Vol. 48 No. SIG x(ACS 19) 情報処理学会論文誌：コンピューティングシステム Aug. 2007

GPGPUアプリケーションの開発を支援するための性能モデル

伊藤信悟[†,☆] 伊野文彦[†] 萩原兼一[†]

GPGPU（General-Purpose Computation on Graphics Processing Units）とは，GPUをグラフィクス処理の枠を超えて汎用問題に適用する試みのことである．本稿では，典型的なGPGPU実装を対象として，GPUによる高速化の見込みを予測するための性能モデルを提案する．提案モデルは，GPGPU実装の多くがメモリ集中型の問題を対象として規則的にデータを参照する点に着目し，実装全体の性能を主記憶，ビデオメモリおよびGPU内演算器間の各データパスの転送性能で表す．転送性能の各々は，GPGPUアプリケーションとは独立に計測できるバンド幅および遅延時間のみの簡単な組で表す．提案モデルを画像フィルタおよびLU分解に適用し，3世代に渡るGPU上で評価した結果，誤差は最悪で20%であった．GPU内キャッシュの効果がさほど大きくない場合，誤差は10%以内であることから，提案モデルは典型的な実装に対してGPUによる高速化の見込みを見積もる際に有用であると考える．

A Performance Model for Assisting Development of GPGPU Applications

Shingo Ito,[†,☆] Fumihiko Ino[†] and Kenichi Hagihara[†]

GPGPU stands for general-purpose computation on graphics processing units (GPUs), aiming at applying the GPU to general problems beyond graphics problems. This paper presents a performance model for typical GPGPU implementations, which is capable of predicting the possibility of the acceleration achievable by the GPU. Our model focuses on the fact that most of GPGPU implementations deal with memory-intensive problems and have regular access to data. Based on this fact, we represent the entire performance as the transfer performance of data paths connecting main memory, video memory, and processors inside the GPU. Each of the transfer performance here is simply represented by a combination of bandwidth and latency, which are independent of GPGPU applications. We applied the model to an image filter and LU decomposition to estimate their performance on three generations of GPUs. We found that the model has a 20% error at the worst case. We think that the model is useful for estimating the possibility of typical GPU-accelerated implementations, because the observed errors are less than 10% if GPU cache does not have significant effects on performance.

1. はじめに

近年，GPU[1),2)]（Graphics Processing Unit）の目覚しい性能向上や高いビデオメモリバンド幅に着目し，GPUを汎用計算（GPGPU：General-Purpose Computation on GPUs）に応用する研究が盛んである[3)]．例えば，数値計算[4)~6)]，バイオ情報[7)]，データベース[8)~11)]などの分野において，キャッシュの最適化や拡張命令セットSSE[12)]を用いる高速なCPU実装と比較して，多くの場合で1桁の性能向上を達成している．これら高速化に成功しているGPGPU実装の特徴として，対象問題がメモリ集中（memory intensive）型であり，かつ高い並列性を持つ点が挙げられる．また，多くの問題においてデータ参照が規則的である点も注目すべき特徴である．例えば，ランダム性の強い整列問題でさえ，アーキテクチャ制約の強いGPU上では参照が規則的なバイトニック整列を用いざるを得ない[9),10)]．

典型的なGPGPU実装は，ストリームプログラミングモデル[13)]を基に，GPU内でプログラム可能な演算器FP （Fragment Processor）上でSIMD演算やベクトル演算による高速化を試みる．例えば，行列積の実装[4)]では，行列データをテクスチャ（模様を表す

† 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
☆ 現在，株式会社カプコン
Presently with Capcom Co., Ltd.

画像）として保持し，そのテクスチャ上の各画素を並列に描画する．その際，GPU 上で動作するプログラムとして画素ごとの計算式を記述する．一方，CPU 側のプログラムでは，OpenGL[14] もしくは DirectX[15] などのグラフィクス API を用いて描画のための制御処理などを記述する．したがって，高速な GPGPU 実装の開発は，単に新しい言語の習得だけでなく，グラフィクスに関する知識を必要とする．

このように，GPGPU の敷居は低くない．したがって，GPU による高速化の見込みを予測できれば，プログラミングの労力に見合う性能を得られるか否かをあらかじめ判断でき有用である．特に，GPGPU 実装の開発者が手軽に扱える性能モデルがあれば，彼らのアルゴリズム開発を支援できる．

そこで本稿では，上記の典型的な GPGPU 実装を対象として，そのアルゴリズム開発において有用となる性能モデルを提案する．提案モデルは，GPGPU 実装における一連の処理がストリームプログラミングモデルに基づくことに着目し，いかに多くのデータを主記憶，ビデオメモリおよび FP までの各データパスに流せるかということを転送バンド幅および転送遅延の単純な組でモデル化する．提案モデルの新規性は，GPGPU 実装を開発する前に，その高速化の見込みを手軽に予測できる点にある．

以降では，まず 2 章で GPU の性能モデルに関する既存研究を紹介し，3 章で GPU のアーキテクチャについて述べる．次に，4 章で典型的な GPGPU 実装を示し，5 章で提案する性能モデルを示す．6 章でいくつかの GPU および GPGPU 実装を用いた実験結果を示す．最後に，7 章で本稿をまとめる．

2. 関 連 研 究

Govindaraju ら[16] は，科学技術計算を対象として，テクスチャキャッシュの振る舞いを考慮したメモリモデルを提案している．このモデルは，nVIDIA GeForce 7800 GTX のキャッシュサイズ（非公開）が 128KB であることを実験により推測している．また，モデルを基に，キャッシュを効率よく使うアルゴリズムを開発し，行列積に対して 1.2〜1.8 倍，整列に対して 2〜3 倍の性能向上を達成している．

しかし，モデル化の対象がビデオメモリの振る舞いであるため，このモデルは実行時間を提示しない．また，GPU 内部の計算のみを対象にしているため，主記憶・ビデオメモリ間における入出力データのやりとりを含めた高速化の見込みは予測できない．さらに，ユーザはキャッシュレベルの事象を緻密に解析する必要があるため，手軽に扱いづらい．一方，我々は GPU による高速化の見込みを容易に見積もることを目指している．また，GPU の設計はキャッシュのヒット率を向上することよりもビデオメモリバンド幅を使い切ることを重視している[2]．したがって，GPGPU を試みる初期の開発段階から，キャッシュの効果を考慮することは必然ではないと考える．

Buck ら[17] は，グラフィクスに関する知識を必要としないプログラミング環境 Brook を提案している．彼らは，Brook による実装を CPU 実装と比較するための性能モデルを示している．この性能モデルは，主記憶・ビデオメモリ間におけるテクスチャの転送時間および GPU 側の計算時間が画素数に比例することを前提として，1 画素あたりの転送時間 T_r および計算時間 K_{gpu} をパラメータとして必要とする．K_{gpu} の取得には，対象となる実装を実行する必要があるため，このモデルはプログラム開発前の予測を扱えない．また，パラメータの具体的な取得方法およびモデルの精度は明らかでない．これに対し，提案モデルはプログラム開発前の予測を提供し，パラメータの取得方法とともに精度を示している．

