GPUアプリケーションを高速化するための命令割当て方式

池田孝利† 伊野文彦† 萩原兼一†

近年，GPU（Graphics Processing Unit）は急激に性能を向上していて，その内部プロセッサ（頂点プロセッサ VP とフラグメントプロセッサ FP）はプログラム可能になっている．そこで，描画処理を目的としてきた GPU を，汎用計算の用途に応用する取り組み（GPGPU：General-Purpose computation on GPUs）が始まっている．GPU 内部において VP と FP がパイプラインを構成しているにもかかわらず，多くの GPGPU プログラムは FP のみを多用している．このような実装では，FP の計算負荷が高くなり，パイプライン実行における性能ボトルネックになりうる．本研究の目的は，この性能ボトルネックを解消し，GPGPU プログラムの性能を向上させることにある．その実現のために，本稿では，VP と FP のアセンブリ言語プログラムが与えられた場合，FP プログラムにおける命令の一部を VP プログラムへ移動する手法を提案する．提案手法は，CPU と GPU 間の入出力仕様を変えることなく命令を移動するための条件を明らかにし，VP および FP における計算資源の制限を考慮して命令を移動する．提案手法をガウスフィルタプログラムに適用した結果，GPU における計算時間をおよそ 19 ミリ秒から 11 ミリ秒に短縮でき，その性能を向上させることができた．

An Instruction Allocation Method for Accelerating GPU Applications

Takatoshi Ikeda,† Fumihiko Ino† and Kenichi Hagihara†

Recently, graphics processing units (GPUs) are providing increasingly higher performance with programmable internal processors, namely vertex processors (VPs) and fragment processors (FPs). Such newly added capabilities are driving us to perform general-purpose computation on GPUs (GPGPU), in addition to traditional visualization. Although VPs and FPs are structured in a pipeline, many GPGPU implementations utilize only FPs. Therefore, such implementations may result in lower performance due to the highly loaded FPs, namely the performance bottleneck in the pipeline execution. The objective of our work is to improve the performance of GPGPU programs by eliminating this bottleneck. To achieve this, this paper presents an instruction allocation method that is capable of reducing the FP workload by moving some instructions from the FP program to the VP program, both written in an assembly language. Our method makes clear the requirements for moving such instructions without changing I/O specification between the CPU and the GPU. To achieve higher performance, we also consider the limitation on resources available on VPs and FPs. As a result of applying it to a Gaussian filter program, we find that our method successfully improves the performance with reducing the GPU execution time from approximately 19 ms to 11 ms.

1. はじめに

GPU（Graphics Processing Unit）[1] とは描画処理の高速化を目的とした処理装置である．一般に，GPU はパイプラインアーキテクチャに基づいている（**図 1**）．このパイプラインは，VP（Vertex Processor）および FP（Fragment Processor）と呼ばれる 2 種類のプロセッサを直列に並べた構造を持つ．

近年，ムーアの法則を上回る速度で GPU の性能が向上していて，1 秒間あたりの浮動小数点演算数（FLOPS）に関して CPU の性能を上回るようになった．この魅力的な性能に加え，最近の機能拡張により，プログラム可能な VP および FP，IEEE754 標準の 32 ビット浮動小数点演算，および分岐命令などを利用できるようになったことが，描画用途に特化してきた GPU を，計算量の多い他の用途へ駆り立てている．なお，現在の GPU は，最長で 32 ビットの浮動小数点を扱えるが，汎用計算の用途としては，この演算精度は十分ではない．しかし，高い精度を要求しない問題や，精度を犠牲にしてでも高速処理することが望ましい問題に対しては，GPU の有用性は高い．

GPU を様々な用途へ応用する研究は従来から多く存在する．これらの研究は，(a) 3 次元グラフィクス

† 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

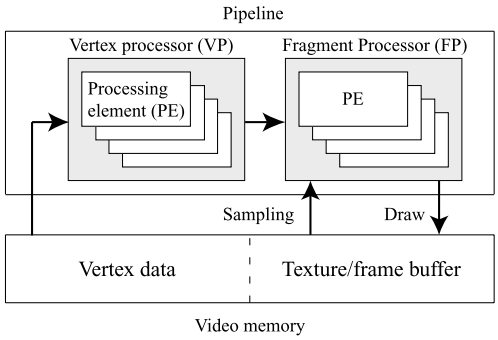

図 1 GPU 内部のパイプラインアーキテクチャ
Fig. 1 GPU pipeline architecture.

レンダリングによる解法，および（b）プログラム可能機能を用いて CPU プログラムを GPU へ適応させる解法に分類できる．前者の例としては，物体間の衝突判定問題[2),3)] やデータのクラスタリング問題[4)] への応用があげられ，後者の例としては，物理シミュレーション[5),6)] や数値計算[7)~10)] への応用がある．

しかし，これらの大半は CPU 実装に対する性能向上を目的としていて，GPU の持つ性能をどの程度効率良く引き出せているのかは明らかでない．特に，(b) に基づく解法では，VP および FP のうち，FP のみを用いるものが大半であり，GPU の性能を十分に引き出せているとはいえない．このような実装においては，計算負荷の高い FP がプログラムの性能ボトルネックになる可能性がある．

そこで本研究では，GPU プログラムを高速化するために，この性能ボトルネックを解消することを目指す．これを実現するために，FP における命令の一部を VP へ割り当てることで，FP における計算負荷を軽減する手法を提案する．提案する命令移動手法は，VP および FP のアセンブリ言語プログラムが与えられたとき，GPU に対する入出力仕様を変更することなく GPU プログラムの高速化を図る．

