行中のホストタスクが存在しない場合，空カーネルは呼び出し直後に処理を終える．一方，実行中のホストタスクが存在する場合，ホストタスクの処理を終えたのちに，空カーネルの処理が開始される．このとき，空カーネルの実行時間 E は $E = H + K$ で与えられる．したがって，一連の連続実行において空カーネルの実行時間 E をそれぞれ計測しておき，$E > K$ のときに実行中のホストタスクが存在するものとみなし，$H = E - K$ とする．

なお，GPU は CPU と非同期で動作しているため，実行時間 E の計測時には，空カーネルを `cudaDeviceSynchronize()` で完了させたうえでタイマを停止する必要がある．タイマには，高精度の時刻を取得できる `QueryPerformanceCounter()` を用いている．

Algorithm 1 に，遊休時間 T を推定するアルゴリズムを示す．なお，このアルゴリズムでは，フレームの更新 1 回あたり複数のホストタスクの処理が必要な場合を考慮している．具体的には，フレームを 1 回更新するまでに，遅延の生じた空カーネルをすべて検出し，それらの実行時間の和を計算している（8 行目）．

提案する推定手法はホストへの外乱を伴う．例えば，ホスト上で空カーネルを実行しているときに，ホストタスクの実行が求められた場合，ホストタスクが遅延してしまう．ただし，この遅延は高々 K であり，今回の実験環境では数十マイクロ秒程度であった．したがって，この推定手法がホストに与えるオーバヘッドは小さなものと考えている．

Algorithm 1 遊休時間の推定アルゴリズム

Input: ホストのフレームレート F および専有時における空カーネルの実行時間 K
Output: ホストの遊休時間 T

$H = 0$;
while 1 フレーム経過していない **do**
  タイマ開始;
  空カーネルを実行;
  タイマ停止;
  $E =$ 空カーネル実行時間;
  if $E > K$ **then**
    $H = H + E$;
  end if
end while
$T = 1/F - H$;

Algorithm 2 サブタスク粒度の決定アルゴリズム

Input: ホストの遊休時間 T
Output: スレッドブロック数 $|A|$

$|A| = 1$;
$S = 0$;
while $T > S$ **do**
  タイマ開始;
  $|A|$ 個のスレッドブロックでカーネル実行;
  タイマ停止;
  $S =$ カーネル実行時間;
  $|A| = \lfloor T/(S/|A|) \rfloor$;
  Sleep(T);
end while
$|A| = |A| - 1$;

Algorithm 3 ゲストタスクの実行アルゴリズム

Input: ホストの遊休時間 T
Output: スレッドブロック数 $|A|$

$S = 0$;
while サブタスクが残っている **do**
  タイマ開始;
  スレッドブロック数 $|A|$ でカーネル実行;
  タイマ停止;
  $S =$ カーネル実行時間;
  if $S > T$ **then**
    $|A| = |A| - 1$
  end if
  Sleep(T);
end while

表 1 実験に用いたプログラム

プログラム	汎用計算	描画処理	用途	精度
行列積 [2]	あり	なし	ゲスト	単精度
ヤコビ法	あり	なし	ゲスト	倍精度
レンダラ	なし	あり	ホスト	単精度
バイラテラル フィルタ [2]	あり	あり	ホスト	単精度

表 2 実験に用いた実行環境

項目	仕様
CPU	Intel Core i7 5930K（3.5 GHz）
GPU（環境 P）	NVIDIA GeForce GTX 1080
GPU（環境 M）	NVIDIA GeForce GTX 780ti
OS	Windows 10 Pro
CUDA バージョン	8.0
ドライババージョン	382.53

3.2 サブタスク粒度の決定

Algorithm 2 に，サブタスク粒度 $|A|$ を決定するアルゴリズムを示す．このアルゴリズムは，Algorithm 1 で得たホストの遊休時間 T を入力として，T 内に処理を完了できる最大のサブタスク粒度，すなわちスレッドブロック数 $|A|$ を返す．

