CPUとGPUを用いた並列GEMM演算の提案と実装

大島聡史[†] 吉瀬謙二[†,☆]
片桐孝洋[†] 弓場敏嗣[†]

画像処理用のハードウェアであるGPU（Graphics Processing Unit）の性能向上にともない，GPUの演算能力を様々な分野で活用する研究がさかんである．我々はGPUを用いた新しい計算方式として，CPUとGPUの並列処理による数値計算方式を提案してきた．本稿では既存の数値計算ライブラリに対して計算方式を適用し，性能チューニングのための検討およびベンチマークプログラムを用いた性能評価を行った．計算方式をBLASのGEMMに適用して並列GEMMを作成し，これを用いてHPLベンチマークを実行したところ，Pentium4 3.0 GHz単体での実行と比べて最大で1.45倍の性能を達成した．GPUによる演算が単精度であるためHPLの高速化としては問題があるものの，計算方式が現実的なアプリケーションに適用できる可能性を示すことができた．

Proposal and Implementation of Parallel GEMM Routine Using CPU and GPU

Satoshi Ohshima,[†] Kenji Kise,[†,☆] Takahiro Katagiri[†]
and Toshitsugu Yuba[†]

GPUs for numerical computations are becoming an attractive research topics. We have proposed a new computation method of GPU, which utilizes parallel processing based on CPU and GPU. In this paper, we apply this method to existing numerical computation library. We examine a performance tuning method and execute performance experiments using a benchmark program. We also apply the method to the GEMM routine of BLAS and execute the HPL benchmark. As a result, the performance can be improved to 1.45 times by our method, compared with a CPU only environment using Pentium4 3.0 GHz. There is a precision problem depending on GPU's arithmatic precision, but we show such a potentiality that our method can be applied to various applications.

1. はじめに

近年，高度な画像処理の要求にともないGPU（Graphics Processing Unit）の性能が著しく向上している[1]．現在のGPUはCPU（Central Processing Unit）と比べて並列処理やベクトル処理に適したハードウェア構成であるため，画像処理以外の様々な処理にも利用され始めている[2]．今日では，GPUを汎用演算に利用することはGPGPU（General-Purpose computation using GPUs）と呼ばれている[3]．

数値計算は，GPGPUの代表的な適用対象問題の1つである．GPUの得意とする並列処理やベクトル処理は，多くの数値計算問題の高速化に有効であるため，GPUを用いた数値計算の高速化への期待は大きい．GPUを用いることで，CPUより高い演算性能を得ている例もある．

我々はGPUが演算を行う際にCPUにかかる負荷が低いことに注目し，CPUとGPUの演算能力を有効に活用する方法について研究を進めている．以前の研究[4]において，対象の数値計算問題をデータ分割し，CPUとGPUで並列に演算を行うことで処理能力を向上させる新しい計算方式を提案した．本稿では，同提案方式を既存の数値計算ライブラリに適用して新しいライブラリを作成し，CPUとGPUによる並列処理を様々な数値計算問題において利用可能にする．適用対象ライブラリは，BLAS[5]のGEMM（行列の積和演算）関数とする．

これまでのGPGPUにおける数値計算事例では，行列積やFFTのような基本的な数値計算問題の実装が行われてきた．本稿では，提案方式をより現実的な

† 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications
☆ 現在，東京工業大学大学院情報理工学研究科
Presently with Graduate School of Information Science and Engineering, Tokyo Institute of Technology

アプリケーションに適用することで，さらなる実用化の検討を行う．適用対象は HPL（High-Performance Linpack）ベンチマーク[6] とする．1 PC での HPL ベンチマークはその実行時間のほとんどを DGEMM（倍精度浮動小数型データに対する GEMM）が占める．そのため，GEMM の高速化による影響を受けやすく，提案方式が有効に活用できる可能性のあるアプリケーションであると考えられる．ただし，GPU の内部演算精度が単精度であるのに対して，HPL ベンチマークは倍精度演算を要求する．そこで，今後 GPU が倍精度演算に対応した場合を想定し，DGEMM のうち GPU の担当する演算を単精度で実行する形で計算方式の有効性検証を行う．DGEMM への適用の際には，実行時間の内訳などを調査して性能評価を行う．これらをもとに，計算方式が現実的なアプリケーションに対して適用可能であることを検証する．

本稿の構成は，以下のとおりである．2 章では，GPU や GPGPU，および GPGPU の数値計算への適用に関する技術動向と関連研究について述べる．3 章では，我々がこれまでに提案してきた CPU と GPU を用いた並列数値計算方式について述べる．4 章では，並列数値計算方式を単純な行列積和演算へ適用し，性能評価と実行時間の解析を行う．5 章では，並列数値計算方式を BLAS の GEMM に適用し，ライブラリを構築する．あわせて，性能チューニングに関する検討を行う．6 章では，5 章で作ったライブラリを用いて HPL ベンチマークを実行し，性能評価を行う．7 章はまとめの章とする．

2. 研究の背景および関連研究

汎用プロセッサである CPU にとって，高度な画像処理は負荷の高い処理である．そのため，GPU と呼ばれる画像処理専用のプロセッサが用いられている．従来の GPU は，三次元ポリゴンを高速に描画するために性能向上をとげてきた．一方で近年の GPU は，高度な照明処理や陰影処理など複雑な画像処理のニーズに応じて，複雑化および高機能化が顕著である．特徴的な技術としては，ソフトウェア制御によって描画処理の自由度を飛躍的に向上させる，プログラマブルシェーダアーキテクチャがあげられる．頂点処理やピクセル（フラグメント）処理を効率良く行うために，ベクトル演算能力の強化も行われている．画像処理で用いる演算には高い並列性を持つものが多いため，演算器の多重化による高速化も行われている．現在の GPU は，CPU を超える大量のトランジスタによって構築されている．また，CPU と比較して速度向上やアーキテクチャの改良が速く，理論性能が 100 GFLOPS を超えるなど，発展の著しいハードウェアである．最近では，OS やデスクトップ環境さえもが高いグラフィックス性能を要求する傾向にあるため，今後も GPU の性能向上や高性能な GPU の普及が進むと予想される．

GPU の性能が向上するに従い，GPU は画像処理以外の様々な用途にも利用されるようになってきた．GPU を用いて物理シミュレーションと可視化を同時に高速化したり，GPU を一般の数値計算に応用するなど，GPU の新しい応用に関する様々な研究が行われている[7]．これらの GPU を用いた汎用演算 GPGPU は，今後さらに適用範囲が広がることが考えられる．また，GPU 本来の用途の 1 つである PC 向け 3D ゲームにおいては，GPGPU の技術がすでに多くの製品で活用されている．