以降では，まず 2 章で GPU のアーキテクチャについて述べ，(b) プログラム可能機能を用いる解法をまとめる．次に，3 章で提案手法について述べ，4 章でその適用実験の結果を示す．その後，5 章で関連研究を紹介し，6 章で本稿をまとめる．

2. GPU による汎用計算

本章では，GPU を用いた汎用計算の実現について述べる．まず，現在広く使われている GPU のアーキテクチャの概要を示し，そのプログラミング方法について述べる．

なお，本研究では VP および FP の仕様として，多くの GPU で使用でき，比較的新しい VS1.1 [11),12)] および PS2.0 [11),12)] を前提とする．

表 1 VP レジスタ一覧（∗ 印はスカラレジスタを表す）
Table 1 Registers in vertex processors (VPs).

種類	名前	入出力許可
R_{vi}：VP 入力レジスタ	v0〜v15	Read only
R_{vc}：定数レジスタ	c0〜c95	Read only
R_{vt}：作業用レジスタ	r0〜r11	Read/write
R_{vo}：VP 出力レジスタ	oD0，oD1，oT0〜oT7，oFog∗，oPos，oPts∗	Write only
R_{va}：アドレスレジスタ	a0∗	Read/write

表 2 FP レジスタ一覧（∗ 印はスカラレジスタを表す）
Table 2 Registes in fragment processors (FPs).

種類	名前	入出力許可
R_{fi}：FP 入力レジスタ	v0，v1，t0〜t7	Read only
R_{fc}：定数レジスタ	c0〜c31	Read only
R_{ft}：作業用レジスタ	r0〜r11	Read/write
R_{fs}：サンプラレジスタ	s0〜s15	Read only
R_{fo}：FP 出力レジスタ	oC0〜oC3，oDepth∗	Write only

2.1 GPU のパイプラインアーキテクチャ

図 1 に，GPU のパイプライン構造を示す．GPU のパイプラインは，プログラム可能な 2 つのプロセッサを持ち，前段を VP，後段を FP と呼ぶ．**表 1** および **表 2** に，各々が保持する内部レジスタの一覧を示す．以降では，これらを用いて VP および FP を説明し，VP および FP 間の接続形態を示す．

VP VP は，3 次元座標系での頂点座標演算処理を担う MIMD 型プロセッサである．この処理を高速化するために，VP は複数個のベクトル演算器を要素（PE：Processing Element）として持ち，各演算器は 4×4 行列に対する演算を高速処理できる．これにともない，その内部レジスタはベクトル構造を持ち，4 個のスカラ値を 1 個のベクトルとして保持する．なお，このベクトルレジスタはスカラレジスタとしても扱える．

VP を用いて演算を行うために，CPU は，演算対象の頂点データを頂点配列[13)] としてビデオメモリ上にあらかじめ用意する必要がある．GPU はその配列から n 個ずつ，各々の頂点データを VP 入力レジスタ R_{vi} に格納し VP プログラムを実行する．ここで，n の値は CPU が指定する．VP は，内部レジスタを用いて計算した結果を VP 出力レジスタ R_{vo} へ格納して処理を終える．

このとき，R_{vo} のうち，oD0 および oD1 は $[0, 1]$，oT0〜oT7 は $[L, H]$（L および H は有効なテク

スチャのアドレスの下限および上限）の範囲に収まるよう，範囲内の最近値に置き換えられる．なお，VP 自身は R_{vo} を読み出せない．

VP 出力レジスタ R_{vo}（oD0，oD1，oT0〜oT7）の各々は，後段の FP における FP 入力レジスタ R_{fi}（v0，v1，t0〜t7）に接続していて，FP への入力の一部となる．

VP と FP の接続形態 VP が 3 次元座標系で処理した結果を基に，GPU は FP で処理する 2 次元領域を決定する．この領域は，互いに独立に処理できる画素としてラスタライズされ，それらの画素の座標が FP への入力となる．このとき，VP の演算対象が頂点である一方，FP の演算対象はその頂点が囲む領域内の画素であるため，R_{fi} は画素の位置を基に R_{vo} を線形補間したものになる．この接続形態は，並列性を引き出すためにループを展開するループアンローリング手法[14] に似たものと見なせる．つまり，画素に関するループを FP プログラムとして与えるのではなく，そのループを展開した部分（各画素ごとの処理）を与えることで，高い並列性を引き出す．

FP FP は，2 次元座標系での色値演算処理を担う SIMD 型プロセッサである．この処理を高速化するために，FP は VP と同等のベクトル演算器に加え，VP にはないテクスチャサンプラ機能を持つ．この機能により，FP は，CPU がビデオメモリ上に配置したテクスチャ[15] を取得できる．したがって，FP はテクスチャデータを入力として扱える点で VP と異なる．また，FP は大量のデータを単純かつ高速に処理する必要があるため，FP は分岐命令を実行できない．

FP を用いて演算を行うために，GPU は，VP の出力を補間したものを FP 入力レジスタ R_{fi} に格納し，FP プログラムを実行する．FP では VP 同様，内部レジスタを用いて処理した結果を FP 出力レジスタ R_{fo} へ格納する．この結果は描画データとしてビデオメモリに出力される．

