プログラマブルGPUにおけるLU分解の設計と実装

松井 学[†,☆] 伊野文彦[†] 萩原兼一[†]

GPU（Graphics Processing Unit）とは，描画処理の高速化を目的とした1チッププロセッサのことである．本稿では，プログラマブルGPUの振舞いを解析することを目的として，数値計算の1例としてLU分解を取り上げ，その設計と実装について述べる．この実現のために，我々は，a）繰返し処理，b）分岐処理，およびc）ベクトル演算に関していくつかの方式を実装し評価した．評価実験の結果，1）依存関係のある繰返しに対してはレンダテクスチャを用いた切替え方式がVRAM（Video Random Access Memory）内のコピーを回避でき，LU分解の実行時間を半減できたこと，2）CPUおよびGPUは，分岐処理の効率に関してトレードオフの関係にあり，行列サイズが512を超える場合はCPUによる分岐処理の効率が良いこと，3）今回の実装において浮動小数点演算性能に関する効率は30%弱であり，Fatahalianらが行列積に関して指摘しているように，LU分解に関しても，GPUの演算性能を引き出すために高いバンド幅を持つGPU内キャッシュを必要とすること，および4）GPUによる分解結果がCPUのものと一致することはなく，その主な原因は分解における除算の計算誤差が累積するためであることが分かった．

Design and Implementation of LU Decomposition on the Programmable GPU

Manabu Matsui,[†,☆] Fumihiko Ino[†] and Kenichi Hagihara[†]

The graphics processing unit (GPU) is a single-chip processor whose purpose is to accelerate rendering tasks for interactive visualization. In this paper, to analyze the behavior of the programmable GPU, we describe a design and implementation of LU decomposition as an example of numerical computation. To achieve this, we have developed and evaluated some methods with different implementation approaches in terms of a) loop processing, b) branch processing, and c) vector processing. As a result, our experimental results give four important points: 1) for dependent iterations, a render texture based method avoids copies in the video random access memory (VRAM), cutting the decomposition time in half; 2) there is a tradeoff between CPU- and GPU-based branch methods, and the CPU-based branch provides higher performance for the decomposition of matrices larger than 512×512; 3) the efficiency of floating point operations is at most 30%, and as Fatahalian et al. state for matrix multiplication, the GPU also requires a higher cache bandwidth in order to provide full performance also for LU decomposition; and 4) the GPU provides different decomposition results from those obtained using a CPU, mainly due to the floating point division error that increases the error with the progress of decomposition.

1. はじめに

GPU（Graphics Processing Unit）[1),2)]とは，描画処理の高速化を目的とした処理装置である．近年，GPUは著しい性能向上をとげており，浮動小数点演算に関してCPUを超える実効性能を達成している[3)]．さらに，プログラマブルGPUの普及，プログラム長（命令スロット数）制限の緩和，および条件分岐などの機能追加とともに，本来の描画処理だけでなく，GPUを汎用処理に応用する研究が注目されている．

このような試みは，数値計算の分野においても活発である．Thompsonら[4)]は，GPUを用いた行列積を実装し，単純なCPU実装に対しておよそ3倍の高速化を達成している．一方，Larsenら[5)]は，GPUによる行列積の性能をキャッシュに関して効率の良いCPU実装（ATLAS[6)]）と比較している．その結果，GPU実装がCPU実装よりも良い性能を得るための条件として，1）GPUにおけるコアクロックの速度向上，お

† 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，日本IBMシステムズ・エンジニアリング株式会社
Presently with IBM Japan Systems Engineering Co., Ltd.

よび 2）GPU から VRAM（Video Random Access Memory）への読み書きの速度向上をあげている．そこで，Hall ら[7] は，GPU 内のキャッシュを効率良く使う手法を提案し，理論評価によりその有効性を示している．この手法は，Fatahalian ら[3] により実機において評価され，彼らは良い性能を得るためには GPU 内におけるキャッシュバンド幅の向上が不可欠であると結論づけている．

これら行列積に関する研究だけでなく，共役勾配法[8)~10)]，ヤコビ法[10),11)]，および高速フーリエ変換[12)] に関する報告もある．

このように，GPU を用いた数値計算の試みは多く報告されている．しかし，主要なベンダが GPU の内部仕様をすべて公開していないことや，アーキテクチャの技術革新および機能追加が早急であることが原因で，どのような設計指針が GPU の性能を引き出せるのかは明確でない．

そこで，本研究では連立 1 次方程式を解くために有用であり，また世界中の高性能計算機を順位づけるための性能指標として用いられている LU 分解を対象として，GPU の特性を解析することを目標とする．この実現のために，我々は a）繰返し処理，b）分岐処理，および c）ベクトル演算に関していくつかの方式を実装し評価した．

本稿の主な新規性は，我々の知る限り GPU において LU 分解を初めて実現したこと，また a）〜c）に関する方式を組み合わせ，各々の性能を評価することによりその設計指針を示したことである．

以降では，まず 2 章で GPU について述べ，数値計算への応用手法をまとめる．次に，3 章で我々の実装を構成する各方式について述べ，4 章でその適用実験の結果を示す．5 章では，実際の応用に向けて GPU および我々の実装がさらに解決すべき課題を議論する．最後に，6 章で本稿をまとめる．

2. Graphics Processing Unit（GPU）

本章では，描画処理を高速化するための GPU のアーキテクチャ[13)] について述べ，そのプログラマブル機能を用いて数値計算に応用する既存手法をまとめる．

2.1 アーキテクチャの概要

GPU が高速化対象とする描画処理とは，3 次元空間内に位置する 3 角形のポリゴンを 2 次元平面上に投影し，ピクセル（画素）として表すことである．GPU は，この処理を迅速に行うために，3 種類の処理装置を直列に並べたパイプライン構造を持つ（**図 1**）．ここで，処理装置とは，VP（Vertex Processor）および

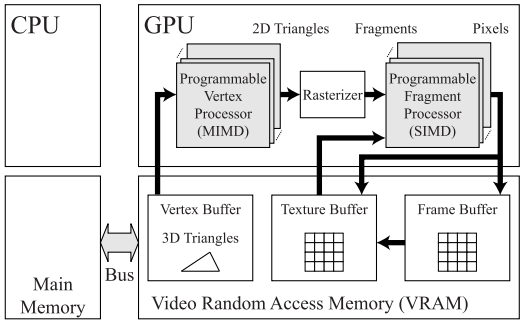


図 1 GPU のパイプラインアーキテクチャ
Fig. 1 GPU pipeline architecture.