文献 18) は，GPU をシミュレートするための枠組みを提案している．この枠組みは，実行履歴を基に，GPU 内の性能ボトルネックをサイクル単位で解析できる．しかし，プログラム可能でない GPU 上のグラフィクス処理を対象にしているため，典型的な GPGPU 実装には適用できない．

3. GPU のアーキテクチャ

本章では，多くの GPU に共通する典型的なアーキテクチャの概要[2] についてまとめる．なお，最新（2006 年 11 月発表）の統合型アーキテクチャに基づく nVIDIA GeForce 8800（G80）は相違点が多いため，典型的なものからの差分を最後にまとめる．

図 1 に，アーキテクチャの概要を示す．GPU は，プログラム可能な演算器として FP および VP（Vertex Processor）を持つ．これらは，ラスタライザ[2] を挟んでパイプラインを構成している．一般に，ラスタライザは VP が出力したデータ（頂点）よりも多数のデータ（フラグメント）を生成するため，FP は VP よりも多くのデータを処理する必要がある．ゆえに，FP は VP よりも数が多く，かつ全体として 1 桁ほど高い浮動小数点演算性能を持つ．したがって，多くの GPGPU 実装は FP のみを計算に用いている．

FP は SIMD 型演算器であり，テクスチャを構成する各画素を並列処理できる．各画素は色 RGB および

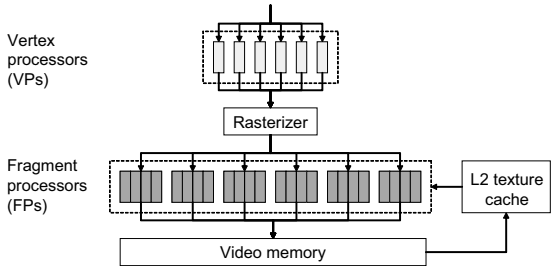


図 1 GPU アーキテクチャの概要
Fig. 1 Overview of GPU architecture.

透明度 A を持ち，これらは独立に処理できるため，FP は 4 要素をベクトル演算できる．なお，GPU は主記憶を直接参照できない．したがって，あらかじめテクスチャを主記憶からビデオメモリへ転送（ダウンロード）する必要がある．同様に，CPU が GPU による計算結果を参照する場合，ビデオメモリから主記憶へ転送（リードバック）する必要がある．

最後に，G80 の主な差分として，統合型アーキテクチャおよびスカラ演算器が挙げられる．G80 では，VP および FP が 1 種類（SP：Streaming Processor）に統合されている．SP は，VP もしくは FP としての役割を動的に変更できるため，G80 は計算負荷に応じて VP および FP の比率を変更できる．ここで，SP はスカラ演算器であるため，G80 はベクトル演算を提供しない．しかし，従来の VP・FP よりも 4 倍以上の SP を持ち，駆動周波数が高いため高速である．

4. 典型的な GPGPU 実装

図 2 の 1〜12 行目に，典型的な GPGPU 実装における処理の流れを示す．多くの実装では，入力データをテクスチャ $\mathcal{I}$ として保持し，FP が $\mathcal{I}$ を参照しながらテクスチャ $\mathcal{O}$ に描画することで出力データを得る．この際，あらかじめ $\mathcal{I}$ を主記憶よりビデオメモリへ転送し，FP による並列計算後に，$\mathcal{O}$ をビデオメモリから主記憶へ転送する．これらの制御には OpenGL[14] や DirectX[15] などのグラフィクス API を用いる．なお，我々の知る限り，GPGPU 実装における描画領域の形状は矩形である．複雑な形状も指定できるが，画素を斜めに横切る境界領域において，処理の扱いが難しい[7]．

FP における画素ごとの描画の詳細はフラグメントプログラムとして記述できる．図 2 の例では，I 種類のプログラム $P_1, P_2, \ldots, P_I$ を用意し，これらを順次差し替えながら描画を繰り返している（5 行目）．この際，マルチパスレンダリング技術を用い，FP の出力を次のレンダリングパスのための FP への入力として戻すことにより，繰り返しを実現する．フラグメントプログラムの記述言語としては，Cg[19] や GLSL[20]（OpenGL Shading Language）などの高級言語やアセンブリ言語がある．図 2 の 13〜17 行目は Cg による記述例である．この例では，各画素について，左右に隣接する 2 画素との和を出力している．

以降では，フラグメントプログラム中で参照する画素数を K とし，$\mathcal{I}$ および $\mathcal{O}$ の画素数を J とする．また，各画素は S バイトのデータを持つものとする．図 2 の例では，3 画素を参照しているため，$K = 3$ である．**表 1** に，本稿で使用する主な記号をまとめる．

```
Inputs: (1) texture I and (2) fragment programs P_1,
        P_2, ..., P_I, each accessing K pixels
Output: texture O
 1: Typical_GPGPU_program {
 2:   Send texture I from main memory to video
        memory;
 3:   foreach rendering pass 1 ≤ i ≤ I {
 4:     Bind texture I;
 5:     Set program P_i to draw texture O;
 6:     foreach pixel 1 ≤ j ≤ J {
          // this loop is automatically parallelized
 7:       Execute P_i for j on a fragment processor;
 8:     }
 9:     Swap texture I for texture O;
10:   }
11:   Send texture O from video memory to main
        memory;
12: }
13: float4 Program_P1(float2 coord:TEXCOORD0,
      uniform samplerRECT tex:TEXUNIT0):COLOR
14:   float4 p = texRECT(tex, coord);
15:   p = p + texRECT(tex, coord + float2(-1,0));
16:   return p + texRECT(tex, coord + float2(1,0));
17: }
```

図 2 典型的な GPGPU 実装における処理の流れ
Fig. 2 Processing flow of typical GPGPU implementations.

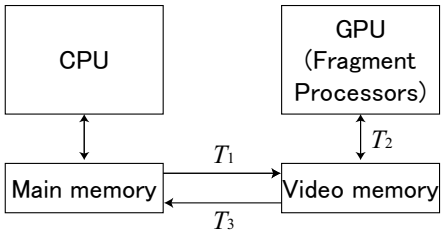


図 3 性能モデル
Fig. 3 Performance model.

表 1 主な記号の一覧
Table 1 Notation.