まず，$|A| = 1$ を初期値として，1 個のスレッドブロックでサブタスクを実行し，その実行時間 S を計測する．$S \leq T$ である限り，このサブタスクは遊休時間 T 内に処理を完了でき，ホストタスクへの外乱はない．この場合，T 内に処理を完了できる最大のサブタスク粒度は $|A| = \lfloor T/(S/|A|) \rfloor$ で与えられる．ここで，$S/|A|$ はスレッドブロック 1 個あたりの実行時間を表す．この手順を反復していけば，サブタスク粒度 $|A|$ は増大していき，そのうち $S > T$ となり 3 行目の while ループを抜ける．このときの $|A|$ は，そのままでは時間 T 内に処理できないため，最後に $|A| - 1$ を最大のサブタスク粒度として出力とする．

このように，提案手法はサブタスク粒度 $|A|$ を一度に推定するのではなくて，漸近的に推定する．したがって，$|A|$ が一定の値に落ち着くまでに推定のためのオーバヘッドが生じる．

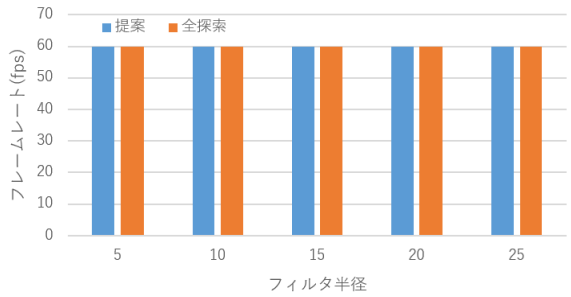
3.3 ゲストタスクの実行

Algorithm 3 に，ゲストタスクを実行するアルゴリズムを示す．3.1 節で推定したサブタスク粒度 $|A|$ を用いて，残りのサブタスクを実行する．ただし，ホストへの外乱を最小化するために，提案手法はスレッドブロックの数 $|A|$ を更新しながらサブタスクを実行していく．$S > T$ を検出した場合，$|A|$ を削減してゲストタスクを実行する．

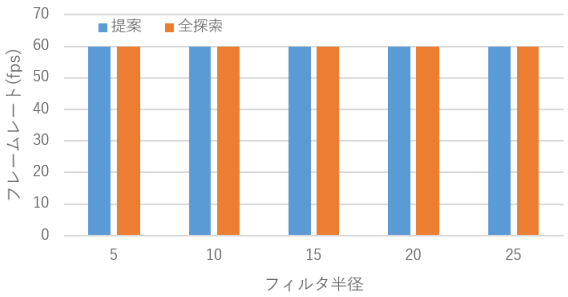
4. 評価実験

提案手法を評価するために，その推定のもっともらしさを全探索に基づく手法と比較する．全探索手法は，スレッドブロックの数 $|A|$ を変えながらホストタスクおよびゲストタスクを同時に実行し，これらのうち，フレームレート F を維持でき，かつゲストタスクの実行時間を最小化する値を $|A|$ として用いる．

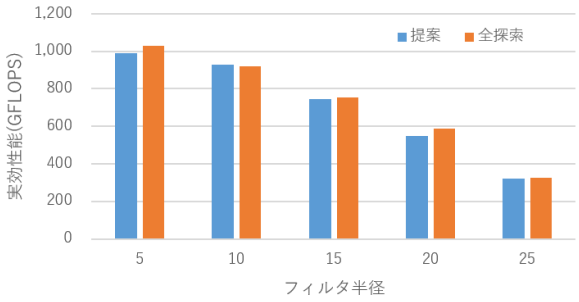
評価に用いたプログラムに関しては，ホストタスクとしてバイラテラルフィルタ [2] およびレンダラを用意し，ゲストタスクとして行列積 [2] およびヤコビ法を用意した（**表 1**）．バイラテラルフィルタは輪郭保持平滑化フィルタ画像を描画し，レンダラはティーポットモデルを描画する．いずれも描画処理ライブラリとして OpenGL[6] を用い，汎