FP（Fragment Processor）と呼ばれる 2 種類のプログラマブルな部分とそうでない部分（ラスタライザ）のことを指す．紙面の都合上，プログラマブルな部分についてのみ述べる．

VP および FP は，ベクトル長 4 のベクトルプロセッサであり，以下の特徴を持つ．

VP： VP は，ポリゴンの頂点を 2 次元平面に迅速に投影するために，頂点座標の幾何変換を高速処理できる．複数の頂点に対して異なる演算を同時に処理するために，VP は MIMD 型[14)] の構造を持つ．なお，ポリゴンは，OpenGL[15)] や DirectX[16)] などのグラフィクス API によりあらかじめ主記憶から VRAM へ転送しておく必要がある．

FP： FP は，画像の質感を向上させるために，描画した画像への模様の貼り付けを高速処理する．具体的には，2 次元画像を構成するフラグメント（画素）をラスタライザから取得し，模様を表すテクスチャとの様々な演算を処理できる．この際，FP はテクスチャを VRAM から取得し，演算結果のピクセルを VRAM へ出力する．FP の演算対象は，色および透明度を表す 4 チャネル（RGBA）のデータであるため，(C1) FP はベクトル演算によりこれらを 1 度に処理できる．さらに，複数のフラグメントに対して同一の演算を同時に処理するために，FP は SIMD 型[14)] の構造を持つ．通例，テクスチャは 2 次元画像であるため，(C2) FP は 2 次元データの各々に対して独立な演算を高速処理するものと見なせる．

従来，VP および FP には様々な強い制約があった．たとえば，フロー制御あるいは多くの命令を含むプログラムを実行できなかった．しかし，最近の機能拡張は一部の制約を緩和しつつある．たとえば，現在の VP および FP には IEEE754 標準[17)] の 32 ビット単精度

表 1 開発した実装を構成する方式と理論性能
Table 1 Theoretical performance of proposed methods.

実装	方式の組合せ			理論性能			
	ベクトル演算	分岐処理	繰返し処理	パス		VRAM 内コピー	
				回数	重み	回数	量 (B)
I1	なし	なし	コピー方式	$2N$	1	2N	$32N(N^2-1)/3$
			切替え方式			0	0
I2	なし	あり	コピー方式	N	1	N	$16N(N^2-1)/3$
			切替え方式			0	0
I3	あり	なし	コピー方式	$8N$	1/4	$2N$	$2N(4N^2+3N-4)/3$
			切替え方式			0	0
I4	あり	あり	コピー方式	$4N$	1/4	$2N$	$2N(4N^2+3N-4)/3$
			切替え方式			0	0

浮動小数点表現☆や分岐処理を扱えるものがある[2]．また，既述の API を用いて，VRAM 上のデータを主記憶へ転送（リードバック）できる．この際，データは AGP もしくは PCI Express などのバスを経由する．データ転送の間，パイプラインが停止するため，データ転送は回数および量ともに抑制する必要がある．

2.2 数値計算への応用

1 章であげた既存研究は，文献 4) を除けば，FP のみを数値計算に使用する．VP を使用しない理由としては，VP が CPU を上回る性能を持たないこと[7] があげられる．たとえば，後述する実験環境では，FP が 1 秒間に 3.6 G 個のベクトル（フラグメント）を処理できるのに対し，VP が処理できるベクトル（頂点）は 338 M 個にとどまっている．上記の理由から，以降では FP のみの使用を前提とする．

既存研究では特徴（C1）および（C2）に着目し，プログラマブル FP による数値計算を実現している．たとえば，行列積 $\mathbf{XY}=\mathbf{Z}$ の場合，要素 Z_{ij} の各々は独立に計算できるため，$\mathbf{X}$ および $\mathbf{Y}$ を各々1 枚のテクスチャとして保持し，それらを参照しながら $\mathbf{Z}$ を VRAM へ出力できる．すなわち，互いに独立に処理できる繰返しからなる 2 重ループに対しては，それらを 1 度の描画（シングルパスレンダリング）で処理できる．さらに，ベクトル演算を使用できるようにデータ構造を工夫し，VRAM の使用量を削減している．たとえば，テクスチャは 4 チャネルのデータを保持できるため，サイズ $N\times N$ の行列を $N/4\times N$ のテクスチャとして保持し，4 行 1 列を 1 度に処理する．

一方，データ依存などが原因で各要素を独立に計算できない場合，あるいはプログラム長の制約によりプログラムを分割する必要がある場合は，工夫が必要である．前者は，本研究で取り組む課題の 1 つである．後者は，既存の描画手法（マルチパスレンダリング）により解決できる．この手法では，FP が出力した画像を FP への入力（テクスチャ）として再び与える．この際，FP に割り当てる（分割後の）プログラムを順次変えていくことで，分割前のプログラム動作を実現する．以降では，最終結果を得るまでにデータがパイプラインを繰り返し流れた回数（描画回数）のことをパス数と呼ぶ．

既存研究が明らかにした知見をまとめると，GPU を効率良く使うための条件は以下のとおりである．

- GPU–CPU 間のデータ転送回数および量を抑えること
- 計算量が多く，かつ独立に処理できる 2 重ループを 1 回の描画で処理すること
- ベクトル演算を使用できるようにデータ構造を工夫し，VRAM の使用量を削減すること
- VRAM の参照回数を抑えること