数値計算は，GPGPU の適用がさかんに議論されている対象分野である．その理由として，GPU の得意とする並列処理やベクトル処理が，多くの数値計算問題を高速化するうえで有効であることがあげられる．これまでに，FFT[8]，行列積[9]，LU 分解[10] など，様々な数値計算問題が GPU を用いて解かれてきた．こうした研究の中には，GPU を用いることで CPU を超える演算性能を達成しているものも少なくない．GPU の性能向上にともない，今後も様々な問題に GPU が利用されることが期待される．一方で，GPU の内部演算精度は単精度であり，倍精度演算を行える CPU と比べると精度が低いことが知られている．さらに単精度演算についても，CPU と比較すると精度が低いという指摘がなされている[11]．そのため今後 GPU が数値計算の分野で本格的に利用されるためには，反復計算など計算アルゴリズムを工夫して演算精度をカバーする，精度が致命的な問題は CPU に解かせる，GPU の精度改善が進むことに期待するなどの対応が必要である．また，GPU 上のビデオメモリ量や扱えるテクスチャ（画像データ格納用のメモリ）サイズなどの制限によって，GPU の扱える問題サイズが制限されている．具体的には，最近のコンシューマ向けのハイエンドなグラフィックスカードは，搭載されているビデオメモリ量が 512 MB 以下のものがほとんどであり，作成できる最大テクスチャサイズも 4,096 ピクセル × 4,096 ピクセル（ただし 1 ピクセルあたり 4 つの単精度浮動小数データを格納できる）に制限されている．

我々の行っている CPU と GPU による並列処理は，異機種分散環境（不均質な計算環境）の 1 つの例であると見なすことができる．不均質な計算環境について

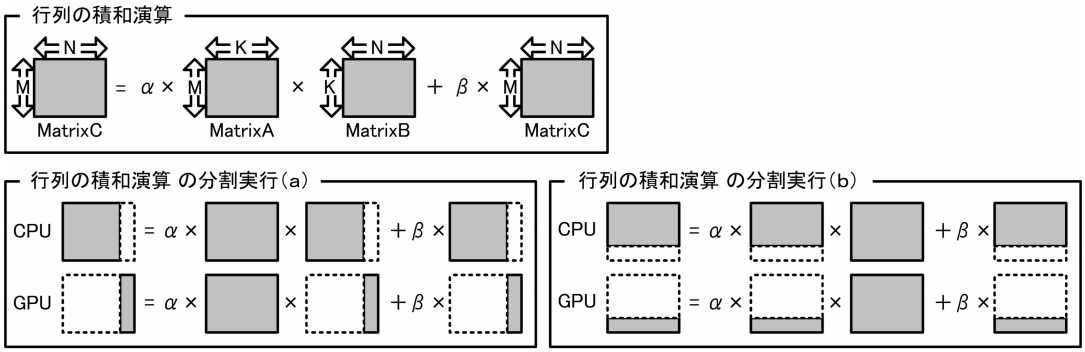


図 1 行列の積和演算の分割実行
Fig. 1 Divided execution of matrix multiplication.

は，近年性能の異なる PC を構成要素とする PC クラスタなどに関する研究が進んでいる．こうした研究においては，問題の分割方法，効率の良いスケジューリング手法，演算精度など様々なテーマが扱われている[12]．これらは数値計算ライブラリの構築において重要なテーマであり，本研究が今後さらに検討を進める必要のあるテーマでもある．

3. CPU と GPU を用いた並列数値計算方式

我々は既存の GPGPU による数値計算の研究において，GPU が演算を行っている際に CPU にかかる負荷が低いことを利用し，CPU と GPU で並列に演算を行うという計算方式を提案した[4]．本章ではこの計算方式について述べる．

CPU と GPU による問題分割の例としては，GPU が本来の性能を最も発揮できる場の 1 つである PC ゲームにおいて，GPU が描画処理を行うのと同時に CPU が衝突判定など描画以外の様々な演算を行っている．これは，CPU と GPU による対象問題の機能分割と見なすことができる．一方，我々の提案方式は数値計算を対象とすること，対象問題をデータ分割するため CPU と GPU の性能差に合わせた問題分割割合の設定が容易であること，行列積のような基本的な数値計算問題（いわゆる計算カーネル）に対しても適用できることが特徴である．また計算方式の適用実験として，**図 1** のように行列の積和演算を 2 分割し，CPU と GPU で並列に解くことで性能を向上させた．

CPU による行列積については，ATLAS[13] の SGEMM 関数（cblas_sgemm）を用いた．ATLAS は，行列やベクトルに関する様々な演算をまとめた関数インタフェース BLAS[5] の 1 つの実装である．CPU の性能を効率良く引き出すことができるため，多くの数値計算ライブラリやアプリケーションで用いられている．BLAS および ATLAS には多くの関数が用意されている．SGEMM は単精度浮動小数に対する行列積和演算であり，ほかにも倍精度浮動小数を対象とする DGEMM などがある．先の研究および本稿で用いている ATLAS のバージョンは 3.6.0 である．

GPU による行列積については，Fatahalian ら[9] による実装方法を用いた．彼らは GPU のハードウェア実装やシェーダ言語仕様がベクトル計算に適していることに注目し，GPU 上のメモリに対する行列データの配置を調整し，効率良く計算できるようにして性能を向上させた．我々も効率良くベクトル計算が行えるように，連続するデータを 1 つのピクセルに割り当てた．また CPU-GPU 間のデータ転送については，演算に必要なデータを行列ごとに一括転送し，性能の向上を図った．

以上の結果，CPU のみや GPU のみの場合と比較して最大で 1.5 倍程度の性能を得ることに成功した．さらに，CPU と GPU の性能比を事前に調査しておくことにより，並列実行時の性能向上率が最も高くなる問題分割割合の予測が可能であることを示した．同研究の課題として，最適な問題分割割合をより正確に求めること，GPU の演算精度について詳細に評価すること，そして提案方式を数値計算ライブラリに組み込むことをあげた．

なお，以降 CPU と GPU を用いた並列数値計算方式のことを単に計算方式と呼ぶ．

4. 行列積に対する計算方式の適用

4.1 行列積に対する計算方式の適用

本章では，計算方式を行列積に適用し，有効性の確認を行う．また，現実の数値計算プログラムにおいては SGEMM よりも DGEMM が重視される傾向があるため，提案方式を SGEMM だけではなく DGEMM