2.2 GPU のアセンブリ言語

GPU では，アセンブリ言語および GPU プログラミング専用の高級言語を使用できる．本研究では，アセンブリ言語で記述したプログラムを対象とする．

以下に，アセンブリ言語における命令の例を示す．

```
add  r1, r0, c0
```

ここで add はオペコード，r1 は出力オペランド，r0 と c0 は入力オペランドである．オペランドには，ベクトルレジスタ全体あるいはその一部を指定できる．後者の場合，添字（x，y，z，w またはこれらの組合せ）をレジスタに付記する．

VP および FP プログラムの長さは，プログラム中のアセンブリ命令が消費するスロットの数で測る．GPU におけるアセンブリ命令の大半は 1 スロットを消費するが，超越関数命令のように 10 スロットを消費するものもある．また，テクスチャデータにアクセスする命令は，状況によりスロット数が変動する．

プログラムが必要とするスロット数は，必ずしもその実行時間に比例しないが，実行時間の概略値として使用できる．VP および FP が扱えるスロット数は，それぞれの仕様で規定されている．本実験で使用した GPU では，各々の上限はそれぞれ 128 および 96 スロットである．ただし，FP では，算術命令およびテクスチャデータへのアクセス命令は 64 および 32 スロット以下である必要がある．

2.3 VP および FP のプログラミング

GPU を用いて問題を解くとき，VP および FP が担当する部分を分けて，全体の処理を実現する必要がある．現在，実装および設計の容易さから広く使用されている実装手法[16] では，GPU 内部の SIMD アーキテクチャ部をストリームプロセッサと見なす．この手法では，対象の問題から核となる処理（カーネルと呼ぶ）を抽出し，その処理を FP を主体として実装する．具体的には，FP のテクスチャサンプラ機能で処理対象データを読み取り，FP で処理して出力する．しかし，FP に処理が集中し VP の担当部分はほとんどないため，VP の計算負荷は軽い．

図 2 および **図 3** に，この実装手法に基づく VP および FP プログラムの例を示す．この例では，CPU が用意するデータ配列に対して，畳み込み演算を処理する．この実装では，データ配列を 3 次元空間の四角形として GPU に処理させる．VP ではこの四角形の 4 つの頂点を 3 次元空間座標として処理し，FP では 2 次元で四角形を塗りつぶす作業として処理する．

3. 提案する高速化手法

本章では，FP の負荷を VP へ移動することで GPU プログラムを高速化する手法について述べる．

3.1 高速化の方針

FP の負荷を VP へ移動するために，FP プログラムにおける命令の一部を VP プログラムへ移行する．まず，FP でのみ実行できて VP で実行できない命令が存在することに着目し，VP で実行できる命令を FP プログラムから抽出する．次に，抽出した命令のうち，移動の前後で CPU との入出力仕様を変更せず，かつ

```
0001  mov  oPos,   v0
0002  mov  oT0.xy, v1
```

図 2 VP プログラムの例（畳込み演算）
Fig. 2 Example of VP program for convolution.

```
0001  mov   r0.z,  t0.y
0002  mov   r1.y,  c4.x
0003  mov   r0.y,  c4.x
0004  rcp   r0.w,  c3.x
0005  mov   r0.x,  -r0.w
0006  add   r1.x,  r0.w,  r0.w
0007  add   r0.xy, r0,    t0
0008  add   r1.x,  r1.x,  r0.x
0009  add   r1.y,  r0.z,  r1.y
0010  texld r2,    r0,    s0
0011  texld r0,    r1,    s0
0012  texld r1,    t0,    s0
0013  mul   r2,    r2,    c0.x
0014  mad   r1,    c1.x,  r1,  r2
0015  mad   r0,    c2.x,  r0,  r1
0016  mov   oC0,   r0
```

図 3 FP プログラムの例（畳込み演算）
Fig. 3 Example of FP program for convolution.

正しく動作できるものを選択し，VP および FP プログラムを修正する．

提案手法により FP の命令数は減少するため，高速化が期待できる．しかし，VP の命令数と VP から FP へ出力するデータ数は増加するため，それらに起因するオーバヘッドが増加し，低速化する可能性もある．

3.2 移動不能な命令の条件

本節では，FP から VP へ移動不能な命令の条件について述べる．

明らかに移動できない命令は，FP のみが持つ機能に関わる命令であるか，もしくはパイプラインの前段である VP のプログラム実行時に確定していないデータに関わる命令である．これらは，オペコードとオペランドに基づいて判断できる．さらに，命令が果たす役割に関する条件，他の命令から伝播する条件を考慮する必要がある．

C1：FP 固有の命令である FP と VP の命令セットは同一ではなく，FP だけが実行できる命令が存在する．**表 3** に，そのような FP 固有のオペコードを示す．なお，テクスチャサンプラ機能に関する命令は代替できない一方，複数命令の代替による移動も考えられるが，簡単のため本稿では扱わない（図 3 の 10〜12 行の texld）．

表 3 FP のみで使用できるオペコード
Table 3 Opcodes executable only on FPs.