3. GPU を用いた LU 分解の設計

本章では，GPU を用いた LU 分解の設計を示す．1 章であげた a）〜c）に関していくつかの方式を示し，各々の理論性能を解析する．**表 1** に，開発した各実装の特徴と理論性能をまとめる．

3.1 LU 分解

LU 分解とは，N 次元連立 1 次方程式 $\mathbf{Ax}=\mathbf{b}$ の解法である（**図 2**）．具体的には，係数行列 $\mathbf{A}$ を下三角行列 $\mathbf{L}$ および上三角行列 $\mathbf{U}$ の積に分解し，前進消去および後退代入を用いて解 $\mathbf{x}$ を得る．

LU 分解には 3 通りのアルゴリズム（外積法，内積法およびクラウト法）がある[18]．いずれも計算量は同じであるが，データ参照の局所性や並列性などに関して違いがある．本研究では，CPU と比較してキャッシュサイズが小さく[2]，高い VRAM バンド幅を持つ GPU に対し，分解時におけるデータ参照量の少ない外積法を選択する．なお，現在のところピボット選択

☆ 5.1 節で後述するように，表現に関しては IEEE 標準に準拠しているが，それを用いて計算した結果は，IEEE 標準が規程する誤差の範囲内に必ずしも収まっていない．

```
1: Algorithm LU {
2:   for (i = 0; i < N; i++) {
3:     for (j = i + 1; j < N; j++) {
4:       A_ji = A_ji/A_ii; /* update L */
5:       for (k = i + 1; k < N; k++)
6:         A_jk -= A_ik * A_ji; /* update U */
7:     }
8:   }
9: }
```

図 2 LU 分解のアルゴリズム（外積法）
Fig. 2 LU decomposition algorithm (right-looking method).

```
1: Algorithm twopassLU {
2:   for (i = 0; i < N; i++) {
3:     for (j = i + 1; j < N; j++) /* draw */
4:       A_ji = A_ji/A_ii; /* update L */
5:     for (j = i + 1; j < N; j++) /* draw */
6:       for (k = i + 1; k < N; k++)
7:         A_jk -= A_ik * A_ji; /* update U */
8:   }
9: }
```

図 3 **L** および **U** を 2 パスで更新する実装 I1（総計 $2N$ パス）
Fig. 3 I1: A two-pass implementation.

や多段同時消去は考慮していない．

3.2 設計方針

GPU を用いて LU 分解を実現するとき，解決すべき課題は以下の 3 点である．

a) データ依存のある繰返し処理：行列積では各要素を独立に計算できるのに対し，LU 分解では最外 i ループを独立に計算できない．ゆえに，LU 分解は 1 度の描画で処理できない．このような依存のある繰返しに対しては，描画を繰り返す必要があり，その実現が課題である．

b) **L** および **U** に起因する分岐処理：行列積では同一の代入文を各要素に適用できるのに対し，LU 分解では行列内の位置に応じて，各要素に適用する代入文（**L** もしくは **U** の更新）を使い分ける必要がある．FP は SIMD 型のアーキテクチャを持つため，この使い分けのための分岐処理を実現することが課題である．

c) ベクトル演算の使用：行列積と同様，ベクトル演算を使用することが GPU の性能を引き出すために必要である．さらに，データ構造を工夫することで VRAM の使用量を削減すべきである．

以降では，上記の課題を解決する各方式について述べる．なお，a）〜c）には各々2 つの方式があり，表 1 に示した実装 I1〜I4 は，これらを組み合わせたものである．I1 は，パス数の増大を認める代わりに，分岐処理を排除する．一方，I2 は分岐処理を認める代わりに，パス数を削減する．残りの I3 および I4 は，各々 I1 および I2 に対してベクトル演算を使用する．

3.3 繰返し処理に対する方式

図 2 に示したアルゴリズムは，ループ間の依存関係に関して以下の 3 点の特徴を持つ．

(1) 最外 i ループは独立に処理できない．

(2) 中間 j ループにおいて **L**（要素 A_{ji}）を更新したあとに，最内 k ループを実行する必要がある．

(3) 最内 k ループは独立に処理できる．

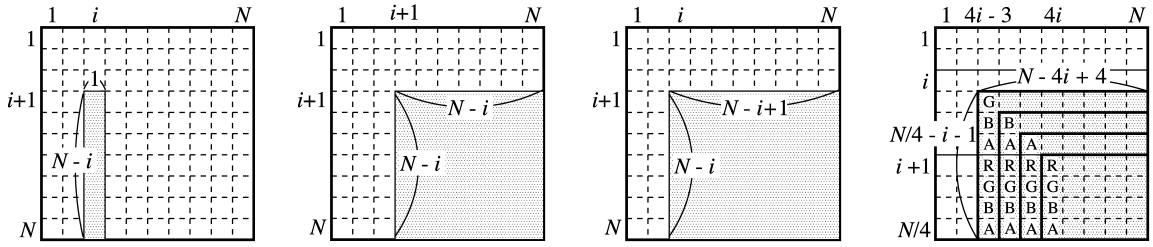
上記の特徴を基に，繰返し処理を効率良く実行するための方式について考える．

(1) より，3.2 節で指摘したように，最外 i ループは 1 度の描画で処理できない．そこで，依存のある繰返しについて，すなわち描画の繰返しについて考える．現在の GPU では，FP が VRAM へ出力するデータを FP へ直接入力することはできない．そこで，データの受渡しに関して，以下の 2 通りの方式が考えられる．

- コピー方式（ピクセルバッファ）：FP はピクセルバッファへ描画を行い，描画後にその内容をテクスチャにコピーすることで，自身への入力とする．この方式は，コピーのためのオーバヘッドをともなう．
- 切替え方式（レンダテクスチャ）：入力用および出力用のテクスチャを 1 枚ずつ用意し，描画のたびに各々の用途を切り替える方式である．この方式は，コピーを回避できる利点がある．