分類	記号	説明	備考
入力	I	レンダリングパス数	GPGPU 実装のアルゴリズム解析により得られるパラメータ
	J	テクスチャの画素数	
	K	フラグメントプログラム中の参照画素数	
	S	1 画素当たりのデータ量	
	B_d	T_d の転送バンド幅（$1 \leq d \leq 3$）	計測プログラムの実行により得られるパラメータ
	L_d	T_d の転送遅延（$1 \leq d \leq 3$）	
出力	T	全体の実行時間	
	T_1	主記憶からビデオメモリへの転送時間	
	T_2	ビデオメモリおよび FP 間の転送時間	
	T_3	ビデオメモリから主記憶への転送時間	
—	P_i	i 番目のフラグメントプログラム（$1 \leq i \leq I$）	

5. 性能モデル

図 3 に，提案する性能モデルを示す．この性能モデルは 10 個のパラメータを入力とし，GPGPU 実装の実行時間 T およびその内訳 T_1〜T_3 を出力する（表 1）．パラメータのうち，GPGPU 実装のアルゴリズム解析により得られるものは I，J，K および S である．残りの転送バンド幅 B_1〜B_3 および転送遅延 L_1〜L_3 は，後述する計測プログラムにより得る（5.3 節）．

提案モデルは，GPGPU 実装がデータを流すデータパスのうち，主記憶・ビデオメモリ間およびビデオメモリ・FP 間に着目し，各々の性能を B_1〜B_3 および L_1〜L_3 で表す．この際，前者は転送の向きを区別してモデル化する．実行時間 T は，以下の式で与える．

$$T = T_1 + IT_2 + T_3 \quad (1)$$

ここで，T_1〜T_3 は下記の通りである（表 1 参照）．

$$T_1 = JS/B_1 + L_1 \quad (2)$$

$$T_2 = KJS/B_2 + L_2 \quad (3)$$

$$T_3 = JS/B_3 + L_3 \quad (4)$$

提案モデルの特徴として，以下の 3 点が挙げられる．

C1. バンド幅と遅延でデータ転送性能を表す点

C2. ベクトル演算を含めた FP の計算性能をデータ転送性能で表す点

C3. 実行時間 T を各データパスの転送時間の和（式 (1)）で表す点

このうち C1 の妥当性は 6 章で検証する．以降では，残りの C2 および C3 の妥当性を示す．

なお，Buck らのモデル[17] では，実行時間 T を $J(K_{gpu} + T_r)$ で与える（2 章）．したがって，提案モデルは GPGPU 実装に依存するパラメータ K_{gpu} を IKS/B_2 に置き換えている．IKS/B_2 は GPGPU アプリケーションとは独立に計測もしくは決定できるため，開発前の予測を実現できる．さらに，モデルの表現力に関しては，提案モデルは転送遅延の項 $L_1 + IL_2 + L_3$ を T に含めている．この重要性は 6.2 節で示す．

5.1 GPU 内計算のモデル化

FP の計算性能をデータ転送性能として表す理由は，下記の 2 点による．

- ビデオメモリからの読み出しオーバヘッドが大きいこと．テクスチャ上の画素を読み出す tex 命令（アセンブリ命令）は，他の計算命令と比較して 20 倍以上に相当する実行時間を要する．
- ビデオメモリからの読み出しが頻繁であること．多くの GPGPU 実装は少なくとも 1/6 の高い頻度で tex 命令を呼び出している（**図 4**）．

このように多くの GPGPU 実装では，重い tex 命令を高い頻度で呼び出していることから，提案モデルは C2 の方針を採用している．

なお，FP の数を F として，FP1 基あたりのフラグメントプログラムの実行回数 A は $A = J/F$ である．したがって，厳密には T_2 における J は式全体にかかるべきものである（$T_2 = (KS/B_2 + L_2)J/F$）．しかし，モデルを簡素なものにするために，提案モデルでは F および A を隠蔽している．

さらに，提案モデルはベクトル演算による高速化効果を陽に表していない．この理由は，その効果をデータ転送性能で表せるためであり，テクスチャのフォーマットごとに下記のように説明できる．

- スカラ（GL_FLOAT_R32_NV）の場合．スカラのテクスチャをベクトル化することにより，1 画素あたりのデータ量 S が 4 倍になる一方，テクスチャサイズ J は 1/4 になる．したがって，実行回数 A が 1/4 に減少するが，提案モデルではこの差異を隠蔽している（上述）．
- ベクトル（GL_FLOAT_RGBA32_NV）の場合．計算がスカラ演算であるときもベクトル演算のときと同様に，ベクトル全体が FP へ転送される

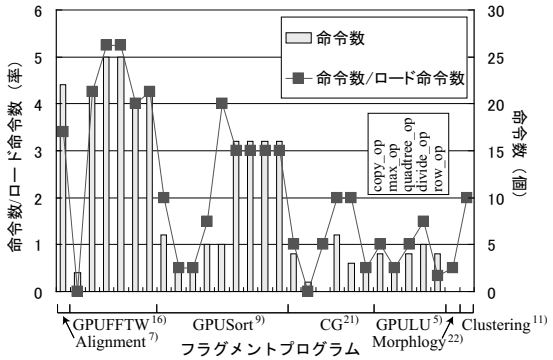


図 4 GPGPU 実装[5),7),9),11),16),21),22)] におけるテクスチャ読み出し命令に対する計算命令の呼び出し比率（いくつかのフラグメントプログラムは命令数が問題サイズ依存だが比率は一定）

Fig. 4 Rate of computation operation over texture load operation in GPGPU implementations[5),7),9),11),16),21),22)]. The number of instructions of some fragment programs depends on problem size but has the same rate.

ため，ベクトル化による高速化効果はない．実際に，スカラ演算とベクトル演算の実行時間は同一であった．提案モデルは T_1 および T_2 で同じ値 S を用いるため，効果のないことを表せている．

5.2 データ転送のモデル化

特徴 C3 において T_1〜T_3 を加算する理由は，下記のように説明できる．

- T_1 および T_2 について．ビデオメモリへのデータのダウンロードが完了しない限り，SIMD 型の FP はそのデータを参照する計算を開始できない．したがって，T_1 および T_2 を加算する必要がある．実際に，計算領域をテクスチャ全体から一部に削減しても，実行時間が変わらないため，全体の転送後に計算が開始されていた．
- T_2 および T_3 について．主記憶へのリードバック時には，GPU におけるパイプライン上のデータを追い出す（flush）必要がある[3),5),23)] ため，T_2 および T_3 を加算する必要がある．

なお，テクスチャのフォーマットに応じて，転送バンド幅および転送遅延の実効値は大きく変わり得る[23)]．例えば，8 ビットデータを持つテクスチャへの処理は，32 ビット版と比較して 4 倍を超える速度で動作している[24)]．同様に，これらの実効値はドライバのバージョンに強く依存しているため，テクスチャのフォーマットおよびドライバのバージョンごとにパラメータ値を使い分ける必要がある．

また，FP 上の計算と並行して，その計算で参照しないデータをダウンロードすることは可能である．したがって，データ量が多く，1 枚のテクスチャで保持できない場合，あるいはビデオメモリ容量が不足する場合，データ転送および GPU 内計算をパイプライン状にオーバラップできる．開発初期を焦点にする提案モデルは，そのような高度な実装[25)] を想定していないが，実行時間を $\max(T_1, T_2)$ で表せる可能性はある．

```
void cpuProgram(Region &region, // 描画領域
  TextureObject *texture, // テクスチャオブジェクト
  BufferSpecifier &drawBuffer) // 出力用バッファ
{
  glBindTexture(texture); // 入力用テクスチャをバインド
   // ダウンロード時間 T1 の計測
  StopWatch.Start(); // 計測開始
  glTexSubImage(); // 主記憶からビデオメモリへの転送
  StopWatch.Stop(); // 計測終了
   // 計算時間 T2 の計測
  cgGLBindProgram(fragmentProgram); // 図 (b) 参照
  glDrawBuffer(drawBuffer); // 出力用バッファを指定
  glClear(GL_COLOR_BUFFER_BIT); // クリア
  StopWatch.Start();
  glBegin();
  glRecti(region); // 描画領域を指定
  glEnd();
  glFinish(); // 発行済 OpenGL コマンドを強制実行
  StopWatch.Stop();
   // リードバック時間 T3 の計測
  StopWatch.Start();
  glReadPixels(); // ビデオメモリから主記憶への転送
  StopWatch.Stop();
}
```