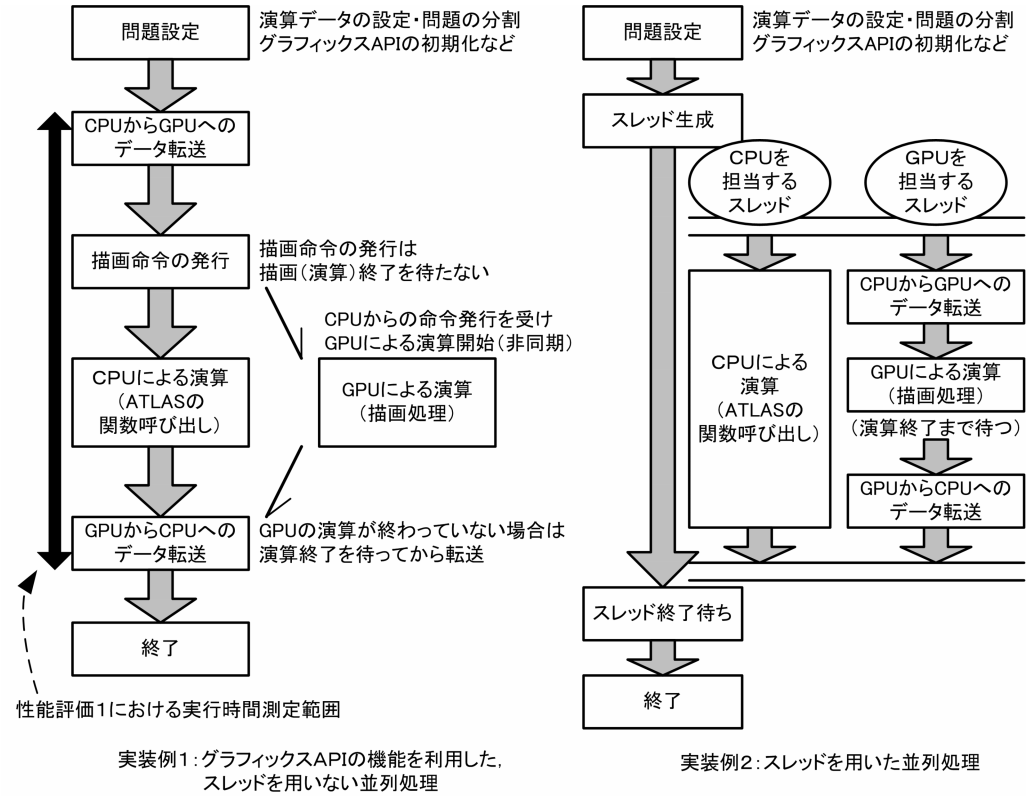


図 2 並列プログラムの構造
Fig. 2 Structure of a parallel program.

に適用することも検討する．DGEMM への適用においては，GPU の演算精度がハードウェアの制限により単精度浮動小数までであるため，CPU 上で演算データを倍精度から単精度に変換して GPU へ転送し，GPU 上では SGEMM への適用の際と同様に計算を行い，CPU へ書き戻されたデータを単精度から倍精度に変換する．しかし，この方法では演算精度の低下が発生するため，DGEMM の評価としては問題が生じる．そこで，データの変換時間や転送時間などを測定し，今後 GPU の演算精度が向上し倍精度演算が可能になったときの性能について検討する．

CPU と GPU を用いて並列処理を行うためには，2 つの実装方法が考えられる．1 つは，GPU へのデータ転送と描画命令発行，CPU による演算，GPU からの演算結果の取得を逐次的に行う方法（**図 2** の実装例 1）である．もう 1 つはスレッドを用いて CPU の制御と GPU の制御を独立させる方法（図 2 の実装例 2）である．本稿では前者の実装方法を用いた．実装には OpenGL と GLSL [14] を利用した．実装方法や実装言語による性能の差異の有無については，今後の課題とする．

表 1 評価環境
Table 1 Experimental environment.

CPU	Pentium4 3.0 GHz
メインメモリ	1.0 GB
GPU	GeForce7800GTX
ビデオメモリ（VRAM）	256 MB
GPU 接続バス	PCI-Express x16
OS	Vine Linux 3.2 (kernel 2.4.31)
プログラミング環境	GCC 3.3.2, OpenGL, GLSL

実験に用いた評価環境は**表 1** のとおりである．

4.2 性能評価 1

SGEMM および DGEMM に対して計算方式を適用し，性能を測定した．演算対象の行列は一辺の長さが 512 の倍数である正方行列とし，辺の長さを問題サイズと呼ぶことにする．結果を**図 3** に示す．グラフの横軸には問題サイズを，縦軸には実行時間をとった．並列実行時の実行時間は，CPU と GPU の問題分割割合を 10%ずつ変化させて実行時間を測定し，最も高速に解けたときの時間をとっている．ただし，測定結果にはグラフィックス API の初期化時間やシェーダプログラムのコンパイル時間を含めていない．実験の結果，SGEMM では問題サイズ 1,536 から，DGEMM では

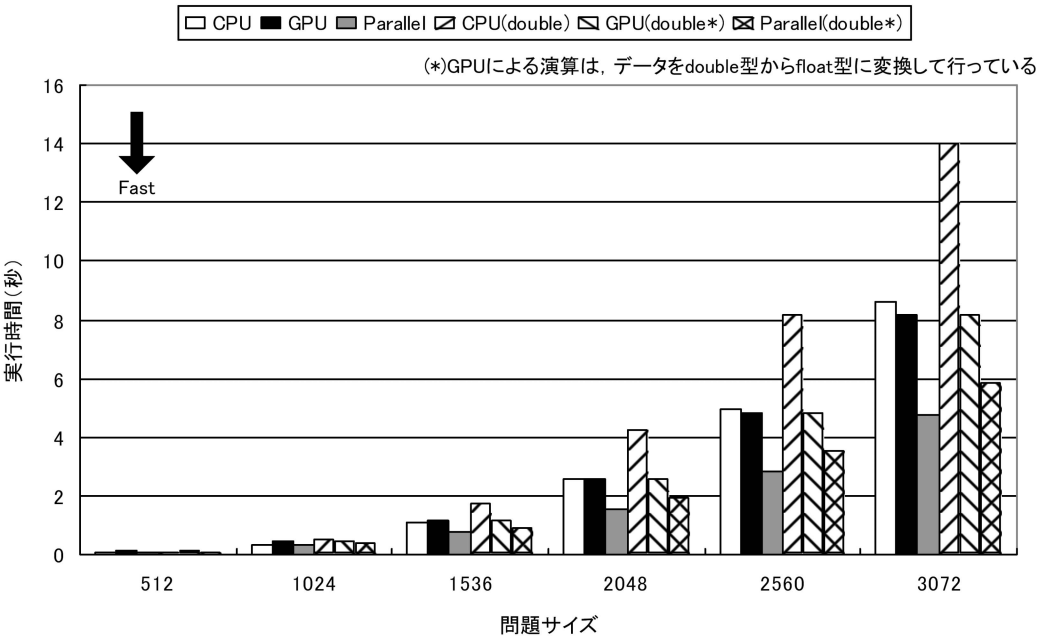


図 3 行列積に対する計算方式の適用結果
Fig. 3 Execution result of matrix multiplication.