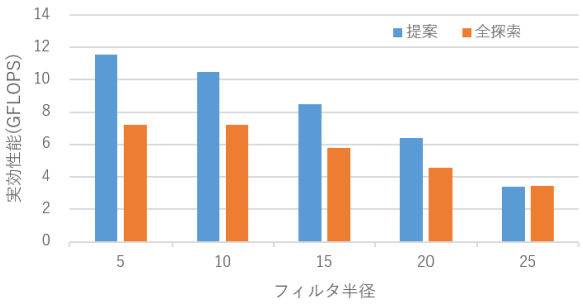
(a) フィルタのフレームレート（行列積実行時）



(b) フィルタのフレームレート（ヤコビ法実行時）



(c) 行列積の実効性能（フィルタ実行時）



(d) ヤコビ法の実効性能（フィルタ実行時）

図 4 マルチタスク実行時におけるゲストタスクおよびホストタスクの性能

用計算の開発環境として CUDA 8.0[2] を用いた．なお，フレームレートの計測には，Fraps 3.5.99[7] を用いた．

表 2 に実験環境を示す．プリエンプション機能のある Pascal アーキテクチャを持つ環境 P に加え，プリエンプション機能のない Maxwell アーキテクチャを持つ環境 M を用意した．

4.1 サブタスク粒度の推定

提案手法の推定結果を評価するために，全探索手法により得たサブタスク粒度 $|A|$ を提案手法のものと比較した．評価には，環境 P を用い，ゲストタスクとして行列積およびヤコビ法，ホストタスクとしてバイラテラルフィルタを用いた．なお，適切なサブタスク粒度はホストタスクの実行時間 H に依存するため，バイラテラルフィルタのフィルタサイズごとにサブタスク粒度を推定した．

表 3 より，行列積をゲストタスクとする場合，提案手法は全探索手法とほぼ同様のスレッドブロック数 $|A|$ を推定し，両者の差は 1 以内であった．Algorithm 2 に示すように，サブタスクの実行時間 H が遊休時間 T を超える場合，提案手法は $|A|$ を 1 つだけ減少させる．したがって，全探索手法よりも 1 つだけ少ない $|A|$ を推定した．

一方，ヤコビ法をゲストタスクとする場合，提案手法は全探索のスレッドブロック数 $|A|$ と比べて総じて多い数を推定した．原因は考察中であるが，手法間の Sleep の方法に差があることが考えられる．

表 3 提案手法および全探索手法の推定したサブタスク粒度 $|A|$

フィルタ	行列積			ヤコビ法		
半径	提案	全探索	差	提案	全探索	差
5	33	34	1	251	134	117
10	28	28	0	231	129	102
15	25	25	0	194	111	83
20	17	18	1	145	94	51
25	10	10	0	74	60	14

4.2 マルチタスク実行時の性能

4.1 節で得られたスレッドブロック数 $|A|$ を用い，ホストタスクおよびゲストタスクを同時に実行し，ホストのフレームレート F およびゲストの実効性能 P を計測した（**図 4**）．評価には，環境 P を用い，ホストタスクとしてバイラテラルフィルタを用いた．また，ゲストタスクとして行列積およびヤコビ法を用い，それらの実行時間を Z，行列（あるいは画像）の大きさを $N \times N$ として，それぞれの実効性能 P を $2N^3/Z$ および $4N^2/Z$ で与えた．なお，行列積は単精度，ヤコビ法は倍精度の浮動小数点演算を用いた．

図 4(a) および**図 4(b)** の結果より，ゲストタスクが行列積であれヤコビ法であれ，いずれの手法もフレームレート F を目標の 60 fps に維持でき，ゲストタスク実行時にホストへの外乱を回避できている．このとき，行列積の実効性能 P に関しては，全探索と比べて最大 5%の低下に抑えた（**図 4(c)**）．

一方，ヤコビ法の実効性能 P に関しては（**図 4(d)**），全

表 4　安易な同時実行におけるレンダラのフレームレート（fps）

ホストタスク	環境 P		環境 M	
	専有時	同時	専有時	同時
フィルタ	60	2	60	ほぼ 0
レンダラ	60	12	60	ほぼ 0

探索と比べておよそ 60%ほど増加した．増加した原因として，4.1 節で得られたスレッドブロック数 $|A|$ が全探索と比べて大きいことが挙げられる．

次に，Pascal の持つプリエンプション機能を評価するために，ゲストタスクを分割せずにそのままホストタスクと同時に実行し，フレームレート F およびゲストの実効性能 P を計測した（表 4）．評価には，環境 P および環境 M を用い，ホストタスクとしてレンダラおよびバイラテラルフィルタ，ゲストタスクとして行列積を用いた．フィルタ半径は 20 とした．