(a) CPU program

```
// K × 1 画素の和を計算（K = 3 かつベクトルの場合）
float4 fragmentProgram(float2 coord:TEXCOORD0,
  uniform samplerRECT tex:TEXUNIT0):COLOR
{
  float4 out = texRECT(tex, coord);
  out += texRECT(tex, coord + float2(-1,0));
  out += texRECT(tex, coord + float2(1,0));
  return out;
}
```

(b) Fragment program

図 5 パラメータ計測プログラムの擬似コード

Fig. 5 Pseudocode for parameter estimation.

5.3 パラメータ値の計測方法

図 5 に，パラメータ値を得るための計測プログラムを示す．パラメータ B_2 および L_2 の取得には，$\sqrt{J}\times\sqrt{J}$ 画素のテクスチャ $\mathcal{I}$ に対し，画素ごとに自身を中心とした周辺 $K \times 1$ 画素の和を計算するフラグメントプログラムを用い（図 5(b)），K を変えながら実行時間 T_2 を計測する．ここで，$K \times 1$ 画素の参照は中心を最初に行い，残りを左端から右端へ順に行う．また，十分な並列性を持つよう大きな値を J に用い，$\mathcal{I}$ の各画素にはランダム値を格納する．なお，計測対象は

表 2 実験環境
Table 2 Experimental environment.

GPU	6800 GTO（NV45）	7800 GTX（G70）	8800 GTX（G80）
ビデオメモリ容量（MB）	256	256	768
ビデオメモリバンド幅（GB/s）	28.8	51.2	86.4
フィルレート（Gpixel/s）	4.2	15.6	36.8
VP（基）	5	6	128（可変）
FP（基）	12	24	
ドライババージョン	77.77		97.44
バス	PCI Express 16X		
CPU	Pentium-4 3.0 GHz	Pentium D 3.0 GHz	Pentium-4 3.4 GHz

表 3 アプリケーションに依存するパラメータ値
Table 3 Application-specific parameter values.

アプリケーション	フラグメントプログラム	I	J	K	S
モルフォロジ[22]	morphology	1	1024^2	4〜1024	4
LU 分解[5]	copy_op	$\sum_{u=0}^{N-2} 4$	$N-u$	1	4
	max_op	$\sum_{u=0}^{N-2} \sum_{v=0}^{\lceil \log (N-u)/2 \rceil} 1$	2^v	2	
	quadtree_op	$\sum_{u=0}^{N-2} \sum_{v=0}^{\lceil \log (N+1-u) \rceil} 2$	$\lfloor (N+1-u)/2^v \rfloor$	2	
	divide_op	$\sum_{u=0}^{N-2} 1$	$N-u$	1	
	copy_op	$\sum_{u=0}^{N-2} 1$	$N-1-u$	1	
	row_op	$\sum_{u=0}^{N-2} 1$	$(N-1-u)^2$	3	

glBegin から glEnd までの部分である．

実行時間 T_2 を K ごとに計測後，横軸を転送量，縦軸を T_2 とするグラフに計測値をプロットし，最小二乗法により傾き B_2 を推定する．一方，切片 L_2 に対しては，$J=1$ かつ $K=1$ としたときの実行時間を与える．

パラメータ B_2 の計測において，周辺 $K \times 1$ 画素の和を計算する理由は 3 つある．まず，FP に tex 命令を実行させるためには，その命令が読み出す値を用いた計算が必要であること．次に，加算命令は計測に対する外乱（perturbation）が小さいこと．最後に，開発初期の GPGPU 実装において，ありがちで単純な参照パターンであること．特に，テクスチャの物理アドレスは z 字状に並んでいるため[2]，横 1 列に画素を走査すれば，連続参照と連続でない参照が交互に出現してバランスがよい．

残りのパラメータのうち，B_1 および B_3 はテクスチャのサイズを変えながら，glTexSubImage および glReadPixels に要する実行時間を計測し，最小二乗法により推定する．L_1 および L_3 に対しては，1×1 画素のテクスチャに対する各関数の実行時間を与える．

なお，転送バンド幅 B_2 を取得するとき，フラグメントプログラムが参照する画素数 K を変えながら計測することが重要である．K を固定したままテクスチャサイズ J を増やすことも可能である．しかし，K が小さな値の場合，さほどテクスチャを読み出さないプログラムの実行回数を増やすことになる．したがって，バンド幅というよりは遅延時間がメモリ読み出し性能を支配することになり，正確なバンド幅を取得できない．実際に，後述する実行環境において $K=1$ のとき，後者の取得方法は前者よりも 15〜73%ほど低い値を返した．多くの GPGPU 実装はメモリ集中型であることから B_2 を正確に取得することは重要である．

6. 実　　験

提案モデルの精度を検証するために，モルフォロジカルフィルタ[22] および LU 分解[5] を用いた実験結果を示す．また，実際の GPGPU 実装における参照パターンを分類し，提案モデルの汎用性を検証する．なお，実験で用いたアプリケーションは，規則的にデータを参照する典型的な GPGPU 実装であり，I，J および K の重み，テクスチャのフォーマットおよび参照パターンが異なる．

実験には，3 世代に渡る nVIDIA 社の GPU（NV45, G70 および G80）を用いた（**表 2**）．なお，双方の実装ともに nVIDIA 社の OpenGL 拡張を用いているため，他社の GPU では実行できない．

6.1 パラメータ値の取得

表 3 に，実装依存のパラメータ値を示す．まず，モルフォロジカルフィルタはシングルパスの実装である（$I=1$）．入力として 1024×1024 画素からなる元画像（$J=1024^2$）および K 個の黒点数を持つ構造要素を必要とし，それらの集合演算を行う．なお，描画

表 4 実験環境に依存するパラメータ値
Table 4 Machine-specific parameter values.

