社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2009－OS－110 (15)
2009／1／29

GPUグリッドにおいて描画および科学計算を並行処理するための制御手法

荻田章博[†1] 伊野文彦[†1] 萩原兼一[†1]

本稿では，GPUを持つグリッド環境において，描画および科学計算の両立を目的として，両者を並行処理するための制御手法について述べる．GPU上でこれらを同時に実行すると，描画のフレームレートや科学計算のスループットが極度に低下する問題がある．そこで提案手法では，フレームレートの低下を回避するために，科学計算側のタスクを分割し，タスク間の実行間隔をフレームレートに合わせる．また，フレームレートやスループットを最大化するために，いくつかの方針にしたがった性能尺度を提案する．実験の結果，一定の周期で描画を繰り返すアプリケーションに対しては，提案手法により両者の性能を制御できることが分かった．

A Method for Concurrent Processing of Graphics and Scientific Computation on the GPU Grid

Akihiro Ogita,[†1] Fumihiko Ino[†1] and Kenichi Hagihara[†1]

This paper describes a method for concurrent processing of graphics and scientific computation on the GPU grid. One critical problem on this environment is that the framerate of graphics applications and the throughput of scientific applications can significantly drop when they are executed at the same time on the same GPU. To prevent such significant drop, the proposed method divides scientific tasks into smaller portions and executes each of them at the same intervals as in the concurrent graphics application. We also propose some performance metrics to maximize the framerate and throughput. Experimental results show that our method is useful to control the framerate and throughput for graphics applications that periodically updates the display.

1. はじめに

GPU（Graphics Processing Unit）[1),2)]とは，グラフィクス処理を高速化するための演算器であり，ゲームなどを通じて一般家庭に普及している．GPUはCPUと比較して高い浮動小数点演算性能を持ち，SIMD型[3)]の並列計算が可能である．特に，開発環境CUDA（Compute Unified Device Architecture）[4)]の公開以降，プログラミングの敷居が低下し，実行時間の長い科学計算の高速化に貢献している．

一方，計算グリッドはネットワーク上の計算機群を統合する分散計算システムである．例えば，Folding@Homeシステム[5)]は，個人が所有する計算機群のうち，遊休状態にあるものを仮想的に集め，4PFLOPSを超える実効性能を提供する．このシステムは可視化資源というよりは計算資源としてGPUに着目し，全体性能の過半数をGPUが提供する．一方，GPUの台数は全体に対して数%に過ぎない．したがって，GPUはより少ない台数で高い性能を引き出せ，グリッドシステムにおいてGPUに着目する意義は大きい．

既存システム[5),6)]は，スクリーンセーバを用いて遊休状態の計算資源を検出する．これにより，資源所有者が計算資源上でローカルアプリケーション（LA）を実行していないときに限り，グリッドアプリケーション（GA）を実行する仕組みを実現している．つまり，LAとGAの同時実行を回避し，一方（あるいは双方）の性能が極度に低下しないよう実行を排他的に制御している．実際に，GPU上で複数のアプリケーションを安易に同時実行する場合，資源所有者およびグリッドユーザの双方に対して性能面で解決すべき問題がある．例えば，画面描画のフレームレートが低下してLAのコマ落ちが頻繁に発生したり，GAの実効性能がCPU版よりも低下する可能性がある[7)]．

しかし，このような排他実行は真の資源共有とは言いがたい．実際に，Folding@Homeシステムでも大多

†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

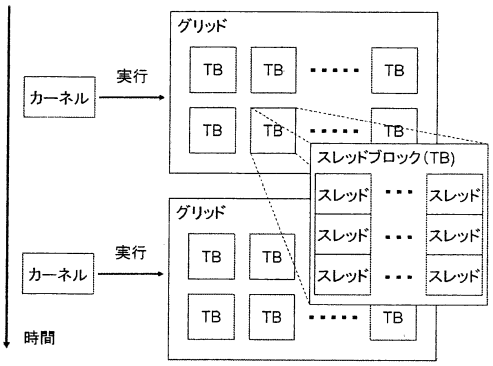


図 1 CUDA におけるスレッドの階層構造

数の計算資源は 24 時間に渡り GA のみを専有的に実行し続けている．GPU はゲームや CAD の高速化を目的として普及しているため，一般家庭や企業においてまだ多くの GPU が放置されている．