Capability	Opcode
Texture reference	texkil, texld, texldb, texldp
Conditional selection	cmp, cnd
Dot product and add	dp2add

C2：FP 固有のレジスタがオペランドに存在する サンプラレジスタには FP 固有の機能があり，VP のレジスタでは代用できない．また，FP 出力レジスタは FP の出力を書き込むので VP へ移動できない，これらのレジスタをオペランドとして含む命令は移動できない（図 3 の 10〜12，16 行）．

C3：FP で非線形の演算をしている FP の入力は，VP の出力を線形補間したものである．したがって，FP プログラム中で非線形の演算をしている場合には，その命令を VP へ移動すると結果が変わる．ゆえに，FP から移動できない．

C4：C1〜C3 に該当する命令から伝播する条件 ある命令について移動不能かどうか判断する際には，その命令の各入力オペランドのレジスタにあるデータがどのように伝播してきたか命令実行順に遡って探索する必要がある．伝播の過程において，C1〜C3 に該当した場合，FP から移動できない（図 3 の 13〜16 行）．

C1〜C4 のいずれかに該当する命令は移動できない．

3.3 命令の移動方法

FP プログラムから VP プログラムへの命令移動について以下の点に着目した．

A1 定数はプログラム実行前に確定するため，FP に与えていた定数は VP にも与えることができる．

A2 VP プログラムの末尾では，実行を終了しているため，すべての作業用レジスタは解放されている．したがって，FP から VP へ移動させる命令は，その末尾へ追加すればよく，後述する例外を除けば特別な工夫は必要ない．

A3 FP の t0〜t7 には VP の oT0〜oT7 の出力値に FP の各 PE ごとに異なるオフセットが加算されている．FP プログラム中で t0〜t7 から読み取ったデータに対して定数（定数からのみ算出される結果を含む）の加減算や移動の命令は，オフセット加算と定数加算の順番は結果に影響しないので単純に移動できる．それ以外の命令についても新たな命令の追加，変更により移動できる場合があると考えられるが，本研究では対象外とした．

条件 C1〜C4 により図 3 の FP プログラムから抽

```
   0001  mov  oPos,   v0
   0002  mov  oT0.xy, v1
→  0003  mov  r0.z,   v1.y
→  0004  mov  r1.y,   c4.x
→  0005  mov  r0.y,   c4.x
→  0006  rcp  r0.w,   c3.x
→  0007  mov  r0.x,   -r0.w
→  0008  add  r1.x,   r0.w,  r0.w
→  0009  add  r0.xy,  r0,    t0
  +0010  mov  oT1.xy, r0.xy
→  0011  add  r1.x,   r1.x,  r0.x
  +0012  mov  oT2.x,  r1.x
→  0013  add  r1.y,   r0.z,  r1.y
  +0014  mov  oT2.y,  r1.y
```

図 4 命令移動後の VP プログラム例（画像フィルタ）
Fig. 4 Example of VP program after moving instructions.

```
0001  texld  r2,  t1,  s0
0002  texld  r0,  t2,  s0
0003  texld  r1,  t0,  s0
0004  mul    r2,  r2,  c0.x
0005  mad    r1,  c1.x, r1,  r2
0006  mad    r0,  c2.x, r0,  r1
0007  mov    oC0, r0
```

図 5 命令移動後の FP プログラム例（画像フィルタ）
Fig. 5 Example of FP program after moving instructions.

出した移動可能な命令（1〜9 行）は，まず，図 2 の VP プログラムの末尾に追加する．その後，以下の手続きにより，命令移動前と同じ動作をするようにオペランド（レジスタ）の変更，および新たに必要となる命令を挿入する．

図 4 および **図 5** に，移動後の VP および FP プログラムの例を示す．この例では，移動可能な命令をすべて移動している．図 4 中の → および + は，それぞれ移動させた命令および追加した命令を表す．なお，オペランドを変更した箇所には下線を付記している．

M1：定数レジスタの割当てと変更 移動した命令で使用している定数レジスタについて，VP における未使用の定数レジスタを新たに割り当て，そのレジスタに変更を行う．

M2：FP 入力レジスタの変更 A2 の例外として，移動した命令で入力オペランドに入力レジスタ R_I を使用した命令 I_I（図 3 の 1 行 mov）を VP へ移動した場合，I_I の入力オペランド（t0.y）を，R_I に対応する VP 出力レジスタ R_O（oT0.y）に変更する必要がある．しかし，VP 出力レジスタは入力オペランドにできないため，R_O を設定した命令 I_W（図 2 の 2 行 mov）の直後に I_W の出力オペランドを適当な未使用の作業用レジスタ R_T（r11.xy）へ保存するために同じオペコードで命令 I_N（mov r11.xy, v1）を新たに挿入する必要がある（理由は M3 で述べるが，この例では挿入は不要）．その手続きにより，VP に追加する命令 I_I の入力オペランドを R_T へ変更すればよい（mov r0.x, r11.y）．

M3：FP 入力レジスタの変更の例外 M2 の例外として，I_W が mov であり，その入力オペランド R_S（v1）の値が VP プログラムの最後まで変更されていなければ，I_N の挿入は省略でき，I_I の入力オペランドを R_S へ変更するだけでよい．図 2 は，これに該当するため，命令の挿入は不要である（mov r0.x, v1.y）．

M4：VP から FP へのデータ伝達命令の挿入 FP 中に残った命令 I_F について，入力オペランドの作業用レジスタ R_F は，FP プログラム内で I_F より先に実行する命令の出力オペランドとなっていない場合，VP から R_F のデータを VP の未使用出力レジスタ R_X に書くための命令 I_X を VP プログラムに挿入する．その後，R_X に対応する FP 入力レジスタ R_Y を FP プログラムの R_F に変更する．I_X の VP プログラムの挿入位置は，VP プログラム中で最後に出力オペランドが R_F である命令の直後である．挿入命令は，M2 の手続き と同様である．