プログラム	GPU	ダウンロード		GPU 内計算		リードバック	
		B_1 (MB/s)	L_1 (μs)	B_2 (MB/s)	L_2 (μs)	B_3 (MB/s)	L_3 (μs)
モルフォロジ（8 ビットベクトル pbuffer[26]）	6800 GTO（NV45）	657	7.4	10,582	30.5	106	53.6
	7800 GTX（G70）	809	7.4	38,573	29.1	107	50.7
	8800 GTX（G80）	682	6.2	69,444	41.6	116	51.7
LU 分解（32 ビットスカラ FBO[27]）	6800 GTO（NV45）	544	5.4	12,005	30.7	796	58.2
	7800 GTX（G70）	743	5.9	33,333	29.5	1,020	29.9
	8800 GTX（G80）	879	6.5	71,942	30.8	1,374	41.3

処理は 8 ビットベクトル値を持つ pbuffer[26] で実装されていて（$S = 4$），ベクトル演算を用いる．実験では，K を 4〜1024 に変えながら実行時間を計測した．

一方，LU 分解は 5 種類のフラグメントプログラムで構成されていて，部分ピボット選択付の実装である．詳細は文献 5) を参照されたい．描画処理は，32 ビットスカラ値を持つ FBO[27]（Frame Buffer Object）で実装されている（$S = 4$）．実験では，行列サイズ N を 128〜2048 まで変えながら実行時間を計測した．

両実装の特徴を比較すると，モルフォロジカルフィルタは J および K の値が大きく，一度に多数の画素を参照する描画を 1 回のみ行う．一方，LU 分解ではこれらの値は小さいが，I の値が大きい．つまり，少ない画素を参照する描画を繰り返す．また，複数のフラグメントプログラムで構成されていて，各々の参照パターンが異なる（6.3 節）．

表 4 に，実行環境に依存するパラメータ値を示す．これらの計測には，5.3 節で示した計測プログラムを用いた．この際，テクスチャのサイズは 1024×1024 画素とし，K を 8〜16 に変えながら B_2 を推定した．また，B_1 および B_3 に対しては，テクスチャのサイズを 1024×256 から 1024×2048 画素まで変えながら計測した．なお，計測はナノ秒単位であり，Win32 関数の QueryPerformanceCounter および QueryPerformanceFrequency を用いた．

実装間の大きな差異は，リードバック時の転送バンド幅 B_3 である．PCI Express バスの理論値が 4GB/s であるにも関わらず，主記憶への転送速度が遅いことは GPGPU 実装の課題であったが，FBO により改善されていることが分かる．

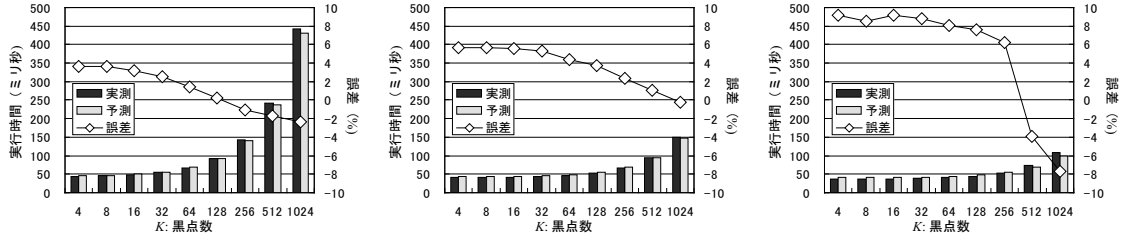
6.2 モデル精度の検証

図 6 に，モルフォロジカルフィルタに対する実行時間 T の実測値 T_m，予測値 T_p およびそれらの誤差 $100(T_p/T_m - 1)$ を示す．なお，CPU 版の実行時間は $K = 1024$ のときにおよそ 2 秒である．GPU 版はおよそ 450 ミリ秒であるため（図 6(a)），CPU 版と比較して 4 倍ほど高速に動作している．

すべての実行環境において，提案モデルは 10%以内の誤差を達成できている．なお，誤差は K の増大とともに減少している．この理由は，**表 5** に示す T の内訳により説明できる．表では，K が増大するにつれ，性能ボトルネックが T_3 から T_2 に移っている．さらに，T_3 の実測値が予測値よりも短い一方，T_2 では逆に実測値の方が長い．結果，K の増大とともに誤差の主因が T_3 から T_2 に移り，誤差が減少していた．なお，T_2 の実測値が長い理由は，計測プログラムとの参照パターンの相違にある．また，T_3 の実測値が短い理由は計算結果の内容にある．モルフォロジカルフィルタは 2 値画像を出力していて，その計算結果は同じ値を多く含んでいる．そのような 2 値データに対し，GPU は転送の一部を省略している可能性が高く，その実測値はランダムデータの転送で得た予測値よりも短い．

なお，$K = 1024$ として，古い NV45 から順に実行環境を比較すると，B_2 を 3.6 倍（G70）および 6.6 倍（G80）にすることにより，実測は 3.0 倍および 4.1 倍の速度向上を果たす一方，提案モデルもこれらに近い 2.9 倍および 4.3 倍の速度向上を予測できている．したがって，将来の GPU における性能を見積もる際にも提案モデルが有用である可能性がある．

図 7 に，LU 分解に対する予測結果を示す．モルフォロジカルフィルタと同様に，G70 および G80 に対しては 10%以内の誤差を達成できている．しかし，NV45 において行列サイズ N を増大させたときに誤差が最大で 20% に達した．この理由は，L1 キャッシュの効果にあると考えられる．**表 6** の内訳を見ると，5 種類のプログラムのうち row_op において誤差が大きい．row_op では，行 x を固定して列 y を変えながら画素 (x, y)，$(x, 0)$ および $(0, y)$ を参照する．このとき，2 回目以降の実行では $(x, 0)$ を FP 内の L1 キャッシュから読み出せる．一方，パラメータ取得のためのプログラムは，L1 キャッシュから読み出せる参照パターンを持たないために，row_op に対する予測値が実測値よりも長くなった．実際に，row_op の参照パターン



(a) 6800 GTO（NV45） (b) 7800 GTX（G70） (c) 8800 GTX（G80）

図 6 モルフォロジカルフィルタに対する予測結果
Fig. 6 Prediction results for morphological filter.

表 5 モルフォロジカルフィルタに対する予測結果の内訳
Table 5 Breakdown of prediction results for morphological filter.