そこで本研究は，そのような埋蔵資源の共有を促進することを目的として，LA におけるフレームレートの低下を抑えつつ，GA を並行実行する手法を提案する．提案手法により，所有者の使用感を損ねることなく，グリッドユーザは GPU を使用でき，より多くの GPU がグリッドへ参加できる仕組みを提供する．

提案手法は，GA として CUDA で記述された科学計算を前提とし，LA として DirectX[8] や OpenGL[9] などのグラフィクス API で記述されたゲームや CAD などのグラフィクスアプリケーションを対象とする．また，OS として Windows を前提とする．GA および LA の性能を制御するために，提案手法は GA 側の CUDA プログラムを修正し，LA との協調的なマルチタスクを実現する．なお，LA 側のプログラムを実際のグリッド環境で変更することは困難である．

以降では，2 章で開発環境 CUDA について述べる．3 章で提案手法について説明し，4 章で実験結果を示す．最後に，5 章で今後の課題とともに本稿をまとめる．

2. 開発環境 CUDA

CUDA[4] は，nVIDIA 社の GPU を対象とする開発環境である．GPU 上で動作するプログラム（カーネル）を C 言語の拡張で記述できるため，既存の CPU プログラムを比較的容易に移植できる．CUDA は，数万個以上のスレッドを M 基のマルチプロセッサ上で SIMD 演算することにより，計算の高速化を図る．現在の GPU において M は 16〜30 である．

図 1 に，CUDA におけるスレッドの階層構造を示す．スレッドは，いくつかのグループに分類されていて，各々のグループをスレッドブロック（TB : Thread Block）と呼ぶ．同一 TB に所属するスレッドは同期できるため，スレッド間のデータ依存や共有メモリの使用が許される．しかし，TB 間にデータ依存が存在してはならない．したがって，すべてのスレッドを同期させるためには，カーネルを 2 つに分割し，各々を逐一実行する必要がある．

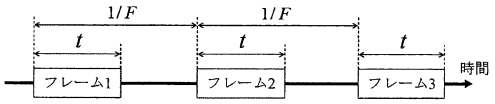


図 2 提案手法が前提とする描画モデル

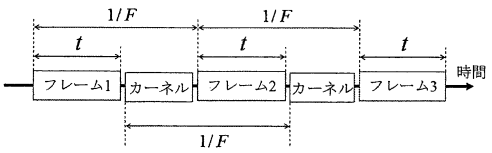


図 3 提案手法が目標とする並行実行

カーネルは TB の集合（グリッド）を処理する．TB はマルチプロセッサへの最小の割当単位である．したがって，各マルチプロセッサの計算負荷を均等にするために，カーネル 1 つあたりが処理する TB 数 n は M で割り切れることが望ましい．なお，レジスタや共有メモリなどの計算資源が許す限り，マルチプロセッサは複数の TB を同時に実行する．この際，TB 間にデータ依存がないため，マルチプロセッサは実行中の TB を自由に切り替えることができる．この切り替えはメモリ遅延の隠蔽に役立つ．

3. 提案手法

一般に，ゲームや動画再生のようなグラフィクスアプリケーションは，映像の再生速度がハードウェアに依存して速くなりすぎないよう，再生速度を一定に保つための制御機構を持つ．そこで提案手法では，LA はフレームを一定周期 F で更新するものとする．**図 2** に，提案手法が前提とする描画モデルを示す．ここで，F は目標とする 1 秒あたりの描画フレーム数（フレームレート）を表し，t はフレーム描画に要する時間を表す．周期 F を維持するためには，$t \leq 1/F$ である必要がある．そうでない場合，フレームのコマ落ちが発生する．なお，提案手法が対象とするグリッド環境では，LA に依存する t を測定することは困難である．

上記の描画モデルにおいて，LA および GA を並行実行したときの様子を**図 3** に示す．予備実験の結果，

LA のためのフレーム描画は GA のためのカーネル実行と排他的に実行されることが分かっている．つまり，現在の GPU は割り当てられたタスクの完了時にタスクを切り替える．ここで，タスクとは，OpenGL プログラムであれば glFlush 関数で実行される一連の描画命令に対応し，CUDA プログラムであれば CPU が起動するカーネルそのものである．したがって，時間 $1/F$ ごとに LA の描画処理を開始できるよう，GA におけるカーネルの実行時間を制御すればよい．カーネルの実行時間が長すぎる場合，$1/F$ 経過後に描画処理を再開できず，フレームレートが低下してしまう．