次に，どの部分を 1 度の描画で処理すべきかを考える．(1) より，2.1 節であげた特徴（C2）を利用するためには，残りの j および k ループを変形することにより，依存のない 2 重ループを構築する必要がある．この変形の仕方として，以下の方式が考えられる．

- ループ分割方式（実装 I1，**図 3**）：**L** および **U** の更新のためのループ（図 2 の 3〜7 行目）を 2 つ（図 3 の 3〜4 行目および 5〜7 行目）に分割する．これら 2 つのループを同時に実行することはできないが，前者（あるいは後者）は独立に処理できる．したがって，この方式は，**L** および **U** を更新するために 2 度の描画を必要とする（**図 5** (a) および (b)）．
- ループ内移動方式（実装 I2，**図 4**）：**L** の更新のための代入文（図 2 の 4 行目）を最内 k ループの内側に移動する（図 4 の 5 行目）．これにより，中間 j および最内 k からなる 2 重ループを独立に処理できる．この方式は，**L** および **U** の更新



(a) **L** の更新（実装 I1） (b) **U** の更新（実装 I1） (c) **LU** の更新（実装 I2） (d) ベクトル化（実装 I3）

図 5 第 i 番目のパスにおける描画領域（$1 \leq i \leq N$）
Fig. 5 Image region rendered at the i-th pass, where $1 \leq i \leq N$.

```
1: Algorithm onepassLU {
2:   for (i = 0; i < N; i++) {
3:     for (j = i + 1; j < N; j++) { /* draw */
4:       for (k = i; k < N; k++) {
5:         if (i == k) A_ji = A_ji/A_ii; /* L */
6:         else A_jk -= A_ik * A_ji/A_ii; /* U */
7:       }
8:     }
9:   }
10: }
```

図 4 **L** および **U** を 1 パスで更新する実装 I2（総計 N パス）
Fig. 4 I2: A one-pass implementation.

を 1 度の描画で実現できる（図 5 (c)）．しかし，代入文をループの内側に移動するため，計算量は 2 倍に増大する．

3.4 分岐処理に対する方式

3.3 節で述べたループ分割方式およびループ内移動方式に対し，分岐処理の実現を考える．

ループ分割方式は，GPU による分岐処理を必要としない．なぜなら，CPU が **L** および **U** を更新するための描画プログラムをあらかじめ区別して，描画領域とともに GPU に与えるためである．ゆえに，この方式では CPU が分岐処理を担当していると見なせる．結果，GPU は指定された領域に対する描画に専念できる．

一方，ループ内移動方式は，GPU による分岐処理を必要とする．その実現のために，図 4 に示す実装 I2 では，行列内の位置を基に代入文を使い分ける．つまり，テクスチャ内の画素ごとに，その位置座標を条件とした分岐処理を実現する．

このように，CPU における実装では分岐処理が不要であるにもかかわらず，GPU では分岐処理を必要とすることはある．LU 分解のように，テクスチャ内の位置座標を分岐条件とする場合は，パス数の増大を認めれば，分岐処理を排除できる．

3.5 ベクトル演算に対する方式

実装 I1 および I2 に対し，ベクトル演算を使用することを考える．図 2 に示したアルゴリズムは，**L** および **U** を更新するための代入文が計算時間の大半を費やす．ゆえに，これらをベクトル演算により高速化すれば効果的である．

そこで，既存研究[3),7)] のように，サイズ $N \times N$ の行列を $N/4 \times N$ のテクスチャとして保持し，ベクトル演算により 4 行 1 列を 1 度に処理する（図 5 (d)）．なお，1 行 4 列を 1 度に処理できるように行列を保持することもできる．しかし，図 5 (a) に示すように，**L** の更新は列方向を独立に処理できるが，行方向は独立に処理できない．したがって，上述のようなデータ構造を採用する．

$N/4 \times N$ のテクスチャに対しベクトル演算により LU 分解を実現するとき，3.4 節で指摘した分岐処理の問題が新たに生じる．つまり，図 5 (d) に示すように，**L** および **U** の更新のための描画領域を i の値に応じてチャネル単位で切り替える必要がある．たとえば，$i = 4$ に対してはすべての RGBA チャネルを描画すべきだが，$i = 5$ に対しては左端の画素に対してのみ，R を除く GBA チャネルを描画する必要がある．

この問題を解決するために，3.4 節で述べた方式を用いる．つまり，I1 のベクトル化は CPU による描画プログラムの切替えで実現し，I2 は GPU による分岐処理で実現する．具体的には，I1 に対しては左端の画素に関して RGBA，GBA，BA もしくは A を描画するプログラムを 4 種類用意して，これらを順に切り替えていく（実装 I3）．I2 に対しては，I3 を基に分岐処理を加えることで，適切な領域を描画する（実装 I4）．

3.6 理論性能の解析

最後に，計算量および VRAM 参照量の観点から各実装の理論性能を解析する．**表 2** に，$N \times N$ 行列に対する結果を示す．

まず，実装 I1 について解析する．i パス目において **L** および **U** を更新する領域は，それぞれ図 5 (a) および (b) である．ゆえに，i パス目において各々が更

表 2 各実装の計算量および VRAM 参照量
Table 2 Amout of computations and referred data.