GPU	内訳	$K=4$		$K=8$		$K=16$		$K=32$		$K=64$		$K=128$		$K=256$		$K=512$		$K=1024$	
		実測	予測	実測	予測	実測	予測	実測	予測	実測	予測	実測	予測	実測	予測	実測	予測	実測	予測
6800 GTO（NV45）	T_1	6.0	6.1	6.0	6.1	5.9	6.1	6.0	6.1	5.9	6.1	5.9	6.1	5.9	6.1	6.0	6.1	6.1	6.1
	T_2	1.5	1.5	3.1	3.1	6.3	6.1	12.6	12.1	25.1	24.2	50.0	48.4	99.8	96.8	199.6	193.6	399.0	387.1
	T_3	34.9	38.0	34.8	38.0	34.8	38.0	34.7	38.0	34.7	38.0	34.8	38.0	34.9	38.0	34.6	38.0	34.7	38.0
	T	44.0	45.6	45.5	47.1	48.6	50.1	54.8	56.2	67.3	68.3	92.3	92.5	142.3	140.9	241.8	237.6	441.6	431.2
7800 GTX（G70）	T_1	5.0	4.9	5.0	4.9	5.1	4.9	5.0	4.9	5.1	4.9	5.1	4.9	5.1	4.9	5.1	4.9	5.2	4.9
	T_2	0.5	0.4	0.9	0.9	1.8	1.7	3.6	3.3	7.1	6.7	13.8	13.3	27.5	26.6	54.8	53.1	109.5	106.2
	T_3	34.2	37.3	34.2	37.3	34.1	37.3	34.1	37.3	34.1	37.3	34.1	37.3	34.1	37.3	33.8	37.3	33.5	37.3
	T	40.4	42.7	40.8	43.1	41.6	43.9	43.3	45.6	46.9	48.9	53.5	55.5	67.2	68.8	94.3	95.3	148.8	148.4
8800 GTX（G80）	T_1	5.7	5.9	5.9	5.9	5.7	5.9	5.8	5.9	6.0	5.9	5.7	5.9	5.9	5.9	5.7	5.9	5.9	5.9
	T_2	0.3	0.3	0.5	0.5	1.1	1.0	2.0	1.9	3.9	3.7	7.6	7.4	15.0	14.8	36.2	29.5	71.3	59.0
	T_3	30.5	34.5	30.4	34.5	30.4	34.5	30.3	34.5	30.3	34.5	30.3	34.5	30.2	34.5	30.0	34.5	29.7	34.5
	T	37.2	40.6	37.7	40.9	37.8	41.3	38.8	42.2	40.8	44.1	44.4	47.8	51.9	55.1	72.7	69.9	107.7	99.4

でパラメータ値を再取得したところ，誤差は 20%から 5%に減少したため，誤差を増大させている理由は参照パターンの違いにある．

なお，より高い B_2 を持つ G70 および G80 では L1 キャッシュの効果が弱まることから，誤差は相対的に小さいと考えられる．また，L1 キャッシュのヒット率は N とともに増大するため，その効果も N とともに向上する．したがって，効果を隠蔽する提案モデルの誤差は N とともに増大している（図 7）．

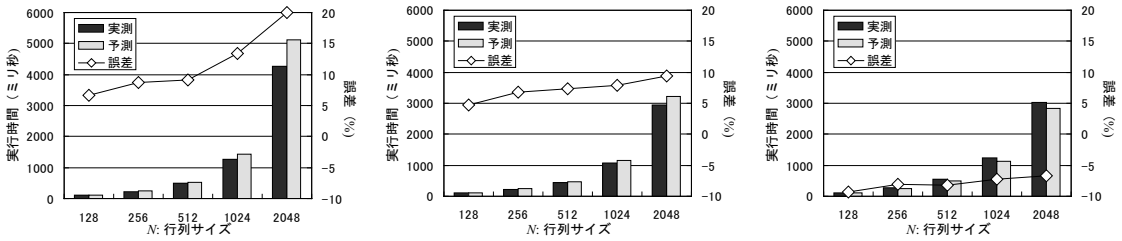
G80 を G70 と比較すると，G80 は 2.2 倍の B_2 を持つにも関わらず，実測の実行時間が 80 ミリ秒ほど長い．一方，提案モデルも 400 ミリ秒ほど長いことを予測している．この理由は G80 における L_2 の長さにある．表 6 を見ると，B_2 の向上により G80 は row_op を短縮できているものの，残りの部分は逆に長い．これらに共通する特徴は，J が小さい一方で I が大きい点である（表 3）．つまり，計算量の少ない描画を多く繰り返している．この場合，式 (1) が示すように，T において L_2 が支配的になり，長い L_2 を持つ G80 は性能を低下させていた．

一方，NV45 に着目すると，row_op の実行時間が T の 55%を占めているため，B_2 を向上することは T の短縮に有効である．このように LU 分解の実装では，B_2 の向上とともに性能ボトルネックが移り変わっていて，今後は短い L_2 を持つ GPU あるいは長い L_2 を隠蔽できるアルゴリズムやデータ構造の開発が必要である．

6.3 汎用性の検証

図 8 に，逐次参照およびランダム参照におけるバンド幅 B_2 を示す．計測には GPUBench[28] を用いた．なお，逐次参照は 4 枚のテクスチャを入力として，各画素を積和しながら出力する．また，ランダム参照は 2 枚のテクスチャを用いた間接参照により実現している．つまり，片方のテクスチャにアドレスをデータとして格納し，そのアドレスを基にもう一方のテクスチャを参照する．双方の参照パターンにおいて $K=4$ である．

逐次参照およびランダム参照を比較すると，後者は前者の 1/6〜1/3 程度のバンド幅である．ここで，逐次参照と同様に，計測プログラムの参照パターンは規則性を持つことから，参照パターンを隠蔽する提案モデルは，ランダム参照を多く含み T_2 が性能ボトルネッ



(a) 6800 GTO（NV45） (b) 7800 GTX（G70） (c) 8800 GTX（G80）

図 7 LU 分解に対する予測結果
Fig. 7 Prediction results for LU decomposition.

表 6 LU 分解に対する予測結果の内訳（単位：ミリ秒）
Table 6 Breakdown of prediction results for LU decomposition.

GPU	内訳	$N=128$		$N=256$		$N=512$		$N=1024$		$N=2048$	
		実測	予測	実測	予測	実測	予測	実測	予測	実測	予測
6800 GTO（NV45）	T_1	0.2	0.1	0.6	0.5	2.3	1.8	7.7	7.4	30.1	29.4
	copy_op	20.1	19.6	40.4	39.3	80.7	78.7	161.4	157.6	318.2	315.3
	max_op	28.8	27.7	64.7	63.2	131.2	126.4	291.2	284.3	654.0	631.8
	quadtree_op	46.6	55.3	104.9	126.4	211.1	252.8	472.2	568.7	1047.7	1263.6
	divide_op	4.3	3.9	8.5	7.9	17.2	15.8	34.0	31.6	68.3	63.1
	row_op	4.1	4.6	11.2	13.3	46.5	59.5	286.1	382.8	2120.7	2875.6
	T_2（小計）	104.0	111.1	229.6	250.1	486.6	533.2	1244.8	1424.9	4208.8	5149.4
	T_3	0.2	0.1	0.5	0.4	1.4	1.3	5.2	5.1	20.6	20.1
	T	104.3	111.3	230.6	250.9	490.3	536.3	1257.8	1437.3	4259.4	5199.0
7800 GTX（G70）	T_1	0.1	0.1	0.4	0.3	1.7	1.4	5.8	5.4	22.3	21.5
	copy_op	19.3	18.5	38.6	37.1	75.8	74.4	153.5	148.9	302.5	297.9
	max_op	27.6	26.2	62.0	59.8	124.5	119.6	279.1	269.1	626.0	597.9
	quadtree_op	45.0	52.4	101.5	119.6	202.2	239.2	457.0	538.2	1011.8	1195.8
	divide_op	4.1	3.7	8.0	7.4	16.2	14.9	32.4	29.9	64.9	59.8
	row_op	4.0	4.0	8.3	9.7	26.5	32.6	133.2	170.5	910.7	1183.6
	T_2（小計）	100.0	104.8	218.5	233.7	445.1	480.7	1055.1	1156.5	2915.8	3335.0
	T_3	0.1	0.1	0.3	0.3	1.0	1.0	3.9	4.0	15.4	15.7
	T	100.2	105.0	219.2	234.3	447.8	483.1	1064.8	1165.8	2953.5	3372.3
8800 GTX（G80）	T_1	0.1	0.1	0.3	0.3	1.2	1.1	4.7	4.6	17.9	18.2
	copy_op	26.0	19.6	52.2	39.3	104.4	78.8	208.6	157.8	418.4	315.7
	max_op	36.7	27.8	83.5	63.4	167.1	126.8	374.8	285.3	834.2	634.0
	quadtree_op	50.3	55.5	114.0	126.9	228.6	253.7	511.9	570.6	1138.4	1268.0
	divide_op	5.2	3.9	10.3	7.9	20.9	15.8	41.5	31.7	82.9	63.4
	row_op	3.9	4.0	7.9	8.8	21.4	23.1	86.5	90.8	530.7	537.7
	T_2（小計）	122.1	110.8	267.9	246.3	542.4	498.2	1223.2	1136.2	3004.6	2818.8
	T_3	0.1	0.1	0.3	0.2	0.8	0.8	3.0	3.0	11.6	11.7
	T	122.4	111.0	268.6	246.8	544.5	500.1	1230.9	1143.7	3034.1	2848.7