図 2〜図 5 における命令はすべて 1 スロットを消費するため，提案手法は FP プログラムが必要とするスロット数を 16 から 7 スロットに削減できた．一方，VP プログラムは 2 から 14 スロットに増加した．

なお，この例では双方のプログラムが必要とするスロット数の和は増加しているが，使用するレジスタをうまく変更することで命令を削除できることもある．

移動の際には，レジスタやスロットなどの計算資源の不足が原因で，抽出した命令のすべてを移動できない場合がある．この場合，抽出した命令のうち実行順の遅いものから順に対象から除外し，計算資源の不足を起こさない範囲で移動できる．この際，単純な命令の除外ではなく，効果的に選択し，除外することもできる．その手法は，別の機会に述べる．

4. 適 用 実 験

本章では，提案手法の有用性を評価するために，そ

表 4 実験環境
Table 4 Experimental environments.

CPU	Pentium4 3.0 GHz FSB 800 MHz 64 bit FSB
Main memory	1 GB
Graphics bus	AGP 8X
GPU	ATI Radeon X800 Pro Core clock 475 MHz Memory 256 MB Memory width 256 bit Memory clock 900 MHz 6 PEs for VP 12 PEs for FP
OS	Windows 2000
Rendering API	DirectX 9.0

の適用実験の結果を示す．実験では，以下の観点から提案手法を評価した．

(1) FP プログラムにおけるスロットの削減数
(2) 実行時間の削減
(3) 削減したスロット数と実行時間についての特性
(4) 多様な問題に対する適用可能性

4.1 実験環境

表 4 に，実験に用いた計算機の主な仕様を示す．

実験に用いたアプリケーションは $N \times N$ 画素の 2 次元画像に対する $K \times K$ 近傍ガウスフィルタ[17] である．実装は，FP を主体とする典型的なものとし，RGB 各 8 ビット精度のカラー画像入出力に対して RGB 各 32 ビット精度浮動小数点で演算する仕様とした．

また，本アプリケーションは 1 次元 2 パスフィルタになっていて，その浮動小数点演算数 M は $M = 2 \cdot (K \cdot N^2) \cdot 6 = 12KN^2$ である．

4.2 提案手法による高速化の効果

提案手法の適用効果を確認するために，移動可能命令を段階的に移動させた．これにより，FP プログラムのスロット F は段階的に減少し，同時に VP プログラムのスロット V は段階的に増加する．手法適用前は $V = 2$，$F = 63$ スロットであり，最大適用時は $V = 18$，$F = 43$ スロットであった．

表 5 に，$K = 13$，$N = 256$〜1024 のもとで，提案手法を段階的に適用したときのプログラムの実行時間を示す．表中の GPU→CPU 欄は，結果画像を GPU から CPU へ転送する際に要したデータ転送時間である．また，CPU から GPU への画像データ転送を含むフィルタ実行時間を t，総実行時間を T，手法適用前（$F = 63$）の t および T をそれぞれ t_o および T_o，手法適用前後の実行時間比をそれぞれ t/t_o および T/T_o，浮動小数点演算速度 $S = M/t$ として記す．なお，時間の単位はミリ秒である．

表 5 の Filter 欄から，提案手法により GPU 処理時間を短縮できていることが分かる．しかし，Total 欄より，GPU から CPU へのデータ転送時間を含む実行時間 T はさほど短縮できていない．これは，GPU でのフィルタ処理の実行時間に比べて結果画像を CPU へ転送するためのデータ転送時間が長いためである．

GPU から CPU への転送速度が遅い理由は，使用した GPU の特性[10] にあると考えられる．この問題は，GPU から CPU への転送速度が AGP8X に比べて高速（最大 4 GB/秒）である，PCI Express 16X 規格用の GPU では軽減されると考えられる．

また，手法適用による，FP プログラムのスロットの削減が GPU プログラムの実行速度に与える影響も，表 5 の Filter 欄には示されており，提案手法適用により，F の減少とともに処理が高速になっていることが分かる．したがって，FP の命令数の減少による影響よりも，VP の命令数と VP から FP へ出力するデータ数の増加による影響は小さいと考えられる．

削減した FP プログラムのスロットと速度向上効果についての特性は表 5 の Filter 欄（t/t_o）に示されており，F と t/t_o 比には直線的な関係がある．これは，本手法により FP プログラムから削除される命令はすべてテクスチャサンプラ機能に関する命令以外であるため，命令実行時にビデオメモリにアクセスすることがなく，1 命令あたりの実行時間がほぼ一定であったためと考えられる．また，すべての移動可能な命令を移動したとき，適用前と比べて $N = 256$ の場合に 53%，$N = 512$〜1024 の場合に 58〜59% の実行時間に短縮できている．どちらも，F の削減にともなう改善効果が表れているが，F の削減（43/63 スロット = 68%）と比べて効果が高い．この原因としては，FP プログラムの命令削除により，FP プログラム中のテクスチャ・アクセス命令の実行間隔が短縮されたことにより，各 PE からのテクスチャ・アクセス順序に変化が生じハードウェアが効率的に動作した可能性が考えられる．

提案手法により，VP へ命令を移動しすぎた場合には，VP が性能ボトルネックになる可能性が考えられるが，VP と FP では実行回数が違うため，命令移動による実行時間の変化にも違いがある．

2.3 節の実装手法では，1 つの $N \times N$ 画素の四角形の塗りつぶし処理に対して，VP は頂点数 4 回，FP は四角形の画素数 N^2 回の処理を実行することになり，VP の実行回数は，FP の実行回数に比べて少ない．

VP および FP がそれぞれの PE で並列に処理を分

表 5 $K \times K$ 近傍ガウスフィルタの実行時間（$K = 13$）
Table 5 Execution time for $K \times K$ neighborhood Gaussian filter, where $K = 13$.