実装	計算量	VRAM 参照量（B）
I1	$2N(N^2-1)/3$	$8N(2N^2-N-1)$
I2	$4N(N^2-1)/3$	$64N(N^2-1)/3$
I3 および I4	$2N(N^2-1)/3$	$N(4N^2+7N+4)$

新する要素の数は，それぞれ $(N-i)$ および $(N-i)^2$ 個である．処理全体における要素の数 S_L および S_U に対しては，i に関して 1 から N までの和を計算すればよく，各々以下のように定まる．

$$S_L = \sum_{i=1}^{N}(N-i) = N(N-1)/2 \quad (1)$$

$$S_U = \sum_{i=1}^{N}(N-i)^2 = N(2N^2-3N+1)/6 \quad (2)$$

一方，各々の更新における演算数および参照する要素数は，**L** ではともに 2 であり，**U** ではそれぞれ 2 および 3 である．ここで，アセンブリ言語において，除算は逆数に対する乗算を組み合わせて実現されていることに注意されたい．さらに，1 要素あたりのデータ量は 16 バイト（4 チャンネル ×4 バイト）であるため，I1 の計算量およびデータ参照量は，各々 $2S_L + 2S_U$ および $16(2S_L + 3S_U)$ となり，値が定まる（表 2）．

次に，I2 について解析する．i パス目おいて **LU** を更新する領域は図 5 (c) である．したがって，I1 に対する解析と同様に，i パス目において更新する要素の数は $(N-i)(N-i+1)$ であり，処理全体では以下のように定まる．

$$\sum_{i=1}^{N}(N-i)(N-i+1) = N(N^2-1)/3 \quad (3)$$

さらに，1 要素あたりの演算数および参照要素数はそれぞれ 4 であるため，表 2 のように定まる．

最後に，I3 および I4 について解析する．I3 および I4 の計算量および VRAM 参照量は等しいため，I3 の解析についてのみ述べる．I3 では，描画領域から単純に計算量を算出できない．その理由は，3.5 節で述べたように，パス i ごとに描画すべき RGBA チャネルが異なるからである．したがって，データ参照量および計算量をそれぞれ別に解析する．

まず処理領域からデータ参照量を算出する．図 5 (d) に i パス目で更新する領域を示す．ここで，図中の太線は，処理を開始する要素を格納している色，すなわち，$A_{i,j}$ を格納する RGBA ごとに処理領域が異なることを示す．この図より，i パス目に更新される **L** および **U** の要素数 L_i および U_i を算出することで，処理全体の要素数である式 (4) および式 (5) を得る．各更新で参照する要素数がそれぞれ 2 および 3 であるので，式 (4) および式 (5) を用いて実装 I1 と同様にすることで，表 2 のデータ参照量を得る．

$$\sum_{i=1}^{N/4} L_i = N(N+2)/8 \quad (4)$$

$$\sum_{i=1}^{N/4} U_i = N(4N^2+3N-4)/48 \quad (5)$$

次に計算量を算出する．i ステップの計算量を算出するとき，ベクトル演算のベクトル長が異なる領域，すなわち，図 5 (d) においてベクトル長が 4 である部分およびそれ以外の領域に分けて考える．たとえば，$A_{i,j}$ が色 G に格納されているとき，ベクトル長 3 である GBA の領域およびベクトル長 4 である RGBA の領域に分けて考える．領域に分割した後，i ステップにおいて **L** および **U** を更新する要素数を算出し，要素数に対して 1 要素の更新に必要な演算数 O_i^L および O_i^U を乗ずることで i ステップ目の計算量を算出する．なお，O_i^L および O_i^U は，ともにベクトル長 4 の場合 8 であり，以下ベクトル長が 1 減ることに 2 ずつ減る．以上より，表 2 における実装 I3 の計算量は，次の式 (6) および式 (7) の和となる．

$$\sum_{i=1}^{N/4} L_i O_i^L = N(N-1) \quad (6)$$

$$\sum_{i=1}^{N/4} U_i O_i^U = N(2N^2-3N+1)/3 \quad (7)$$

4. 評価実験

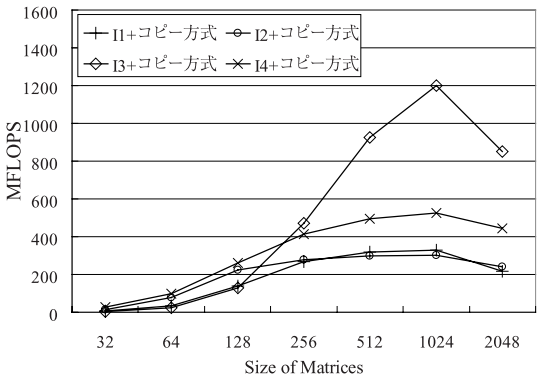
本章では，a）〜c）に関して GPU の振舞いを解析するために，以下の観点から実装 I1〜I4 の性能を評価する．

- 分岐処理の計算コストおよび VRAM 内コピー時間のトレードオフ
- ベクトル演算による高速化
- キャッシュバンド幅の使用効率

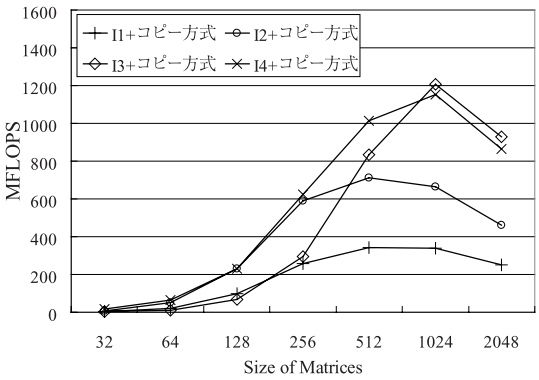
表 3 に，使用した計算機環境を示す．以降では，各々を GeForce および Quadro と略記する．各実装には，C++，OpenGL および Cg [20] を用いた．なお，現在のところ，Linux 上の動作環境ではレンダテクスチャを使用できない．したがって，この環境ではピクセルバッファ（コピー方式）のみを用いた．

4.1 分岐処理および VRAM 内コピーのトレードオフ

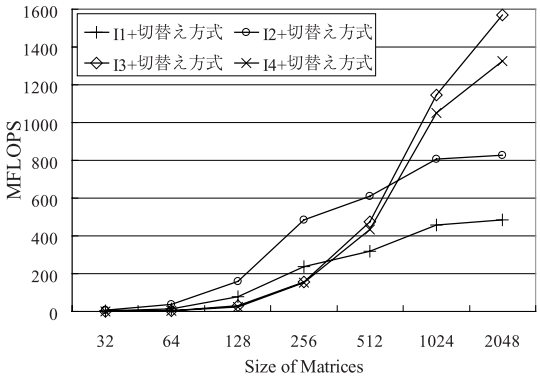
図 6 に，行列サイズ $N = 32$〜2048 としたときの



(a) GeForceFX 5900Ultra（コピー方式）



(b) QuadroFX 3400（コピー方式）



(c) QuadroFX 3400（切替え方式）

図 6 開発した実装の浮動小数点演算性能
Fig. 6 Measured performance of proposed methods.