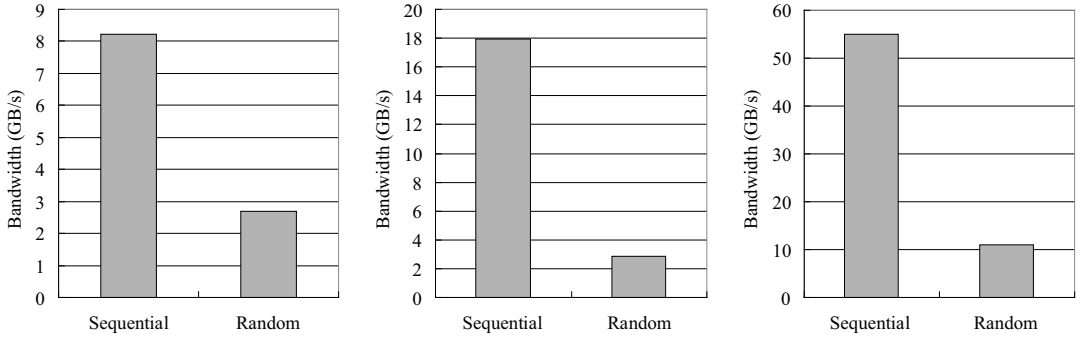
クとなる GPGPU 実装を精度よく見積もれない．したがって，提案モデルの汎用性は，GPGPU 実装における参照パターンの分布に依存する．

表 7 に，図 4 に挙げた GPGPU 実装に対する参照パターンの分類を示す．対称参照およびランダム参照を除けば，すべての参照パターンが LU 分解およびモルフォロジカルフィルタのいずれかに分類できている．また，対称参照を含め，これらの参照パターンは規則性があり連続的なものが多い．したがって，この分類および精度検証の結果より，提案モデルは規則的にデータを参照する典型的な GPGPU 実装に対し，GPU による高速化の見込みを精度よく見積もれると考える．

なお，1 章で述べたように，制約の強い GPU アーキテクチャが参照パターンを規則的なものに変更してしまう可能性はある．

7. ま と め

本稿では，典型的な GPGPU 実装を対象として，その性能を予測するための性能モデルを提案した．提案モデルは，多くの GPGPU 実装がメモリバンド幅ネッ



(a) 6800 GTO（NV45） (b) 7800 GTX（G70） (c) 8800 GTX（G80）

図 8 参照パターンごとの転送バンド幅 B_2
Fig. 8 Bandwidth B_2 with different access patterns.

表 7 GPGPU 実装[5),7),9),11),16),21),22),28)] における参照パターンの分類
Table 7 Classification of access patterns in GPGPU implementations[5),7),9),11),16),21),22),28)].

分類	画素 (x, y) における参照場所	実装	フラグメントプログラム	描画領域
単一	特定画素 (X, Y)*	LU 分解[5)]	divide_op	1 次元
自身	自画素 (x, y)	LU 分解	copy_op	1 次元
		CG[21)]	Copy SelfComponentMultiply	
		GPUFFTW[16)]	copy	2 次元
		GPU-Sort[9)]	laststepfptext	
先頭	先頭画素 $(x, 0)$，$(0, y)$ および (x, y)	LU 分解	row_op	2 次元
相対	相対画素 $(x + X, y + Y)$ および (x, y)	LU 分解	max_op	1 次元
		CG	sumReduction	
		GPUFFTW	copy 以外の 5 つ	2 次元
周辺	上下左右領域 $\{(x + x', y + y') \mid -X \leq x', y' \leq X\}$	測定プログラム		1 次元
		モルフォロジカルフィルタ[22)]		2 次元
	左上領域 $(x - 1, y)$，$(x - 1, y + 1)$，$(x, y + 1)$	Alignment[7)]		2 次元
	右領域 $\{(4x + x', y) \mid 0 \leq x' \leq X\}$	GPU-Sort	lfillr lfillg lfillb lfilla	2 次元
複数	複数テクスチャ $\mathcal{I}_1, \ldots, \mathcal{I}_n$ 上の自身 $\mathcal{I}_1(x, y), \ldots, \mathcal{I}_n(x, y)$	LU 分解	quadtree_op	1 次元 $(n = 2)$
		CG	scaledAdd ComponentMultiply	
		GPUBench[28)]	sequential	2 次元 $(n = 4)$
		Clustering[11)]		
対称	左右対称画素 $(X - x, y)$ および (x, y)	GPU-Sort	minfptext maxfptest	2 次元
	対角対称画素 $(X - x, Y - y)$ および (x, y)		Combine_two_channels lastminusonestepfptext	
ランダム	間接参照 $\mathcal{I}_1(\mathcal{I}_2(x, y), \mathcal{I}_3(x', y'))$	CG	MatrixVectorMultiplication	1 次元
		GPUBench	random	2 次元

*：X および Y は定数

クであり，規則的にデータを参照することに着目し，主記憶，ビデオメモリおよび FP 間のデータパスの性能で実装全体の性能をモデル化する．各データパスの性能はバンド幅および遅延時間だけの単純な組で表す．提案モデルは，GPGPU アプリケーションとは独立に計測できるパラメータを用いることにより，プログラム開発前の予測を可能にする．

提案モデルを画像フィルタおよび LU 分解に適用し，3 世代に渡る GPU 上で評価したところ，誤差は最悪で 20%であった．GPU 内キャッシュの効果がさほど大きくない場合，誤差は 10%以内である．また，ランダム参照に対しては，よい精度を提供できない可能性

が高い．したがって，提案モデルは規則的にデータを参照する典型的な実装に対し，GPU による高速化の見込みを見積もる際に有用であると考える．

今後の課題としては，より高度な実装への対応が挙げられる．例えば，5.2 節で挙げたパイプライン型のデータ転送を用いる実装や，CPU および GPU の双方を用いてデータ並列計算を行う実装[6] が候補である．