そこで，提案手法は LA におけるフレーム描画の合間に，適切な長さのカーネルを実行することにより，GA および LA の並行実行を実現する．具体的には，空き時間 $1/F - t$ 以内に実行が完了するカーネルを LA 側のフレーム描画ごとに 1 回だけ実行し，協調的なマルチタスクを実現する．提案手法は，以下に挙げる 3 つの要素技術からなる．

(1) タスクの分割．カーネル 1 つあたりの実行時間が $1/F - t$ 以下になるよう，GA のタスク（グリッド）を分割する．

(2) 実行順序の制御．LA 側のフレーム描画および GA 側のカーネル実行が交互に実行されるよう，GA 側のカーネル呼び出しの周期を F に合わせる．

(3) タスク粒度の制御．LA 側のフレームレートおよび GA 側のカーネル性能を最大化するタスクの分割数 d を選択する．

以降では，各々について述べる．

3.1 タスクの分割

提案手法は，CUDA において TB が互いに独立であることに着目し（2 章），TB 単位でタスクを分割する．このとき，TB そのものの大きさや各スレッドが処理する内容は変更しない．したがって，カーネルを変更する必要はない．替わりに，1 回のカーネル呼び出しで処理する TB 数 n を変更すればよい．具体的には，CPU からカーネルを呼び出すときに与える引数を修正し，引数を変えながら呼び出しを繰り返せばよい．このように，GPU 上で動作するカーネルそのものは変更しないため，メモリ遅延の隠蔽（2 章）を除いて最適化の状況を壊す恐れはない．

一般に，カーネルが処理する仕事量は TB 数 n に比例する．そこで提案手法では，カーネルの実行時間が $B\lceil n/M \rceil$ で表せるものとする．ここで，B はマルチプロセッサが TB を 1 つ処理するときに要する時間である．一般に，最適化を施したカーネルでは n は M の倍数であり，M よりも十分に大きい（2 章）．したがって，分割前のカーネル実行時間が K $(> 1/F - t)$ のとき，タスクを $\lceil K/(1/F - t) \rceil$ 個に分割すれば，(1) を満たせる．ただし，GPU は可能な限り複数の TB を同時に実行するため，この見積りは必ずしも正確なものではないことに注意されたい．

3.2 実行順序の制御

(1) を満たすようタスクを分割したとしても，次のフレームが描画される前に，複数のカーネルを立て続けに実行してしまうと，LA においてフレームのコマ落ちが発生する．つまり，カーネルの各々が空き時間 $1/F - t$ 以内に実行を終えたとしても，それらの和が $1/F - t$ を超えてしまうと分割の意味がない．(2) は，そのような事態を避けるために，フレームの合間にカーネルを高々1 回ほど呼び出すことを実現する．

一般に，実際のグリッド環境では，LA および GA は独立に実行されている．また，任意の LA および GA に対して，両者を同期するための仕組みを開発することは難しい．そこで，提案手法は GA におけるカーネルの呼び出し周期を，LA におけるフレーム描画の周期 F に合わせる．この制御は，待ち関数を CPU 側のプログラムに挿入し，カーネルを呼び出すたびに待ち時間 $w = 1/F$ を調節することにより実現する．なお，現在の CUDA では，カーネルはノンブロッキング実行されるため，カーネルそのものの実行時間を w に反映する必要はない．

待ち関数には，Win32 API が提供する Sleep 関数を用いる．Sleep 関数を呼び出した直後，GA のプロセスは待機状態へ遷移する．したがって，ビジーループによる待ちと比較して，CPU 負荷を軽減できる．ただし，秒間 30〜60 フレーム程度の描画を実現するために，提案手法は 1 ミリ秒単位の正確な待ちを必要とする．一方，Sleep 関数の分解能はハードウェアタイマの分解能や OS のタイムスライスに依存して決まる．例えば，マルチプロセッサ上で動作する Windows であれば，デフォルトで 15 ミリ秒である．そこで，GA を実行しているときに限り，コンテキストスイッチの間隔をデフォルトの 15 ミリ秒から 1 ミリ秒に変更し，Sleep 関数の分解能を向上する．具体的には，timeBeginPeriod 関数を用いて変更すればよい．なお，一部のゲームは自身の制御のためにデフォルト値を変更していて，LA 側があらかじめ分解能を向上させている可能性はある．

3.3 タスク粒度の制御

3 章で述べた通り，グリッド環境において LA の描画時間 t を計測することは難しい．そこで，何らかの性能尺度のもとで，(1) を満たすよう GA のタスク粒

度を制御する必要がある．提案手法は，単独実行におけるフレームレート F，並行実行におけるフレームレート f，および分割前のカーネル実行時間 K が与えられたとき，次のフレームレートも f であるものとして，タスク分割数 d を決定する．そのために，提案手法は 3 種類の性能尺度 V_1〜V_3 を提供する．