表 3 実験環境（バンド幅（BW）の計測は GPU Bench [19] による）
Table 3 Experimental environments.

GPU	nVIDIA GeForce FX 5900Ultra	nVIDIA Quadro FX 3400
Core clock	450 MHz	350 MHz
Texture fill-rate	3.6 Gpixels/s	5.6 Gpixels/s
VRAM capacity	128 MB	256 MB
VRAM BW	27.2 GB/s	28.8 GB/s
Texture cache BW	11.4 GB/s	15.6 GB/s
Bus	AGP8X	PCI Express
CPU	Pentium 4 2.6 GHz	Pentium 4 2.8 GHz
OS	Red Hat Linux 9	Windows XP

GPU における浮動小数点演算性能（FLOPS）を示す．GeForce では実装 I3 の性能が最も良く，$N = 1024$ において約 1.2 GFLOPS である．一方，Quadro では I3 に切替え方式を用いた性能が最も良く，$N = 1024$ において約 1.6 GFLOPS である．2 つの実験環境を比較すると，図 6 (a) および (b) より，実装 I2 および I4 の性能が Quadro において相対的に向上していることが分かる．したがって，新世代の Quadro は分岐処理のオーバヘッドを軽減できている．

一方，各実験環境に共通して，分岐処理を用いない I1 および I3 が，分岐処理を用いる I2 および I4 よりもそれぞれ性能が良い．そこで，この結果に関して，**表 4** に示す Quadro の実行時間 A の内訳（GPU 計算時間 G，CPU 計算時間 C および VRAM 内コピー時間 T）から考察する．なお，GPU および CPU は非同期で動作するため，各々を同期させるための命令 glFinish() を計測対象の前後に埋め込んだ．また，実行時間 A は主記憶から VRAM への行列要素の転送および VRAM から主記憶への計算結果の読み出しを含まない．

まず，GPU 計算時間 G に着目する．I2 は I1 に対して，I4 は I3 に対してそれぞれ G が増加している．各実装の相違点は分岐処理の使用である．ゆえに，G の増加は分岐処理のコストと考えられる．一方，CPU 計算時間 C は各実装でほぼ同じであることから，プログラム切替えのためのバインド・解放のコストはほぼ無視できるといえる．ただし，行列サイズが小さい場合，I3 および I4 における C が相対的に大きいため，1 回の描画に費やす時間が短い場合は GPU が分

表 4 QuadroFX 3400 における開発した各実装の実行時間 A（ミリ秒）およびその内訳（GPU 計算時間 G，CPU 計算時間 C および VRAM 内コピー時間 T）
Table 4 Breakdown of measured time on QuadroFX 3400. A, G, C, and T represent the entire time, the GPU calculation time, the CPU calculation time, and the VRAM copy time, respectively.

N	I1 + コピー方式				I2 + コピー方式				I3 + コピー方式				I4 + コピー方式			
	A	G	C	T	A	G	C	T	A	G	C	T	A	G	C	T
32	9	6	2	1	7	6	1	1	39	12	26	1	51	1	48	1
64	13	8	3	2	10	7	2	1	55	15	38	3	64	3	56	5
128	26	14	6	6	19	12	4	3	86	20	60	6	94	6	82	6
256	79	41	11	26	55	36	6	13	160	36	108	15	160	17	126	16
512	438	250	24	164	335	240	13	82	365	102	201	62	360	84	214	63
1024	3291	2022	64	1205	2691	2050	37	604	1306	566	391	350	1334	592	391	351
2048	34942	21629	108	13206	30545	23752	95	6698	10079	5875	781	3422	10489	6307	761	3421
N	I1 + 切替え方式				I2 + 切替え方式				I3 + 切替え方式				I4 + 切替え方式			
	A	G	C	T	A	G	C	T	A	G	C	T	A	G	C	T
32	8	5	3	0	6	5	1	0	50	28	21	0	63	43	21	0
64	15	7	8	0	9	6	3	0	69	38	31	0	77	46	32	0
128	26	13	13	0	17	10	7	0	103	42	61	0	114	57	57	0
256	67	38	29	0	49	36	13	0	208	68	140	0	206	69	136	0
512	318	243	75	0	306	264	42	0	418	149	269	0	409	164	245	0
1024	1603	1470	133	0	1756	1690	66	0	1096	596	500	0	1135	650	485	0
2048	11564	11249	315	0	13309	13181	128	0	4477	3483	994	0	5048	4124	924	0

A：行列要素を保持するテクスチャが与えられてから LU 分解のための描画を終えるまでの時間.
G：描画のための一連の命令 glBegin(GL_QUADS)〜glEnd() が費やした時間.
T：VRAM 内コピーのための命令 glCopyTexSubImage2D() が費やした時間.
C：$C = A - G - T$. 主な内訳は，FP が実行するプログラムのバインド・解放やテクスチャのバインド・解放のためのオーバヘッド.