問題サイズ 1,024 から提案方式による速度向上が見られた．また，CPU のみおよび GPU のみのうち高速なものと比較すると，SGEMM では問題サイズが 3,072 のときに最大で 1.71 倍の性能，DGEMM では問題サイズが 3,072 のときに 1.40 倍の性能が得られた．それぞれの問題サイズにおいて最も高い性能が得られた点における GPU の問題担当割合を見ると，SGEMM ではすべて 50%，DGEMM ではすべて 70%であった．先の研究と比較すると CPU と GPU の性能バランスが異なるものの，問題サイズがある程度大きい際に計算方式の効果が高いこと，最適な問題分割割合については問題サイズよりも CPU と GPU の性能差に左右されることなど，同様の傾向を得ることができた．

なお，BLAS のインタフェースには様々なデータ配置に対応するためにいくつかのパラメータがある．GPU で行列積を行う際には，CPU-GPU 間のデータ転送の前後でデータの並べ替えを行うことで対応できる．データ転送時には並べ替えを行わず，プログラマブルシェーダの対応で解決することもできる．本稿の実験では並べ替えを行っていないが，対象問題のデータ配置によっては並べ替えが必要となる可能性がある．3,072×3,072 の float 型データに対して単純に縦横を反転させる並べ替えを行ってみたところ，1 秒程度の時間を要した．対象問題によってはこれらの処理によって性能が低下する可能性があることを考慮する必要がある．

OpenGL の初期化および GLSL プログラムのコンパイルについては，問題サイズなどに関係なく 0.46 秒の時間を要した．対象問題サイズが小さい場合には大きな割合を占める可能性がある一方，対象問題サイズが大きい場合には大きな影響を及ぼさないと考えられる．また，1 回の初期化に対して何度も演算を繰り返すことができるため，影響を小さく抑えることが可能である．

4.3 SGEMM に対する性能解析

問題サイズ 3,072 における並列実行について細かく見てみることにする．まず，並列実行において最も高い性能が得られた際に CPU と GPU がそれぞれ行っていた演算を，並列処理を行わず個別に実行して実行時間を測定する．これらの時間差から，並列処理のオーバヘッドを推定する．次に，GPU については演算のためのデータを転送せずに演算を行う．これらから，CPU と GPU による個別の演算時間と CPU-GPU 間のデータ転送時間を求める．続いて，GPU の初期化時間などについても測定する．CPU-GPU 間のデータ転送時間そのものを測定していないのは，CPU から GPU へデータを転送する際に GPU 上のメモリにすべてのデータが到達し終えるタイミングや，GPU から CPU へデータを転送する際に GPU 上のメモリからデータが転送され始めるタイミングを正確に知ることが困難なためである．グラフィックス API の関

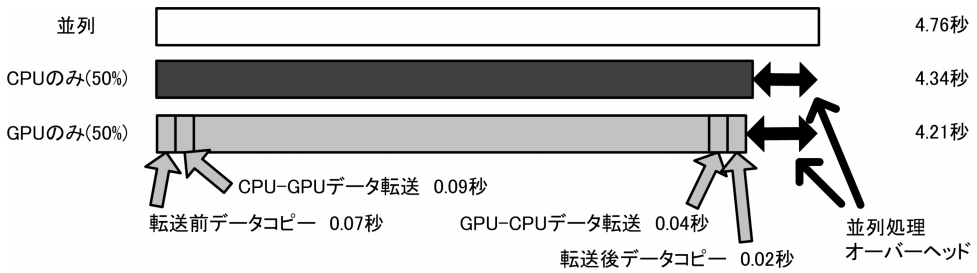


図 4 SGEMM の実行時間の内訳
Fig. 4 Breakdown of SGEMM's execution time.

数呼び出しタイミングと GPU の内部処理タイミングが一致しているとは限らないため，GPU の内部処理タイミングを詳細に解析することは困難である．

まず，問題サイズ 3,072 の SGEMM について測定を行う．並列実行時の全体の実行時間は 4.76 秒であり，このときの GPU の問題担当割合は 50%である．CPU 単独で全体の 50%を解くと，4.34 秒の時間を要した．また，GPU 単独で 50%を解くと，4.21 秒となった．並列実行時との時間差が，並列処理のオーバヘッドであると考えられる．オーバヘッドの主な要因としては，データ転送や GPU への命令発行に CPU の処理が必要であることがあげられる．さらに，GPU 単独による実行において，CPU から GPU へのデータ転送を省略すると 4.12 秒，GPU から CPU へのデータ転送を省略すると 4.17 秒，両方向のデータ転送を省略すると 4.08 秒となった．以上から，CPU から GPU へのデータ転送に 0.09 秒，GPU から CPU へのデータ転送に 0.04 秒程度かかったと推測できる．また転送されるデータ量がそれぞれ 72 MB および 18 MB であることから，転送速度は 800 MB/sec および 450 MB/sec と算出される．GPU の接続に用いている PCI-Express x16 は片方向 4 GB/sec で双方向同時転送が可能という理論性能を持つため，この速度での転送は十分に可能であると考えられる．また，CPU から GPU へのデータ転送前には演算データを GPU 上で扱いやすいように再配置し，データを書き戻した後には元に戻している．再配置の際には，GPU へデータを配置する際に行列のデータサイズが 4 の倍数である必要があるため，サイズの確認と転送用のバッファへのコピー，必要に応じてデータのパディングを行っている．これらの処理時間は 0.07 秒および 0.02 秒であった．以上から GPU による演算時間は，合計時間からデータ転送と再配置の時間を差し引いた 3.99 秒であると考えられ，実行時間の多くが演算に利用されていたと推測できる．ここで述べた実行時間の内訳を，**図 4** に示す．

最後に CPU と GPU の性能比をもとに，並列実行時の理論性能を算出し，実測性能との比較を行う．CPU のみによる問題サイズ 3,072 の SGEMM の実行時間が 8.63 秒，GPU のみでは 8.15 秒であることから，CPU の性能は GPU の 0.94 倍相当であると考えることができる．よって，この環境における並列実行時の理論性能は GPU のみでの実行の 1.94 倍であると見なすことができる．実測値についても同様に計算すると，GPU の 1.71 倍となる．以上から，この実験では理論性能の 0.88 倍という高い性能が得られていたことが判明した．

4.4 DGEMM に対する性能解析

続いて DGEMM についても測定を行う．並列実行時の全体の実行時間は 5.87 秒であり，このときの GPU の問題担当割合は 70%である．CPU 単独で全体の 30%を解くと 4.24 秒，GPU 単独で 70%を解くと 5.82 秒の時間を要した．SGEMM と同様にデータ転送を省略した実行時間を測定すると，CPU から GPU へのデータ転送を省略した場合に 5.72 秒，GPU から CPU へのデータ転送を省略した場合に 5.78 秒，両方を省略した場合に 5.67 秒という結果が得られた．CPU から GPU へのデータ転送量は 86.4 MB，GPU から CPU では 25.2 MB である．以上から，CPU から GPU へのデータ転送は 0.10 秒で 864 MB/sec，GPU から CPU へのデータ転送は 0.04 秒で 630 MB/sec という結果が得られた．GPU による演算時間は，合計時間からデータ転送と再配置の時間を差し引いた 5.54 秒であると考えられ，SGEMM と同様，実行時間の多くが演算に利用されていたと推測できる．ただし，CPU-GPU 間のデータ転送時間については，現在の GPU は double 型データを扱えないため，float 型データの転送時間を測定している．GPU による演算時間は，合計時間からデータ転送と再配置の時間を差し引いた 5.54 秒であると考えられ，SGEMM と同様，実行時間の多くが演算に利用されていたと推測できる．一方で，CPU 単独による実行時間は GPU 単独での実行と比べて短く，より多くの問題を CPU に