まず，GA に対する性能尺度 G は，単体実行時の性能に対する比率で与える．単体実行時におけるカーネルの実行時間を T_1 とし，並行実行時における分割済カーネルの実行時間の和を T_2 とすると，G は次式で定める．

$$G = T_2/T_1 \tag{1}$$

次に，LA に対する性能尺度 L は，資源所有者の方針に依存していくつかの匙加減が考えられる．提案手法は，以下に挙げる 3 つの候補 L_1〜L_3 を基に，性能尺度 V_1〜V_3 を提供する．

- V_1：単独実行時のフレームレート F を維持できるときのみ並行実行する方針．この方針は，資源所有者の使用感を低下させないことを保証する．つまり，$f \geq F$ のときに限り G を返す．

$$L_1 = \begin{cases} 1, & \text{if } f \geq F, \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

$$V_1 = GL_1 \tag{3}$$

- V_2：LA および GA を区別することなく両性能の和を最大化する方針．この方針は，効率よく計算資源を稼働させることを目指す．G と同様に，L も単独実行時の性能に対する比率で与える．

$$L_2 = f/F \tag{4}$$

$$V_2 = G + L_2 \tag{5}$$

- V_3：並行実行時のフレームレート f が閾値 ϵ を下回らないよう GA の性能を最大化する方針．この方針は，LA において若干のフレームレート低下を許しつつ，GA の高速化も目指す．基本的には，V_2 と似た性能尺度だが，$f < \epsilon$ において負の値をペナルティとして課している．

$$L_3 = (f - \epsilon)/(F - \epsilon) \tag{6}$$

$$V_3 = G + L_3 \tag{7}$$

ここで，性能尺度 L_3 は $f < \epsilon$ のときに $L_3 < 0$ であり，$f = F$ のときに $L_3 = 1$ である．

4. 実　　験

提案手法を評価するために行なった実験の結果を示す．実験では，(1) および (2) の要素技術により，LA のフレームレートおよび GA の実効性能を制御できることを確認する．また，(3) に対しては，3 つの性能尺度のなかで実用的なものを調査する．

表 1　行列積におけるタスク分割と実行時間

d：タスク分割数	k：カーネル実行時間（ms）		実効性能
	実測値	モデル値	(GFLOPS)
分割なし	299.9	299.9	57.3
4	75.0	75.0	44.5
8	37.5	37.5	37.9
16	18.8	18.7	28.9
32	9.4	9.4	31.6
64	4.8	4.7	15.8
128	2.4	2.3	7.9

4.1 実験環境

実験環境として使用した計算機は，CPU として Intel Core 2 Duo 1.86 GHz を持ち，GPU として nVIDIA GeForce 8800 GTS（G80 アーキテクチャ）を搭載する．この GPU では $M = 12$ である．

GA として CUDA[4] とともに配布されている行列積を用いる．行列サイズは 2048 × 2048 である．各 TB は 16 × 16 スレッドで構成されていて，分割前のカーネルが一度に処理する TB 数 n は 16,384 ($= 2048 \cdot 2048/(16 \cdot 16)$) である．一方，LA には OpenGL で記述された球体のレンダラ[10] を用いる．このレンダラは $F = 60$ fps で動作する．したがって，待ち関数による待ち時間 $w = 1/F$ は 17 ミリ秒とした．

4.2 並行実行の評価

まず，タスク分割によりカーネル 1 つあたりの実行時間 k を短縮できることを確認した．**表 1** に，タスク分割数 d を変動させたときの k を示す．このように，TB 数 n を変更することで k を制御できる．また，モデル値が示すように，k と n はほぼ比例していて，$K = 299.9$ さえ計測できれば，自由に k を制御できる．さらに，実効性能として浮動少数点演算性能 FLOPS を計測した．d の増大とともに FLOPS 値は低下していて，その主な原因は待ち時間の累積 dw にある．待ち時間を除外してカーネル実行の部分のみを比較すれば，さほど FLOPS 値は低下していない．