岐処理を担当した方がよい．このように，CPU および GPU は分岐処理の効率に関してトレードオフの関係にあり，たとえば，切替え方式では $N = 512$ を境にして両者の効率が逆転している．したがって，1 回の描画に費やす時間が長い場合は，プログラムの切替えを用いて CPU が分岐処理を担当する方が効率が良い．

次に，VRAM 内コピー時間 T に着目する．実装 I2 は I1 の T を半減している．また，I3 および I4 の T はほぼ同じである．この結果は表 1 に示すコピー回数に比例している．ゆえに，分岐処理によりコピー回数を削減できれば T を短縮できる．一方，切替え方式を用いた場合，コピーが発生しないため $T = 0$ である．

以上および実行時間 A より，実装 I1 に対する I2 のように，分岐処理の計算コストが VRAM 内コピー時間 T の削減量よりも小さければ，全体性能を向上させることができる．また，切替え方式を用いた I3 に対する I4 のように，T を削減できない場合は A の増大を招くため，分岐処理の使用は避けるべきである．

実験結果より，分岐処理および VRAM 内コピーに関して，次の 3 つの設計指針が考えられる．1）CPU および GPU は，分岐処理の効率に関してトレードオフの関係にあり，行列サイズが大きい場合は CPU が分岐処理を担当するほうが効率が良い．ただし，GPU 内で分岐処理を使用することでコピー量を削減できる場合は，2）分岐処理の計算コストおよびコピー削減量のトレードオフを利用して分岐処理を担当するプロセッサを決定すればよい．また，切替え方式を使用できる環境では，3）切替え方式を用いた実装が VRAM 内のコピーを回避できて効率が良い．

4.2 ベクトル演算による高速化

ベクトル演算を使用する実装 I3 は I1 よりも実行時間 A が短い（表 4）．ベクトル演算使用の効果について GPU 計算時間 G および VRAM 内コピー時間 T の短縮の観点から考察する．

まず，GPU 計算時間 G に着目する．ほぼすべての行列サイズ N において，G を短縮できている．また，$N = 2048$ において実装 I3 の G は I1 の約 1/4 である．この値はベクトル演算のベクトル長 4 の逆数にほぼ等しいことから，ベクトル演算の使用により G を短縮できたと考えられる．

次に，VRAM 内コピー時間 T に着目する．ほぼすべての行列サイズ N において T を短縮できている．T を短縮できた理由は，コピー量を削減できたためと考えられる．すなわち，ベクトル演算を使用するために，サイズ $N \times N$ 行列を $N/4 \times N$ のテクスチャと

して保持することによる．実際に，$N = 2048$ において I3 は I1 の T を約 1/4 に短縮できている．

以上をまとめると，ベクトル演算の使用は GPU 計算速度をベクトル長だけ向上できる．さらに，データ量を削減できるためコピー時間を短縮できる．ゆえに，ベクトル演算の使用は，全体性能向上に不可欠である．

4.3 キャッシュバンド幅の使用効率

我々の実装が，GPU が持つ演算性能よりも低い原因について，キャッシュバンド幅の観点から考察する．

各実装において最も演算性能が良い場合のスループットを，データ参照量（表 2）および GPU 計算時間 G より算出する．ここで，算出の際には Fatahalian ら[3] と同様に演算部分を除去しデータ参照を再現するだけのプログラムを用いた．したがって，データの自動プリフェッチを含め，レジスタに存在するデータの再利用もないものとした簡単なモデルのもとでの算出である．

最も性能が良い組合せは，GeForce では I3 および行列サイズ $N = 1024$ であり，Quadro では切替え方式を用いた I3 および $N = 2048$ である．この際のスループットは，GeForce および Quadro においてそれぞれ 8.6 GB/s および 11.4 GB/s である．ゆえに表 3 より，キャッシュバンド幅の使用効率は各々75%および 73%となる．これらの値は，Fatahalian ら[3] による行列積の使用効率とほぼ同じである．したがって，LU 分解においても，より良い演算性能を得るためには，さらに高いバンド幅を持つ GPU 内キャッシュが必要である．

5. 議　　論

本章では，実際の応用に向けて，GPU および我々の実装が解決すべき課題について議論する．

5.1 GPU に起因する計算誤差

現在の GPU は単精度浮動小数点を扱えるが，倍精度浮動小数点は扱えない．この制限は，高い計算精度を要求する数値計算分野において問題である．このように，ビット長に起因する計算精度の問題は GPU では本質的に解決されていない．

また，最近の GPU が IEEE754 標準の表現を扱えることを述べたが（2.1 節），計算誤差の厳密な保証はいまだ実現できていない[21]．今回の実験においても，計算結果が CPU および GPU 間で一致することは $N > 4$ においてなかった．特に，LU 分解では分解のたびに計算誤差が積み重なっていくため，無視できない課題である．さらに，Hillesland ら[21] が指摘しているように，除算が逆数と乗算の組合せで実現されていることから，除算の計算誤差が他の演算と比較して大きいこと（**表 5**），かつ除算の計算誤差が LU 分解において累積していくことも計算誤差の増大を引き起こしている．なお，実装 I1〜I4 間の計算結果は一致していた．

表 5 Quadro における最下位ビットの計算誤差（計測は Paranoia[21] による）
Table 5 Unit in last place error for Quadro.

演算	誤差	
	IEEE754[17]	Quadro
乗算	[−0.5, 0.5]*	[−0.78125, 0.625]
除算	[−0.5, 0.5]	[−1.19902, 1.37442]
減算	[−0.5, 0.5]	[−0.75, 0.75]
加算	[−0.5, 0.5]	[−1, 0]

*: IEEE754 標準では表現可能な最近値に丸めるため，計算誤差は最下位ビットの半分（±0.5）となる．

5.2 CPU 実装との性能比較

我々の実装が達成した性能はたかだか 1.6 GFLOPS であり，表 3 の理論性能を基に算出した FP の実行効率の 30%を下回る．一方，CPU の実装[22),23)] には，ブロック化[24)] やキャッシュ使用の最適化などにより，80%を超える実行効率を達成するものもある．たとえば，成瀬ら[22)] は Xeon 2.4 GHz を用いて 4.1 GFLOPS を達成している．したがって，GPU による実応用を実現するためには，このような CPU 実装を上回る性能を達成する必要がある．

ただし，森ら[11)] が述べているように，たとえば流体シミュレーション[25)] のように計算部分だけでなく計算結果の可視化に至るまでの過程を含めた設計が許される場合，GPU が得意とする処理を含めた全体の設計は有効である．その場合，処理量が増大するため，パス数やプログラム長（命令スロット数）の制限を考慮する必要がある．