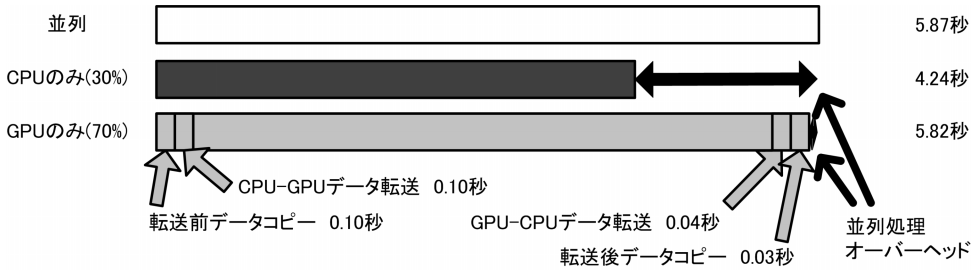


図 5 DGEMM の実行時間の内訳
Fig. 5 Breakdown of DGEMM's execution time.

割り当てた方が高い性能が得られるのではないかと考えることもできる．SGEMM と比べて GPU に関する処理が CPU に与える影響が大きかったためであると推測できるが，詳しくは判明していないため，今後の課題とする．

データ転送前後のデータの再配置，パディングおよび型変換については，転送前に 0.10 秒，転送後に 0.03 秒の時間を要した．また，double 型どうしで同様の処理を行ってみたところ，0.22 秒および 0.06 秒を要した．さらに，double 型データの転送には単純に float 型データの 2 倍の時間が必要であると仮定すると，転送時間は 0.20 秒および 0.08 秒となる．これらの時間は，並列 DGEMM 全体の実行時間と比較すると十分に小さいといえる．以上から，今後 GPU が倍精度演算に対応した場合，転送時間やデータの再配置時間は全体の実行時間に大きな影響を与えないことが推測できる．

ここで述べた実行時間の内訳を，**図 5** に示す．

最後に，SGEMM と同様に理論性能の算出および実測性能との比較を行う．問題サイズ 3,072 の DGEMM において，CPU のみでの実行時間が 14.01 秒，GPU のみでは 8.19 秒であることから，CPU の性能は GPU の 0.59 倍相当となる．よって，この環境における並列実行時の理論性能は GPU のみでの実行の 1.59 倍であると見なすことができる．一方で実測値について計算すると，GPU の 1.40 倍となる．以上から，SGEMM と同様に理論性能の 0.88 倍の性能を得られたと考えられる．

5. 並列 GEMM の提案と実装

既存の数値計算アプリケーションにおいては，性能向上や移植性確保のために数値計算ライブラリが利用されている．そのため，既存の数値計算ライブラリに対して計算方式を適用することができれば，様々なアプリケーションにおいて計算方式を容易に利用できるようになることが考えられる．そこで今回は，BLAS の SGEMM および DGEMM に対する計算方式の適用を目指し，ATLAS の SGEMM および DGEMM に対して計算方式を適用する．4 章における性能評価と同様に，DGEMM への適用においては演算データを倍精度から単精度に変換して演算を行う．演算精度の面では DGEMM への適用には問題があるものの，計算方式の適用実験を様々なアプリケーションに対して行えるようにするという意味で，DGEMM への適用も行うことにした．以降，ATLAS の SGEMM や DGEMM に計算方式を組み込んだもののことを並列 GEMM と呼ぶ．

続いて，並列 GEMM を構築するにあたり，性能と使いやすさを向上させるための検討を行う．

まず，並列 GEMM は BLAS のインタフェースに従うものとする．これにより，既存の数値計算ライブラリやアプリケーションから並列 GEMM を容易に利用できるようになることが期待できる．

一方で，BLAS とは異なる並列 GEMM ならではのチューニングも必要である．

4 章で述べたように，本計算方式は対象問題が小さい場合に高い性能を得ることができない．そのため，CPU のみで問題を解くか，並列処理を行うかのしきい値を設け，小さな GEMM は CPU のみで解くような対応を行うことで全体の性能が向上すると考えられる．特に，性能評価 1 においては DGEMM の問題サイズ，すなわち行列サイズを示す MNK 3 つの値（図 1 参照）をつねに同じ値にしていたのに対し，現実のアプリケーションではこれらの値が同じとは限らない．そのため，最適なしきい値を定めることは容易ではない．

並列実行時における CPU と GPU の問題分割割合も重要である．こちらも 4 章で述べたように，並列実行時に高い性能を得るためには CPU と GPU の性能比にあわせた問題分割を行う必要がある．性能評価 1 においては，問題サイズが一定以上であれば，最も高い性能を得られる分割割合はつねに一定の割合である

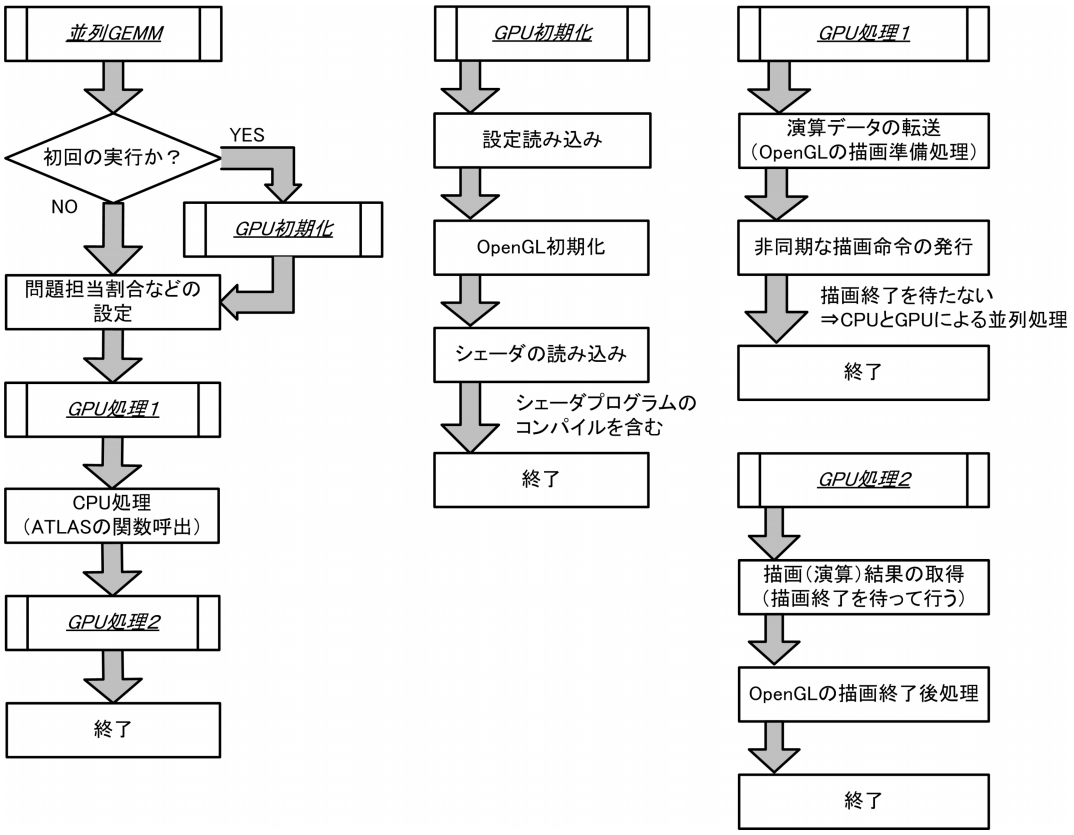


図 6 並列 GEMM の処理フロー
Fig. 6 Flow diagram of parallel GEMM.