次に，(2) の効果を確認する．**図 4** に，タスク分割数 d とともにカーネル 1 つあたりの実行時間 k を変えたときのフレームレート f を示す．レンダラ待ちありは待ち関数を挿入した場合の結果であり，レンダラ待ちなしは挿入しない場合の結果である．図 4 より，待ち関数による効果を確認できる．待ちなしでは，$k \geq 9.4$ において $f < F$ だが，待ちありでは $k = 9.4$ においても F をほぼ達成できている．また，タスク粒度が減少するとともに f が向上している．このことからタスク分割により f を向上できている．

次に，待ち関数を挿入した状態で LA および GA を

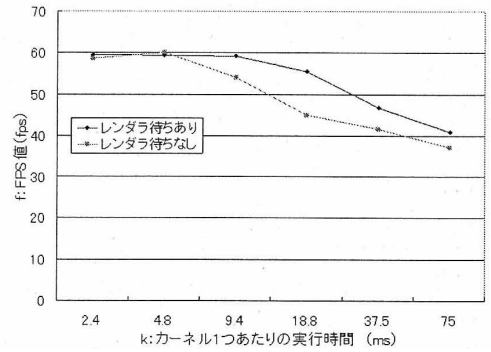


図 4 待ち関数の挿入によるフレームレートの制御

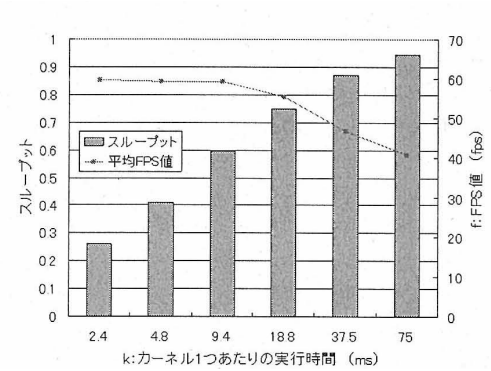


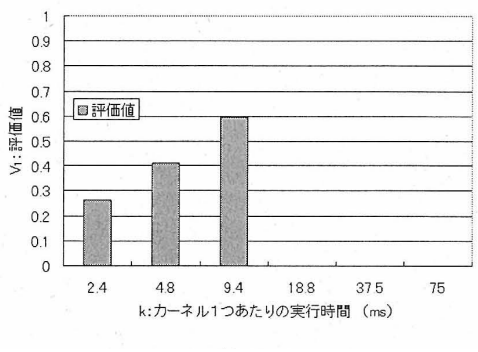
図 5 タスク粒度を変えたときの LA のフレームレートおよび GA のスループット

並行実行し，LA のフレームレート f および GA のスループットを計測した．**図 5** に，タスク分割数 d とともに k を変えたときの結果を示す．スループットは k に比例して増加している．この理由は，CUDA プログラムに挿入した待ち関数の数にあり，d が小さいほど待ち時間の総和が小さく，スループットが向上する．このように，GA および LA の実効性能はトレードオフの関係にある．両者に関してよいバランスを実現するためにも，性能尺度の決め方が大切である．

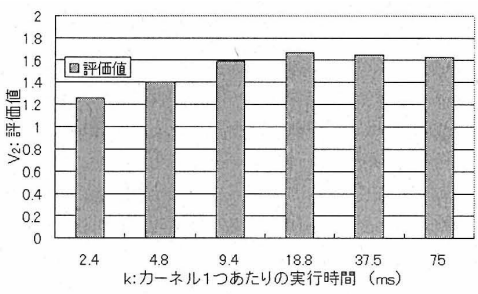
4.3 性能尺度の評価

図 6 に，3 つの性能尺度に対する評価結果を示す．待ち時間 w を 17 ミリ秒としたため，$F = 59$ として評価した．また，V_3 に対しては，閾値 ϵ を 40 もしくは 50 とした．

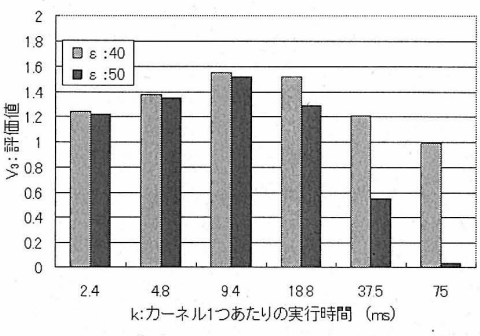
図 6(a) に示すように，V_1 は f の低下を避けるために，$k \geq 1/F$ において $V_1 = 0$ となる．逆に，$k < 1/F$ では，GA の性能尺度がそのまま評価値となるため，GA の性能が最大化される．実験に用いた LA では，カーネル 1 つあたりの実行時間 k が 9.4 ミリ秒のときに，V_1 は最大となった．