ホストタスクおよびゲストタスクをそのまま実行した場合，いずれの環境においてもホストタスクのフレームレート F は低下した．ただし，プリエンプション機能を備える環境 P は，その低下を 2 fps に留めている．したがって，Pascal アーキテクチャを用いることで，マルチタスク実行時のフレームレート低下をわずかに抑制できているが，60 fps 程度の描画を同時に保証するためには協調マルチタスクが必要である．

4.3　ホストタスクの実行時間推定

ホストタスクの実行時間 H に関して，空カーネルによる推定精度を評価した．評価には，環境 P および環境 M を用い，レンダラおよび行列積の H を推定した（**図 5**）．

図 5 では，推定値が実測値と一致する場合に，破線上に計測点が出現する．結果として，いずれのアーキテクチャやプログラムにおいても，プログラムの実行時間 H が 1 秒未満であれば，空カーネルによる推定手法は最大でも 1%の誤差で実測値を推定できている．ただし，H が 1 秒以上である場合，誤差が大きい．

この誤差は実運用の観点からは問題ないと考えている．ホストタスクとして考慮すべきプログラムは描画を伴うものが多く，1 秒を超えて GPU プログラムが実行され続けることは稀である．仮に，そのようなプログラムがあれば，そのフレームレート F は専有時においても 1 fps を下回ることになり，実用に耐えないためである．例えば，動画やゲームであれば F は 20〜30 fps である．したがって，1 秒未満の H に対する 1%の誤差は良好な結果である．

4.4　推定に要する実行時オーバヘッド

環境 P を用い，ホストタスクの実行時間推定に要するオーバヘッドを評価した．実験では，バイラテラルフィルタの実行中に行列積をゲストタスクとして同時に実行し，

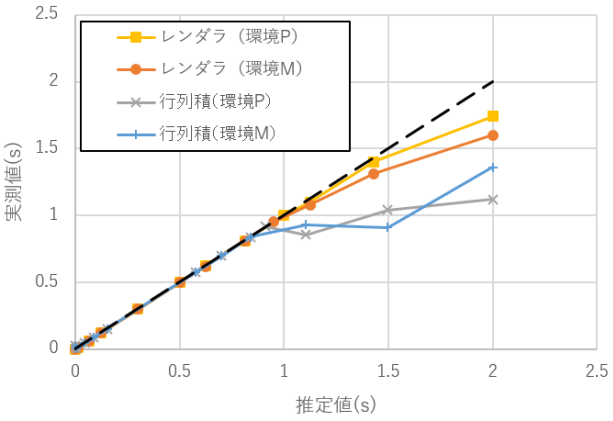


図 5　ホストタスクの実行時間推定結果

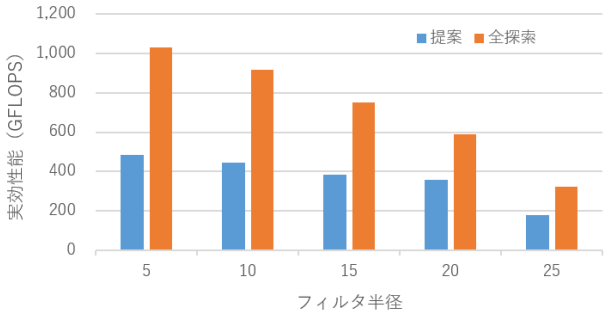


図 6　推定オーバヘッドに起因する実効性能の低下（行列積）

推定値が定まるまでの実効性能 P を計測した（**図 6**）．この収束過程のサブタスク粒度 $|A|$ は最終的な値と比べてやや小さいため，4.3 節の結果よりも P が低下することに注意されたい．なお，提案手法は最終的な値を得るまでに 4 回の反復実行を必要とした．また，全探索手法はプログラム実行時の推定を必要としないため，オーバヘッドは生じない．

図 6 に示すように，推定オーバヘッドにより P は全探索のときと比べて 50%程度にまで低下した．原因として，$|A|$ が小さい場合には 1 スレッドブロックあたりのカーネル実行時間 $S/|A|$ を $|A|$ が大きな場合と比べて大きく推定してしまうことが挙げられる．