という結果が得られた．そのため並列 GEMM についても，最も高い性能を得られる分割割合はハードウェア環境と対象問題の種類によって一意に定まる可能性が高い．

これらのパラメータは対象問題の特徴や実行するハードウェアの性能に応じて適切に定める必要があるため，外部に設定ファイルを用意することで対応できるようにする．将来的には，この設定ファイルは自動チューニングの手法を用いて自動生成できるようになることも考えられる．

以上をふまえて，次章の性能評価においては，しきい値や分割割合についていくつかの値をとって性能を比較する．

最後に，**図 6** に並列 GEMM の処理フローを示す．このフロー図に従い OpenGL と GLSL，そして ATLAS を用いてプログラムを作成した．次章では，並列 GEMM を用いてベンチマークプログラムを実行し，性能評価を行う．

6. HPL を用いた性能評価

6.1 HPL の解析

並列 GEMM の有効性検証として，既存のベンチマークプログラムに並列 GEMM を適用する．今回は HPL ベンチマーク（以下 HPL）を適用対象とする．HPL は TOP500[15] に用いられていることで有名なベンチマークプログラムであり，1 PC での HPL はその実行時間のほとんどを DGEMM が占める．そのため，並列 GEMM による GEMM の高速化の影響が十分に反映されることが期待できる．ただし，本来の HPL は倍精度で行うものであるため，GPU の担当する演算が単精度となる並列 GEMM では演算精度に問題がある．しかしながら，並列 GEMM を HPL に対して容易に適用することができ，精度に問題があっても高速に実行することができれば，並列 GEMM が現実的なアプリケーションに対して適用可能であるかの検証として意味を持つと考えられる．

HPL は 16 の性能パラメータによるチューニングを行うことができる[16]．これらのパラメータのうち，

1 PC での実行において性能を大きく左右するのは，問題サイズとブロックサイズの 2 つのパラメータである．

問題サイズについては，基本的には大きな問題サイズを用いた方が高い性能が得られる．しかしながら，使用可能な物理メモリサイズを超える大きさになると，性能が著しく低下する．今回の実験環境において，問題サイズを 1,000 ずつ変更しながら CPU のみを用いて HPL を実行したところ，問題サイズ 11,000 までは性能が向上するが，12,000 になると急激に低下するという結果が得られた．

ブロックサイズについては，CPU のキャッシュサイズに応じた値を指定することで高い性能を得ることができる．値の選択については ATLAS のインストールログが参考になる．今回の環境では，72 の倍数のときに高い性能が得られることを確認した．

以上から，今回の実験環境においては，問題サイズ 11,000，ブロックサイズ 72 において，最大性能 4.04 GFLOPS を得ることができた．

ところで，性能評価 1 において，並列 GEMM が高い性能を発揮するには GEMM の問題サイズがある程度大きい必要があることを述べた．仮に，HPL 内で実行される DGEMM の問題サイズがすべて小さい場合には，並列 GEMM が高い性能を得ることは難しい．そこで，HPL における DGEMM の実行について確認した．その結果，HPL 内では以下に示すパターンで DGEMM が繰り返し実行されることが判明した．

(1) DGEMM の問題サイズ M は，HPL の問題サイズから次第に小さな値へと変化してゆく．
(2) DGEMM の問題サイズ N と K は，次項の場合を除いてブロックサイズ未満の小さな値である．
(3) “一定の周期” で，問題サイズ N が問題サイズ M + 1，問題サイズ K がブロックサイズ，という “比較的大きな DGEMM” が実行される．
(4) ブロックサイズが大きいほど，“一定の周期” は長い周期になり，HPL 全体において “比較的大きな DGEMM” が実行される回数も減少する．

なお問題サイズの MNK は，図 1 に対応している．

以上をもとに，次節では並列 GEMM を用いた HPL ベンチマークが高い性能を得られるように，性能チューニングを行う．

6.2 並列 GEMM の性能チューニング

並列 GEMM の性能チューニングとして，HPL の問題サイズ，CPU のみで GEMM を行うか並列計算を行うかのしきい値，HPL のブロックサイズ，そして CPU と GPU の問題分割割合についての検討を行う．

HPL の問題サイズについては，CPU 単体であれば 11,000 程度まで性能が向上することが確認できている．しかしながら，並列 GEMM では CPU-GPU 間のデータ転送に一定のメインメモリが必要となることから，HPL の問題サイズを小さくする必要がある．そこで，今回は HPL の問題サイズを 8,000 とする．問題サイズを 8,000 にしたときの CPU のみによる HPL の性能は，3.90 GFLOPS に低下することを確認した．

次に CPU のみで DGEMM を行うか並列計算を行うかのしきい値について検討する．並列 GEMM は各 DGEMM の問題サイズが小さすぎると高い性能を得られない．本章における DGEMM の問題サイズは性能評価 1 と異なり MNK の 3 パラメータに左右されるため，網羅的に実験するのは効率的ではない．そこで，HPL による DGEMM 呼び出し時のパラメータを利用する．前節の HPL 解析結果で述べた，“比較的大きな DGEMM” のみを並列計算する対象とすれば，HPL 内の大きな DGEMM のみを効率良く高速化することができ，性能向上につながることが考えられる．そこで，DGEMM の問題サイズ M および N が，ともに指定されたしきい値を超えている場合にのみ並列計算を行うことにする．しきい値にはいくつかの値をとり，性能の変化を調べる．

続いてブロックサイズについて検討する．前節において，CPU 単体であれば 72 の倍数のときに高い性能が得られることを述べた．並列 GEMM は問題サイズが大きなときに高い性能が得られるため，72 の倍数の大きな値を利用することが望ましい．しかしながら値を大きくしすぎてしまうと，並列実行条件を満たす DGEMM の回数が減り，並列 GEMM による性能向上が望めなくなる可能性もある．これは，並列実行条件を満たす回数が (問題サイズ − しきい値)/ブロックサイズ + 1（除算は切り上げ）となるためである．以上から，ブロックサイズについては 72 の倍数を用い，徐々に増加させて性能の変化を調査する．