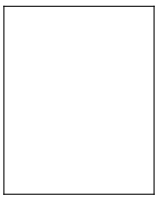
謝辞 本研究の一部は，科学研究費補助金基盤研究（B）（2）（18300009）および特定領域研究（17032007）の補助による．有益な御意見を頂いた査読者の方々に深く感謝致します．

参 考 文 献

1) Pharr, M. and Fernando, R.(eds.): *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA (2005).
2) Montrym, J. and Moreton, H.: The GeForce 6800, *IEEE Micro*, Vol.25, No.2, pp.41–51 (2005).
3) 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).
4) Fatahalian, K., Sugerman, J. and Hanrahan, P.: Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication, *Proc. 19th SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware (GH'04)*, pp.133–137 (2004).
5) Galoppo, N., Govindaraju, N.K., Henson, M. and Manocha, D.: LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'05)* (2005). 12 pages (CD-ROM).
6) 大島聡史，吉瀬謙二，片桐孝洋，弓場敏嗣：CPU と GPU を用いた並列 GEMM 演算の提案と実装，情報処理学会：コンピューティングシステム，Vol.47, No.SIG12(ACS15), pp.317–328 (2006).
7) Liu, W., Schmidt, B., Voss, G. and Müller-Wittig, W.: Streaming Algorithms for Biological Sequence Alignment on GPUs, *IEEE Trans. Parallel and Distributed Systems*, (to appear).
8) Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M.C. and Manocha, D.: Fast Computation of Database Operations using Graphics Processors, *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD'04)*, pp.215–226 (2004).
9) Govindaraju, N.K., Raghuvanshi, N., Henson, M., Tuft, D. and Manocha, D.: A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors, Technical Report TR05-016, University of North Carolina-Chapel Hill (2005).
10) Greß, A. and Zachmann, G.: GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures, *Proc. 20th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'06)* (2006). 10 pages (CD-ROM).
11) Takizawa, H. and Kobayashi, H.: Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing, *The J. Supercomputing*, Vol.36, No.3, pp.219–234 (2006).
12) Klimovitski, A.: Using SSE and SSE2: Misconceptions and Reality, *Intel Developer Update Magazine* (2001).
13) Khailany, B., Dally, W.J., Kapasi, U.J., Mattson, P., Namkoong, J., Owens, J.D., Towles, B., Chang, A. and Rixner, S.: IMAGINE: MEDIA PROCESSING WITH STREAMS, *IEEE Micro*, Vol.21, No.2, pp.35–46 (2001).
14) Shreiner, D., Woo, M., Neider, J. and Davis, T.: *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, fifth edition (2005).
15) Microsoft Corporation: DirectX, Asm Shader Reference (2005). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/reference.asp.
16) Govindaraju, N.K., Larsen, S., Gray, J. and Manocha, D.: A Memory Model for Scientific Algorihms on Graphics Processors, *Proc. High Performance Networking and Computing Conf. (SC'06)* (2006). 10 pages (CD-ROM).
17) Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware, *ACM Trans. Graphics*, Vol.23, No.3, pp.777–786 (2004).
18) Sheaffer, J.W., Luebke, D. and Skadron, K.: A Flexible Simulation Framework for Graphics Architectures, *Proc. 19th SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware (GH'04)*, pp.85–94 (2004).
19) Mark, W.R., Glanville, R.S., Akeley, K. and Kilgard, M.J.: Cg: A system for programming graphics hardware in a C-like language, *ACM Trans. Graphics*, Vol.22, No.3, pp.896–897 (2003).
20) Rost, R.J.: *OpenGL Shading Language*,

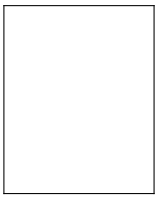
Addison-Wesley, Reading, MA, second edition (2006).
21) Corrigan, A.: Implementation of Conjugate Gradients (CG) on Programmable Graphics Hardware (GPU) (2005). http://www.cs.stevens.edu/%7Equynh/student-work/acorrigan_gpu.htm.
22) 前野滝授，伊野文彦，萩原兼一：モルフォロジー演算の高速化のための GPU 実装の比較，第 10 回 Visual Computing ／グラフィクスと CAD 合同シンポジウム予稿集，pp.239–244 (2006).
23) nVIDIA Corporation: Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL, Technical Brief TB-02011-001_v01, nVIDIA Corporation (2005). http://download.nvidia.com/developer/Papers/2005/Fast_Texture_Transfers /Fast_Texture_Transfers.pdf.
24) Xu, F. and Mueller, K.: Ultra-Fast 3D Filetered Backprojection on Commodity Graphics Hardware, *Proc. 1st IEEE Int'l Symp. Biomedical Imaging (ISBI'04)*, pp.571–574 (2004).
25) Ujaldon, M. and Saltz, J.: Exploiting parallelism on irregular applications using the GPU, *Proc. Int'l Conf. Parallel Computing (ParCo'05)*, pp.639–646 (2005).
26) OpenGL Extension Registry: WGL_ARB_pbuffer (2002). http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pbuffer.txt.
27) OpenGL Extension Registry: GL_EXT_framebuffer_object (2006). http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt.
28) Buck, I., Fatahalian, K. and Hanrahan, P.: GPUBench: Evaluating GPU Performance for Numerical and Scientific Applications, *Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^2'04)*, p.C-20 (2004).

(平成 19 年 1 月 22 日受付)
(平成 19 年 4 月 26 日採録)



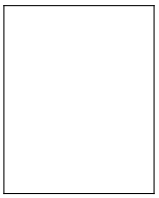
伊藤　信悟

平成 17 年大阪大学基礎工学部情報科学科中退．平成 19 年同大学院情報科学研究科修士課程修了．現在，株式会社カプコン勤務．在学中は，GPU を用いた画像処理の高速化に関する研究に従事．



伊野　文彦（正会員）

平成 10 年大阪大学基礎工学部情報工学科卒業．平成 12 年同大学院基礎工学研究科修士課程修了．平成 14 年同大学院同研究科博士課程中退．同年，同大学助手．博士（情報科学）．平成 15 年国際会議 HiPC'03 最優秀論文賞，平成 16 年先進的計算基盤システムシンポジウム SACSIS'04 最優秀論文賞受賞．並列計算機の応用およびソフトウェア開発環境に関する研究に従事．



萩原　兼一（正会員）

昭和 49 年大阪大学基礎工学部情報工学科卒業．昭和 54 年同大学院基礎工学研究科博士課程修了．工学博士．同大学助手，講師，助教授を経て，平成 5 年奈良先端科学技術大学院大学教授．平成 6 年より大阪大学教授．平成 4〜5 年文部省在外研究員（米国メリーランド大）．平成 15 年国際会議 HiPC'03 最優秀論文賞，平成 16 年先進的計算基盤システムシンポジウム SACSIS'04 最優秀論文賞受賞．現在，並列処理の基礎および応用に興味を持っている．