V（slot）	F（slot）	N	Filter		GPU→CPU	Total		S（GFLOPS）
			t（ms）	t/t_o（%）	（ms）	T（ms）	T/T_o（%）	
2	63	256	1.1	100	2.0	3.4	100	9.7
		512	4.7	100	5.3	10.4	100	8.7
		768	10.5	100	11.9	23.0	100	8.7
		1024	18.7	100	21.0	40.6	100	8.8
6	50	256	0.7	71	2.0	3.0	90	13.7
		512	3.5	74	5.3	9.1	88	11.7
		768	7.8	74	11.9	20.2	88	11.8
		1024	13.8	74	21.0	35.7	88	11.9
7	49	256	0.7	69	2.0	3.0	89	14.1
		512	3.4	72	5.3	9.0	87	12.0
		768	7.6	72	11.9	20.1	87	12.1
		1024	16.5	72	21.0	35.4	87	12.1
10	48	256	0.7	66	2.0	3.0	89	14.6
		512	3.3	70	5.3	8.9	86	12.4
		768	7.4	70	11.9	19.9	86	12.5
		1024	13.2	70	21.0	35.1	86	12.4
11	47	256	0.7	64	2.0	2.9	88	15.1
		512	3.2	68	5.3	8.8	85	12.7
		768	7.1	68	11.9	19.6	85	12.9
		1024	12.7	68	21.0	34.7	86	12.9
14	46	256	0.7	62	2.0	2.9	87	15.7
		512	3.1	66	5.3	8.8	84	13.1
		768	7.0	66	11.9	19.5	85	13.2
		1024	12.4	66	21.0	34.3	84	13.2
16	45	256	0.6	60	2.0	2.8	85	16.1
		512	3.0	64	5.3	8.7	83	13.5
		768	6.7	64	11.9	19.2	84	13.7
		1024	11.9	64	21.0	33.9	84	13.7
18	43	256	0.6	53	2.0	2.8	84	18.4
		512	2.8	59	5.3	8.4	81	14.8
		768	6.1	58	11.9	18.6	81	15.1
		1024	10.9	58	21.0	32.9	81	15.0

担することを考慮し，本実験の条件（頂点数 4，VP の PE 数 6，画素数 $N = 256$〜1024，FP の PE 数 12）により実行回数比を求めると，FP は VP の 2^{13}〜2^{17} 倍の実行回数となる．

したがって，VP に相当な数の命令を移動したことにより VP が性能ボトルネックになる可能性は小さく，FP プログラムのスロットの削減が，全体の実行時間短縮に大きく影響する，と考えられる．

4.3 様々な問題に対する提案手法の適用評価

様々な問題に対して提案手法を適用できるか否かを確認するために，4 種類のカーネル（2 次元畳込み演算，1 次元畳込み演算，整列，総和）を FP に実装し，提案手法を適用した（**表 6**）．ここで，表中の d はカーネルが読み出すデータの数である．残りの記号については，表を参照されたい．

また，**表 7** に，流体シミュレーションの GPU プログラム[18] に適用した結果を示す．表中の横線（—）は，カーネルにおいて未実装部分のスロット数，および何らかの理由で手法を適用できない箇所を表す．

表 6 FP 実装カーネルから移動可能な命令の数
Table 6 Number of movable instructions.

Kernel	d	V_0	P_0	V	P	P/P_0（%）
2D convolution	9	2	55	40	47	85
1D convolution	13	2	63	23	44	70
Sort	5	2	40	14	40	100
Sum	4	2	16	9	9	56

V_0：手法適用前の VP プログラムのスロット数
P_0：手法適用前の FP プログラムのスロット数
V：手法適用後の VP プログラムのスロット数
P：手法適用後の FP プログラムのスロット数
P/P_0：手法適用前後の FP のスロット数比

この GPU プログラムは，13 個のカーネルで構成されていて，これらのカーネルはシミュレーションの大部分を FP に実装している．なお，本来の実装では VP プログラムは存在しないが，評価のために，FP へ最低限の情報を伝達するだけの VP プログラムを追加した（図 2 参照）．

表 7 流体シミュレータ[18] のカーネルから移動可能な命令の数
Table 7 Number of instructions movable from kernel programs in fluid simulator [18].

Kernel	V_0	P_0	V	P	P/P_0 (%)
Clear	—	2	2	2	100
Copy	—	2	2	2	100
Add	—	6	2	6	100
Splat	—	9	2	9	100
Vortex	—	16	2	16	100
Scroll	—	11	8	4	37
Scroll2	—	11	8	4	37
Advect	—	34	2	34	100
Divergence	—	17	10	9	53
Jacobi	—	21	10	13	62
Subgradient	—	19	10	11	58
Display	—	41	10	33	80
ActuallyRender	—	1	2	1	100

表 8 シェーダプログラムから移動可能な命令の数
Table 8 Number of instructions in shader programs.