最後に問題分割割合について検討する．性能評価 1 において行列積を並列実行した際には，問題サイズが一定以上であればつねに同じ問題分割割合で高い性能が発揮された．そのため，ここでも同じような問題分割を行うことで高い性能が得られることが考えられる．ただし，本章では DGEMM が最も高い性能を発揮できるように問題サイズを調整しているのに対し，性能評価 1 ではそれを行っていない．そのため本章の性能評価においては，性能評価 1 と比較して CPU の問題担当割合を大きくした方がバランスがとれる可能性が高い．今回は GPU の問題担当割合を全体の 0%から

100%まで 5%ずつ変化させ，性能を比較する．なお，並列実行条件を満たした際にのみ並列計算を行うため，ここでいう問題担当割合は HPL 全体の問題担当割合とは一致していないことを補足しておく．GPU の問題担当割合が 100%というのは，並列実行条件を満たした際に，対象問題すべてを GPU で解くという意味である．

6.3 性能評価 2

前節の検討結果をもとに，並列 GEMM を用いて HPL を実行した．実行環境は性能評価 1 と同様である．

結果を**図 7**，**図 8** および**図 9** に示す．

図 7 は，ブロックサイズを基準にしたグラフである．横軸にブロックサイズをとり，縦軸に各ブロックサイズにおいてしきい値と分割割合を変化させたときの最大性能をとった．破線は CPU のみで HPL を実行したときの最大性能（4.04 GFLOPS）である．このグラフからは，ブロックサイズが 576 のときに最も高い性能が得られていることが分かる．ブロックサイズが小さいときに性能が上がりにくい理由は，HPL 内で実行される DGEMM の問題サイズが小さくなりすぎてしまい，並列 GEMM の性能が上がらなかったためであると考えられる．このグラフでは示されていないが，ブロックサイズが小さい方が，最大性能が得られたしきい値は大きめであるという結果も出ている．一方ブロックサイズが大きすぎても性能が上がらない．グラフにはブロックサイズ 936 までの結果を示しているが，さらに大きくすると性能は徐々に低下する．この原因は，ブロックサイズが大きすぎると並列実行条件を満たす回数が減るためであると考えられる．ブロックサイズが大きいほど，しきい値を下げても並列 GEMM の効果が得やすくなる一方，実行条件を満たす回数も減少してしまうために，性能が上がらないのである．

続いて図 8 は，GPU の問題担当割合を基準にしたグラフである．横軸に GPU の問題担当割合を，縦軸に各割合における最大性能をとった．破線は図 7 と同様，CPU のみで HPL を実行したときの最大性能である．このグラフからは，GPU の問題担当割合が 50%のときに最も高い性能が得られていることが分かる．性能評価 1 において並列 DGEMM が最大性能を発揮した際に GPU の問題担当割合が 70%であったことと比べると，最適な問題分割割合については差がある．前節で述べたように，性能評価 1 と比較して CPU の性能が相対的に上昇したため，CPU の問題担当割合がやや大きい方が高い性能が得られたと考えられる．

最後に図 9 は，CPU のみで DGEMM を行うか並列計算を行うかのしきい値を基準にしたグラフである．横軸にはしきい値を，縦軸には各しきい値における最大性能をとった．ただし，しきい値に対する性能の変化が小さいため，縦軸の表示範囲を限定した．このグラフからは，しきい値が 1,800 のときに最も高い性能が得られていることが分かる．しきい値の変化は，ブロックサイズや問題分割割合の変化と比べると並列 GEMM の性能に与える影響が小さかった．

全体としては，ブロックサイズ 576，GPU の問題担当割合 50%，しきい値 1,800 のときに最大性能 5.86 GFLOPS を達成した．このとき，並列計算の実

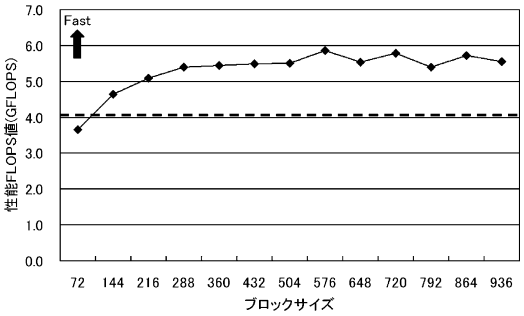


図 7 並列 GEMM による HPL の実行結果 1
Fig. 7 Execution result of HPL by parallel GEMM 1.

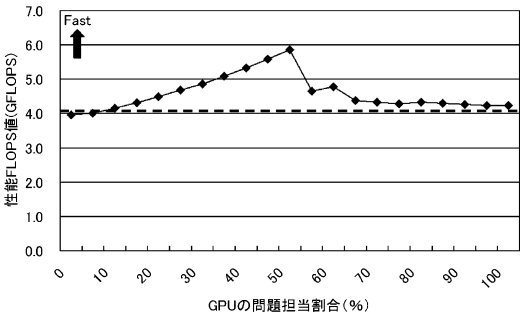


図 8 並列 GEMM による HPL の実行結果 2
Fig. 8 Execution result of HPL by parallel GEMM 2.

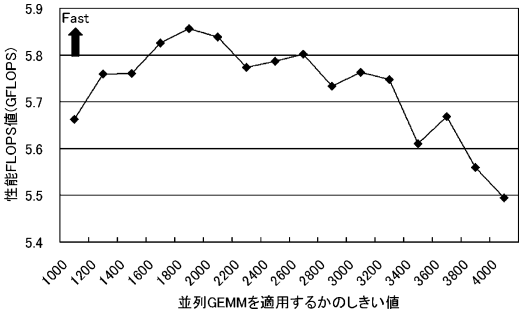


図 9 並列 GEMM による HPL の実行結果 3
Fig. 9 Execution result of HPL by parallel GEMM 3.

行条件を満たした回数は 11 回であった．CPU による HPL の最大性能が 4.04 GFLOPS 値であったことから，1.45 倍の性能が得られたことになる．性能評価 1 にて得られた最大の速度向上比が 1.40 倍であったことと比較すると，計算方式が大変有効に機能したことが分かる．その要因としては，大きなサイズの DGEMM を重点的に高速化できたことや，並列 GEMM を複数回繰り返して実行したために GPU 内部のステート設定などの時間が削減されたことが考えられる．

一方で，演算精度には問題が残った．HPL は演算の途中と終了時に演算精度の確認を行う機能を備えている．並列 DGEMM を用いて HPL を実行した場合，終了時の演算精度確認をパスできなかった．GPU による演算が単精度であるために，計算誤差が大きくなったことが原因であると考えられる．これを解決するためには，GPU の演算精度を向上させる新しい提案が必要となる．

なお，HPL 内の DGEMM 演算をすべて GPU のみで実行することも可能である．しかしながら，問題サイズが小さいときの性能が低いため，高い性能が得られないのは自明である．実験の結果，1 GFLOPS にも満たないという結果が得られた．