5.3 ピボット選択の実現

現在，我々の実装はピボット選択を実現できていないため，分解アルゴリズムそのものが計算誤差を抑制できていない[18)]．したがって，数値安定性を向上させ，実装の用途を広げることも課題である．

一般に，最大の絶対値を持つ要素をピボットとして選択することが良いため[26)]，ピボット選択は多くの分岐処理を含む．我々の知る限り，このようなデータ内容に依存する分岐処理を GPU において効率良く実装する手法は明らかになっていない．

6. ま と め

本稿では，プログラマブル GPU における LU 分解

の設計と実装について述べた．GPU の振舞いを解析するために，我々は a）繰返し処理，b）分岐処理，および c）ベクトル演算に関していくつかの方式を実装し評価した．

評価実験の結果，1）依存関係のある繰返しに対しては切替え方式を用いた実装が VRAM 内のコピーを回避でき，LU 分解の実行時間を半減できたこと，2）CPU および GPU は，分岐処理の効率に関してトレードオフの関係にあり，行列サイズが 512 を超える場合は CPU による分岐処理の効率が良いこと，3）今回の実装において浮動小数点演算性能に関する効率は 30%弱であり，GPU の持つ演算性能を引き出すためには高いバンド幅を持つ GPU 内キャッシュが必要であること，および 4）GPU および CPU の計算結果が一致することはなく，その主な原因は分解における除算の計算誤差が累積するためであることが分かった．

このように，行列積と同様，LU 分解は GPU が不得意とする応用分野の 1 つであることを確認できた．しかし，GPU は CPU（ムーアの法則[27]）を超える速度で性能を向上していて[12]，計算誤差を許せば，高性能計算資源として成り立つ可能性を秘めている．

今後の課題としては，ピボット選択や多段同時消去による高速化があげられる．また，計算誤差の解析にも取り組んでいきたい．

謝辞 本研究の一部は，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．また，有益な御意見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. (Ed.): *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Addison-Wesley, Reading, MA (2004).
2) Pharr, M. and Fernando, R. (Eds.): *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA (2005).
3) 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).
4) Thompson, C.J., Hahn, S. and Oskin, M.: Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis, *Proc. 35th IEEE/ACM Int'l Symp. Microarchitecture* (*MICRO'02*), pp.306–317 (2002).
5) Larsen, E.S. and McAllister, D.: Fast Matrix Multiplies using Graphics Hardware, *Proc. High Performance Networking and Computing Conf.* (*SC2001*) (2001).
6) Whaley, R.C., Petitet, A. and Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project, *Parallel Computing*, Vol.27, No.1/2, pp.3–35 (2001).
7) Hall, J.D., Carr, N.A. and Hart, J.C.: Cache and Bandwidth Aware Matrix Multiplication on the GPU, Technical Report UIUCDCS-R-2003-2328, University of Illinois (2003).
8) Krüger, J. and Westermann, R.: Linear Algebra Operators for GPU Implementation of Numerical Algorithms, *ACM Trans. Graphics*, Vol.22, No.3, pp.908–916 (2003).
9) 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).
10) Moravánszky, A.: Dense Matrix Algebra on the GPU (2003). http://www.shaderx2.com/shaderx.PDF
11) 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).
12) Moreland, K. and Angel, E.: The FFT on a GPU, *Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware* (*GH'03*), pp.112–119 (2003).
13) 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
14) Grama, A., Gupta, A., Karypis, G. and Kumar, V.: *Introduction to Parallel Computing*, 2nd edition, Addison-Wesley, Reading, MA (2003).
15) Shreiner, D., Woo, M., Neider, J. and Davis, T. (eds.): *OpenGL Programming Guide*, 4th edition, Addison-Wesley, Reading, MA (2003).
16) Microsoft Corporation: DirectX (2005). http://www.microsoft.com/directx/
17) Stevenson, D.: A Proposed Standard for Binary Floating-Point Arithmetic, *IEEE Computer*, Vol.14, No.3, pp.51–62 (1981).
18) Dongarra, J.J., Duff, I.S., Sorensen, D.C. and Vorst, H.V.D. (Eds.): *Solving Linear Systems*

on Vector and Shared Memory Computers, SIAM, Philadelphia, PA (1991).
19) Buck, I., Fatahalian, K. and Hanrahan, P.: GPUBench: Evaluating GPU Performance for Numerical and Scientific Application, *Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors* (*GP^2 '04*), p.C-20 (2004).
20) 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).
21) 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). http://www.cs.unc.edu/ibr/projects/paranoia/
22) 成瀬 彰，住元真司，久門耕一：Xeon プロセッサ向け Linpack ベンチマーク最適化手法とその評価，情報処理学会論文誌：コンピューティングシステム，Vol.45, No.SIG 11 (ACS 7), pp.62–70 (2004).
23) Goto, K. and van de Geijn, R.: On Reducing TLB Misses in Matrix Multiplication, Technical Report CS-TR-02-55, The University of Texas at Austin (2002).
24) 寒川 光：LU 分解のブロック化アルゴリズム，情報処理学会論文誌，Vol.34, No.3, pp.398–408 (1993).
25) 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).
26) Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P.: *NUMERICAL RECIPES in C: The Art of Scientific Computing*, Cambridge University Press, Cambridge, UK (1988).
27) Moore, G.E.: Cramming more components onto integrated circuits, *Electronics*, Vol.38, No.8, pp.114–117 (1965).

(平成 17 年 1 月 24 日受付)
(平成 17 年 5 月 16 日採録)

松井 学

平成 15 年大阪大学基礎工学部情報科学科卒業．平成 17 年同大学院情報科学研究科修士課程修了．現在，日本アイ・ビー・エムシステムズ・エンジニアリング株式会社に勤務．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，最優秀若手研究賞受賞．並列ソフトウェア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．