推定オーバヘッドを小さくするための対策として，$|A|$ の初期値として 1 の代わりに大きな値を用いる方法がある．この方法は，事前に適切な $|A|$ の値がおおまかに分かる場合に有用であり，初期値を大きくすることで反復回数を削減できる．ただし，大きな初期値はホストタスクのフレームレートを低下させるために注意が必要である．

5.　関連研究

Ino ら [8] のサイクル共有システムは，CPU の使用率，ビデオメモリの使用量およびマウス・キーボードの操作から遊休 GPU を検出している．このシステムは，マウス・キーボードを操作中の計算資源を繁忙状態とみなし，これ

らの操作イベントを検出すれば，ゲストタスクの実行を即座に中止する．さらに，ホストタスクへの外乱を防ぐために，1 回あたりの実行が 100 ミリ秒以内に完了するようにゲストタスクを分割している．ただし，このシステムが活用できる遊休時間の長さは少なくとも 100 ミリ秒であり，ミリ秒単位の遊休時間は活用できない．

これらソフトウェアに基づく協調マルチタスク手法に対し，Pascal アーキテクチャ [3] は，GPU 向けのプリエンプションを初めて実現した．しかし，ミリ秒単位の遊休時間を活用する用途には不十分で，ホストタスクのフレームレート F を 60 fps 程度に維持できない（4.3 節）．

Chen らが提案するシステム [9] は，GPU プログラムをプリエンプション可能な記述に変換し，独自の API を用いて実行することでプリエンプティブマルチタスクを実現している [10]．しかし，ホストタスクのプログラムを書き換える必要があり，サイクル共有システムのように，ホストタスクのプログラムを入手することが難しい環境には向かない．

6. まとめと今後の課題

本論文では，GPU サイクル共有システムのための協調マルチタスク実行を容易にするために，ゲストタスクの分割数を推定する手法を提案した．提案手法は，呼び出し直後に処理を返す空カーネルを用い，ゲストの立場からホストタスクの実行時間を基に，ゲストタスクのために利用できる遊休時間の長さを推定する．また，推定で得られた時間内にゲストタスクの処理を完了できるよう，適切なサブタスク粒度，すなわちスレッドブロック数を自動的に得る．これにより，従来は必要とされていた，適切なサブタスク粒度の探索を自動化できる．

実験では，提案手法を従来の全探索手法と比較した．結果，提案手法は全探索手法と同様にホストタスクのフレームレートを維持できた．また，ゲストタスクの実効性能を最大で 5%の低下に抑えた．さらに，プリエンプション機能を備える Pascal アーキテクチャにおいて，ゲストタスクならびにホストタスクの安易な同時実行は，フレームレートを 2 fps まで低下させることを確認した．

今後の課題としては，対象とするホストタスクの制約を除去することが挙げられる．例えば，現在はフレームレート F が一定であることを仮定しているが，F が動的に変動するホストタスクへ対応できれば，より多くのホストを許容できる．

謝辞 本研究の一部は，科研費 15K12008，15H01687 および 16H02801 の補助による．

参考文献

[1] The Folding@Home Project: Folding@Home Distributed Computing (2010). `http://folding.stanford.edu/`.
[2] NVIDIA Corporation: CUDA C Programming Guide Version 8.0 (2017).
[3] NVIDIA Corporation: NVIDIA Tesla P100 (2016). `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.
[4] Ino, F., Oka, Y. and Hagihara, K.: A Fine Grained Cycle Sharing System with Cooperative Multitasking on GPUs, *Int'l J. Networking and Computing*, Vol. 4, No. 2, pp. 236–250 (2014).
[5] Ino, F., Ogita, A., Oita, K. and Hagihara, K.: Cooperative Multitasking for GPU-Accelerated Grid Systems, *Concurrency and Computation: Practice and Experience*, Vol. 24, No. 1, pp. 96–107 (2012).
[6] Shreiner, D., Woo, M., Neider, J. and Davis, T.: *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, fifth edition (2005).
[7] Beepa Pty Ltd.: Fraps: real-time video capture & benchmarking (2011). `http://www.fraps.com/`.
[8] 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).
[9] Chen, G., Zhao, Y., Shen, X. and Zhou, H.: EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU, *Proc. 22nd ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'17)*, pp. 3–16 (2017).
[10] Lin, Z., Nyland, L. and Zhou, H.: Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching, *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'16)*, No. 77 (2016).