7. おわりに

本稿では，我々の提案する “CPU と GPU を用いた並列計算方式” を既存の数値計算ライブラリに適用し，新たなライブラリの作成を行った．この際，提案方式の行列積に対する適用をもとに，性能チューニングについても議論を行った．さらに，作成したライブラリを用いて HPL の実行を試みた．その結果，CPU のみで HPL を実行したときに比べて最大 1.45 倍の性能を達成した．ただし，GPU の内部演算精度が単精度であるため，並列 DGEMM と HPL については演算精度に問題が生じた．一方で，これまでに GPU を用いて HPL を解いた例がないことから，GPU を用いて HPL を実行したこと自体にも価値があると考えられる．BLAS のインタフェースにあわせて関数実装を行ったため，HPL ベンチマークに対する修正を最低限に抑えることもできた．提案方式を現実的なアプリケーション，ライブラリ，そしてベンチマークに対して適用し，様々な評価を行ったことには新規性があり，将来 GPU が倍精度演算に対応した場合に有用となることが期待できる．また単精度演算であっても，反復計算を行うなどアルゴリズムの工夫によって演算精度が改善できる可能性がある．以上から，本研究成果は今後多くのアプリケーションにおいて CPU と GPU を活用していくうえで重要であると考えられる．

本研究の今後の課題は以下のとおりである．

(1) 本方式の適用範囲の調査や性能チューニング手法の改善を行うため，多くのアプリケーションや多くの実験環境において性能評価を行う．特に，単精度でも実用性の高いアプリケーションをあげることができれば，本方式が速度と精度の両面で実用的なものとなる．

(2) スレッドを用いた場合と用いない場合の性能比較や，並列処理オーバヘッドの削減など，実装の改善を行う．

(3) 本稿では 1 つの CPU と 1 つの GPU という環境での実装および評価を行ったが，提案方式自体は複数の CPU や複数の GPU を搭載する環境へも適用可能であると考えられる．そこで，複数の CPU や複数の GPU を持つ環境への適用を議論し，実装と評価を行う．ただし複数の CPU を搭載した環境については，数値計算ライブラリやアプリケーションが並列環境用に最適化されている可能性が高いため，それらのプログラムと組み合わせた際にうまく高速化が行えるように検討する必要がある．

参考文献

1) Montrym, H.M.J.: THE GEFORCE 6800, *IEEE MICRO 2005*, Vol.25, No.2 (2005).
2) 森眞一郎，五島正裕，中島康彦，富田眞治：並列ベクトルプロセッサとしてのグラフィクスプロセッサ，情報処理学会関西支部大会 (2004).
3) gpgpu.org: General-Purpose computation on GPUs (GPGPU). http://gpgpu.org/
4) 大島聡史，吉瀬謙二，片桐孝洋，弓場敏嗣：CPU と GPU の並列処理による行列積和演算方式の提案，情報処理学会研究報告，Vol.2005, No.80, pp.139–144 (2005).
5) Higham, N.J.: Exploiting Fast Matrix Multiplication Within the Level 3 BLAS, *ACM Trans. Mathematical Software*, Vol.16, No.4, pp.352–368 (1990).
6) HPL: A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/
7) 宮田一乗，高橋誠史，黒田 篤：GPU コンピューティングの動向と将来像，芸術科学会論文誌 (2005).
8) Moreland, K. and Angel, E.: The FFT on a GPU, *Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware*, pp.112–119 (2003).

9) Fatahalian, K., Sugerman, J. and Hanrahan, P.: Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication, *Graphics Hardware 2004* (2004).
10) Galoppo, N., Govindaraju, N.K., Henson, M. and Manocha, D.: LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, *Proc. ACM/IEEE SC-05 Conference* (2005).
11) Hillesland, K. and Lastra, A.: GPU floating-point paranoia, *Proc. GP2* (2004).
12) Blackford, L.S., Cleary, A., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Petitet, A., Ren, H., Stanley, K. and Whaley, R.C.: Practical Experience in the Dangers of Heterogeneous Computing, *UT-CS-96-330* (1996).
13) Whaley, R.C., Petitet, A. and Dongarra, J.J.: Automated Empirical Optimization of Software and the ATLAS Project, *Parallel Computing*, Vol.27, No.1–2, pp.3–35 (2001).
14) OpenGL.org: OpenGL ホームページ. http://www.opengl.org/
15) TOP500. http://www.top500.org/
16) 笹生 健，松岡 聡：HPL のパラメータチューニングの解析，情報処理学会研究報告，SWoPP 2002, pp.125–130 (2002).

(平成 18 年 1 月 30 日受付)
(平成 18 年 4 月 18 日採録)

大島 聡史（学生会員）

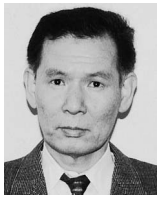
2004 年 3 月電気通信大学電気通信学部情報工学科卒業．2006 年 3 月同大学大学院情報システム学研究科博士前期課程修了．同年 4 月より同大学大学院情報システム学研究科博士後期課程在学中．GPGPU に関する研究に従事．

吉瀬 謙二（正会員）

1995 年名古屋大学工学部電子工学科卒業．2000 年東京大学大学院情報工学専攻博士課程修了．博士（工学）．同年電気通信大学大学院情報システム学研究科助手．2006 年東京工業大学大学院情報理工学研究科講師．計算機アーキテクチャ，並列処理に関する研究に従事．電子情報通信学会，IEEE-CS，ACM 各会員．

片桐 孝洋（正会員）

電気通信大学大学院情報システム学研究科助手．1994 年国立豊田工業高等専門学校情報工学科卒業．1996 年京都大学工学部情報工学科卒業．2001 年東京大学大学院理学系研究科情報科学専攻博士課程修了．博士（理学）．2001 年 4 月日本学術振興会特別研究員 PD，2001 年 12 月科学技術振興機構研究者を経て，2002 年 6 月より現職．2005 年 3 月から 2006 年 1 月まで，米国カリフォルニア大学バークレー校訪問学者．ソフトウエア自動チューニングの研究に従事．2002 年山下記念研究賞受賞．著書『ソフトウエア自動チューニング：数値計算ソフトウエアへの適用とその可能性』（慧文社）（2004）．日本ソフトウェア科学会，日本応用数理学会，ACM，IEEE-CS，SIAM 等各会員．



弓場 敏嗣（フェロー）

1966 年神戸大学大学院工学研究科修士課程修了．（株）野村総合研究所を経て，1967 年通商産業省工業技術院電子技術総合研究所（現在，独立行政法人産業技術総合研究所）に入所．以来，計算機のオペレーティングシステム，見出し探索アルゴリズム，データベースマシン，データ駆動型並列計算機等の研究開発に従事．その間，計算機方式研究室長，知能システム部長，情報アーキテクチャ部長等を歴任．1993 年より，電気通信大学大学院情報システム学研究科教授．並列処理・分散処理の科学技術一般に興味を持つ．工学博士．電子情報通信学会フェロー．日本ソフトウェア科学会，日本ロボット学会，ACM，IEEE 各会員．