Shader	V_0	P_0	V	P	P/P_0 (%)
Ambient shading	6	—	6	—	—
Diffuse shader	9	—	9	—	—
Lambertial diffuse shader	12	—	12	—	—
Phon-blinn	25	—	25	—	—
Multitexture shader	8	3	8	3	100
Shilhouette shader	20	—	20	—	—
Toon shader	28	9	28	9	100
Dot-3 bump mapping	19	3	19	3	100
Fresnel shader	20	—	20	—	—

さらに，**表 8** に，GPU 本来の使い方として，3 次元グラフィクスの陰影付け（シェーダ）プログラムに対して適用した結果を示す．

表 6 および表 7 では，多くのものに手法適用による FP プログラムのスロット削減効果がある．手法適用前後の FP のスロット数比 P/P_0 が最小のものは 37% である．移動できた命令の大半は，テクスチャデータへアクセスする際のアドレス計算に関する命令であった．一方，適用効果がない（$P/P_0 = 100$）理由は，以下のように分類できた．

- N1：FP プログラムがない，もしくは非常に短い（Clear，Copy，Add）．
- N2：コンパイラが生成するコードの並びにより，手法適用が阻害されている（Sort）．
- N3：その他（Vortex，Addvect，ActuallyRender）．

表 8 では，手法により改善できたプログラムはまったくない．これらは，すべて上記 N1 に該当する．

5. 関連研究

GPU 実装プログラムの性能改善については従来から多くの提案がある．それらは，性能ボトルネック部分の処理を簡略化するものや，GPU での処理の一部を CPU が負担することにより処理時間の短縮を達成している[1]．

GPU を使用したゲームに代表される 3 次元グラフィクスアプリケーションの高速化に関しては，いくつかの手法が知られている．

- VP が性能ボトルネックの場合
  - 描画する物体を省略する．
  - 形状モデルを簡略化しポリゴン数を減らす．
- FP が性能ボトルネックの場合
  - スクリーン解像度を低下させ，描画画素を減らす．
  - 使用テクスチャを減らす．

これらの高速化手法は，人間の眼を対象とするグラフィクスアプリケーションに対しては有効である．しかし，描画精度が低下するため，入出力仕様が変わってしまい，汎用計算では適当な方法ではない．

CPU と GPU 間の入出力仕様を変更することなく性能を向上する研究は，我々の知る限り，文献 9) のみである．この研究では，行列積のデータアクセス方法に着目し実装設計変更により性能を向上させるが，行列積演算に特化している．一方，提案手法は，任意の GPU プログラムを性能向上の対象としている．

既存の性能改善技術や研究は，コンパイル前のプログラム設計変更による改善と，コンパイル時の VP または FP のいずれか一方のプログラムの範囲のコンパイラによる最適化である．提案手法は，コンパイル後の VP および FP の両方のプログラムを対象として性能向上を図る．

提案手法は，大域的な命令レベルの最適化手法と命令スケジューリング手法の両方の効果をあわせ持つ．FP の PE による処理をループと見なせば，ループ外で実行できる，ループ不変（Loop invariant），相対定数（Relative constant）をループの外へ移動することで，ループ内の計算量を削減し高速化する，命令の移動（Code motion）[19] であり，VP から受け取る値を帰納変数と見なせば，その値に関する演算（3.2 節，条件 C3 で，線形演算に限られている）命令への手法適用は帰納変数（Induction variable）の強さ軽減（Strength reduction）[19] と同じ計算量の削減効果も持つ．そして，VP での処理は FP の各 PE での処理以前に実行される共通の処理であるため，共通部分の削除（Common subexpression elimination）[19] と同じ計算量の削減効果がある．いずれの命令レベルの最適化手法も，計算量削減により高速化が期待できるが，

本手法では，命令の移動にともなって，パイプライン前後のプロセッサで負荷が移動するため，FP の計算量削減により高速化を達成するためには，FP が VP より高負荷である場合に限られる点で異なる．また，パイプライン前後の命令スケジューリングと見なすこともできるが，通常パイプラインの後段の FP の PE 数は VP の PE 数より多いため，FP から VP への命令移動が全体の計算量の削減効果がある点で既存手法とは異なる．

さらに，本手法には，FP プログラムから命令が減ることで，FP で使用されるレジスタが減少することと，GPU の内部レジスタがベクトルレジスタであること（複数のスカラ要素を 1 命令で扱うことができる）により，命令の移動にともなう，2 次的な命令の削減ができる可能性がある．

6. まとめ

本稿では，GPU プログラムの高速化を目的として，FP における命令の一部を VP へ移動する手法について述べた．提案手法は，CPU と GPU 間の入出力仕様を変更しない点，および問題の実装後に適用するという点において従来手法[5),7),9)] と異なる．

また，アセンブリ言語プログラムを対象としており，以下の 3 つの利点がある．

- VP および FP プログラムの両方を対象にするため，一方を対象とする最適化では改善できない点に対応できる可能性がある．
- アセンブリ言語プログラムを対象にするため，既存手法によるプログラム開発終了後に適用できる．
- CPU と GPU プログラムとの入出力仕様を変更しないので CPU プログラムを変更する必要がない．