(a) V_1



(b) V_2



(c) V_3

図 6 性能尺度の評価

次に，図 6(b) に示す V_2 の結果について述べる．図 4 と比較すると，最大の f をもたらす k において，必ずしも V_2 が最大になっていない．むしろ f を若干低下させる大きなタスク粒度（$k = 18.8$）において，空き時間 $1/F - t$ を無駄なく使用できていると考えられる．しかし，この性能尺度は資源所有者の使用感を考慮しない．したがって，資源所有者が貢献の度合を自由に制御できず，実用面に難がある．

最後に，図 6(c) に示す V_3 の結果について述べる．$\epsilon = 50$ のとき，V_1 と同様に $k = 9.4$ において評価値が最大であるのに対し，$\epsilon = 40$ では $k = 18.8$ で評価値が最大となる．つまり，資源所有者が閾値 ϵ を用いて許容できるフレームレートを高く設定すれば，それに応じて GA のスループットを向上でき，逆も制御できる．このように，V_2 で指摘した問題を ϵ により解決できる可能性がある．ただし，ϵ の効果は GA や LA の組み合わせに依存して変わる可能性もあり，他の組み合わせに関してさらに実験が必要である．

まとめると，資源所有者に対して F を保証する場合，V_1 を使用すべきである．しかし，グリッドユーザに対して高い性能を提供できない可能性もある．一方，V_3 は V_2 における問題を解決できる．資源所有者に対しては，フレームレートを低下させてしまうが，閾値 ϵ は資源所有者側で自由に設定できる．このように，それぞれの性能尺度において実効性能を最大化する k は異なる．トレードオフの関係ゆえに，絶対的な性能尺度やタスク分割数は存在しない．状況に応じて適切な性能尺度を選択する必要がある．

5. ま と め

GPU グリッド環境にて描画および科学計算の両立を実現するために，GPU 上のマルチタスク処理を目指した．提案手法は，描画側のフレームレートや科学計算側のスループットが低下する問題に対し，科学計算側を協調的に実行させることで問題の解決を図る．具体的には，フレームレートの低下を回避するために，科学計算側のタスクを分割し，それらの実行周期を描画側の描画周期に合わせる．結果，描画側に対して十分な描画機会を確保できる．また，科学計算側のスループットおよび描画側のフレームレート間のトレードオフに対しては，いくつかの方針に基づいてタスク分割数を決めるための性能尺度を示した．

実験の結果，一定の周期で描画を繰り返すアプリケーションに対しては，提案手法により両者の性能を制御できることが分かった．また，フレームレートに関して閾値 ϵ を持つ性能尺度は，資源所有者側に科学計算への貢献の度合を調節する権限を与え，その範囲のなかでグリッドユーザは科学計算の性能を最大化できる．

今後の課題としては，フレームレートが動的に変動するアプリケーションへの対応が挙げられる．

謝辞 本研究の一部は，科学研究費補助金基盤研究 (A) (2) (20240002)，若手研究 (B) (19700061) 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) Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E. and Purcell, T.J.: A Survey of General-Purpose Computation on Graphics Hardware, *Computer Graphics Forum*, Vol.26, No.1, pp.80–113 (2007).
2) 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).
3) Grama, A., Gupta, A., Karypis, G. and Kumar, V.: *Introduction to Parallel Computing*, Addison-Wesley, Reading, MA, second edition (2003).
4) nVIDIA Corporation: CUDA Programming Guide Version 2.0 (2008).
5) The Folding@Home Project: Folding@Home Distributed Computing (2008). `http://folding.stanford.edu/`.
6) Kotani, Y., Ino, F. and Hagihara, K.: A Resource Selection System for Cycle Stealing in GPU Grids, *J. Grid Computing*, Vol.6, No.4, pp.399–416 (2008).
7) Kotani, Y., Ino, F. and Hagihara, K.: A Resource Selection Method for Cycle Stealing in the GPU Grid, *Proc. 4th Int'l Symp. Parallel and Distributed Processing and Applications Workshops (ISPA'06 Workshops)*, pp.939–950 (2006).
8) Microsoft Corporation: DirectX (2007). `http://www.microsoft.com/directx/`.
9) Shreiner, D., Woo, M., Neider, J. and Davis, T.: *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, fifth edition (2005).
10) Teranishi, T.: Phong シェーディングサンプルプログラム (2003). `http://www.asahi-net.or.jp/~yw3t-trns/opengl/samples/fshphong/`.