実験では，$K \times K$ 近傍ガウスフィルタに対し，提案手法の効果が高いことおよび FP プログラムのスロット削減と速度向上効果の特性を確認できた．また，提案手法の改善効果は，FP から移動する命令の数に依存しており対象プログラムによっては効果がない場合があった．さらに，命令を移動することにより，VP および FP の計算資源（レジスタとスロット）を消費するため，移動できる命令の数には制約がある．

しかし，FP が SIMD である点に着目し，FP を主体として実装したプログラムでは，テクスチャデータへのアクセスが多く含まれており，その際のアドレス計算に関する命令は移動できることが多いため，手法適用の効果が期待できるプログラムは多い．

以下は今後の課題である．

(1) 演算精度の検証．VP および FP の設計思想は互いに異なるため，各々の演算精度が異なる可能性がある．この場合，FP の命令を VP へ移動することで処理結果が変わる可能性があり，この検証が必要である．また，GPU の演算精度は，プログラミング時に指定した変数の型よりも低い可能性もある[20)]．

(2) 実装の異なる GPU に対する手法適用の検証．前述のように，GPU の内部はブラックボックス的な要素がある．手法適用前後で処理結果が同じであるか否かの検証は，実装の異なる GPU に対しても必要である．

(3) 多様な GPU プログラムに対する提案手法の検証．

(4) 本提案手法に基づく GPU プログラム高速化処理系の開発．

(5) 新しい GPU の仕様（VP からテクスチャデータへアクセスできる，あるいは FP プログラム内で分岐命令が使えるなど）に対しても有効な手法への発展．また，FP よりも VP が高負荷である場合に対する対応．

謝辞 本研究のは一部，科学研究費補助金基盤研究（B）(2)（16300006）および特定領域研究（16035209）の補助による．多くの有益なコメントをいただいた査読者の方々に深く感謝いたします．

参考文献

1) Fernando, R., Harris, M., Wloka, M. and Zeller, C.: Programming Graphics Hardware, *EUROGRAPHICS 2004 Tutorial Note* (2004). http://download.nvidia.com/developer/presentations/2004/Eurographics/EG_04_TutorialNotes.pdf

2) Lengyel, J., Reichert, M., Donald, B.R. and Greenberg, D.P.: Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware, *Proc. SIGGRAPH'90*, pp.327–335 (1990).

3) Hoff, K.E., Culver, T., Keyser, J., Lin, M. and Manocha, D.: Fast Computation of Generalized Voronoi Diagrams Using Computer Graphics Hardware, *Proc. SIGGRAPH'99*, pp.277–286 (1999).

4) Takizawa, H. and Kobayashi, H.: Multi-grain Parallel Processing of Data-Clustering on Programmable Graphics Hardware, *Proc. 2nd Int'l Symp. Parallel and Distributed Processing and Applications* (*ISPA'04*), pp.16–27 (2004).

5) Harris, M.J., Coombe, G., Scheuermann, T. and Lastra, A.: Physically-Based Visual Sim-

ulation on Graphics Hardware, *Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware* (*GH'02*), pp.109–118 (2002).
6) Li, W., Wei, X. and Kaufman, A.: Implementing lattice Boltzmann computation on graphics hardware, *The Visual Computer*, Vol.19, No.7/8, pp.444–456 (2003).
7) Larsen, E.S. and McAllister, D.: Fast Matrix Multiplies using Graphics Hardware, *Proc. High Performance Networking and Computing Conf.* (*SC'01*) (2001).
8) Bolz, J., Farmer, I., Grinspun, E. and Schröder, P.: Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid, *ACM Trans. Graphics*, Vol.22, No.3, pp.917–924 (2003).
9) Fatahalian, K., Sugerman, J. and Hanrahan, P.: Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication, *Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware* (*GH'04*), pp.133–137 (2004).
10) 森眞一郎，篠本雄基，五島正裕，中島康彦，富田眞治：汎用グラフィクスカード上での簡易シミュレーションと可視化，電子情報通信学会技術研究報告，CPSY2004-24, pp.25–30 (2004).
11) DirectX 9.0 Programmer's Reference (2004). http://www.microsoft.com/japan/msdn/directx/downloads.asp
12) DirectX 9.0 日本語ドキュメント (2004). http://www.microsoft.com/downloads/search.aspx?displaylang=ja&categoryid=2
13) Akenine-Möller, T. and Haines, E. (Eds.): *Real-Time Rendering*, AK Peters, Ltd., 2nd edition, chapter 11.4.5 (2002).
14) Hennessy, J.L. and Patterson, D.A. (Eds.): *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers (1996).
15) Foley, J.D., van Dam, A., Feiner, S.K. and Hughes, J.F. (Eds.): *Computer Graphics: Principles and Practice in C*, 2nd edition, Addison-Wesley (1995).
16) 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).
17) 田村秀行（編）：コンピュータ画像処理，オーム社 (2002).
18) Palmar, V.: Navier-Stokes Fluid Simulator. http://www.strangebunny.com/techdemo_stokes.php
19) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques and Tools*, Addison-Wesley (1986).
20) Hillesland, K.E. and Lastra, A.: GPU Floating Point Paranoia, *Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors* (*GP^2'04*), p.C-8 (2004).

(平成 17 年 1 月 24 日受付)
(平成 17 年 5 月 10 日採録)

池田　孝利（学生会員）
平成元年大阪大学基礎工学部生物工学科卒業．現在，同大学大学院情報科学研究科博士課程在学中．並列処理全般に興味を持つ．

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．現在，並列処理の基礎および応用に興味を持っている．