情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-HPC-130 No.11
2011/7/27

三次元有限要素法アプリケーションにおける行列生成処理のCUDA向け実装

大島 聡史, 林 雅江, 片桐 孝洋, 中島 研吾 †1

本稿では三次元有限要素法 (FEM) アプリケーションにおける行列生成処理の CUDA 向け実装について述べる．GPU は高い演算性能・メモリ転送性能を持つため様々な科学技術計算アプリケーションに利用されており，FEM についても多くの研究がなされている．特に，FEM の実行時間の多くは疎行列ソルバーが占めるため，疎行列ソルバーを対象とした GPU 実装の研究が盛んである．本稿では，疎行列ソルバーに次いで実行時間を要する処理である行列生成処理を対象として，1GPU，2GPU および 2GPU と CPU を用いた実装と性能について報告する．

Implementation of Matrix Assembly in 3D Finite Element Method for CUDA

Satoshi Ohshima, Masae Hayashi, Takahiro Katagiri, Kengo Nakajima †1

We describe the implementation of matrix assembly process in 3D Finite Element Method (FEM) using CUDA. Because GPU has high calculation performance and memory transfer performance, GPU is now utilizing for several scientific applications include FEM. Especially, many researches aim at speeeding up of sparse matrix solver because sparse matrix solver has the largest time ratio of execution time of FEM. In this paper, we focus on matrix assembly process which has the second largest time ratio of FEM, and show the result of implementation and performance evaluation.

†1 東京大学 情報基盤センター
Information Technology Center, The University of Tokyo

1. はじめに

高い並列演算性能とメモリ性能を持つ GPU(Graphics Processing Unit) の活用が進んでいる．GPU を用いた汎用計算は GPGPU や GPU コンピューティングと呼ばれており，特に数値シミュレーションなど科学技術計算アプリケーションへの応用についての研究が盛んに行われている．GPU は並列度が高く連続メモリアクセスを主とする計算に適しているため，行列積などの計算においては高い性能を発揮しやすい．しかし，実際の科学技術計算においては GPU に適した計算ばかりが要求されているわけではなく，また GPU に適した計算を必要とするアプリケーションにおいてもアプリケーション全体が GPU に適した計算によって構成されているとは限らない．今後さらに GPU を適材適所で活用していくためには，様々な計算を GPU 向けに実装して性能の分析を行い，実装手法やアルゴリズムの開発を進めていく必要がある．

現在我々は三次元弾性静力学問題を対象として，GPU(CUDA[1]) を用いた有限要素法 (Finite Element Method, FEM) アプリケーションの高速化に取り組んでいる．FEM は数値シミュレーションにおいて多く用いられている解法であり，特に疎行列ソルバーが実行時間の多くを占める．疎行列ソルバーはベクトルや行列に関する演算を多く含むため GPU に適しており，GPGPU 初期の頃から多くの研究開発が行われている[2][3]．我々は疎行列ソルバーとともに，疎行列ソルバーに次いで長い実行時間を要する行列生成処理の GPU 実装にも取り組んでいる[4]．本稿では特に行列生成処理に着目し，1GPU，2GPU および 2GPU と CPU を用いて実装と性能評価を行った結果を報告する．

本稿の構成は以下の通りである．2 章では FEM と GPU について，本稿において注目している特徴を中心に述べる．3 章では係数行列生成の実装と性能評価について，特に 1GPU のみを用いた実装について述べる．4 章では複数の GPU や，GPU と CPU を併用した実装について述べる．5 章はまとめの章とする．

2. 有限要素法と GPU

2.1 有限要素法

本稿の対象アプリケーションである有限要素法は偏微分方程式の数値解法の一つであり，連続体力学が対象とする熱伝導解析，構造解析，流体解析，及びそれらの連成問題など，幅広い分野において利用されている．有限要素法を用いた計算の手順は以下の通りである：

(1) プリプロセス処理

© 2011 Information Processing Society of Japan

1. メッシュ生成

(2) FEM 本体
1. データ入力
2. 行列コネクティビティ生成
3. 係数行列生成
4. 境界条件処理
5. 線形方程式ソルバー (疎行列ソルバー)

(3) ポストプロセス
1. データ処理
2. 可視化処理

FEM 本体の処理手順の中で最も多くの実行時間を必要とするのは，線形方程式ソルバー (疎行列ソルバー) による連立一次方程式の求解である．係数行列が疎であることから反復解法が用いられ，また行列の定値対称性から共役勾配法 (Conjugate Gradient Method, CG 法) の適用が一般的であり，前処理と合わせて適用されることが多い．線形方程式ソルバーが有限要素法全体に占める実行時間の割合は，問題設定等にもよるが，大きい場合には 9 割を越える．

線形方程式ソルバーに次いで多くの実行時間を要するのが係数行列生成である．係数行列生成の計算概要 (プログラムの概形) は 3.1 節にて後述するが，疎行列ソルバーに用いる計算の多くが GPU によって高速化しやすい計算によって構成されているのに対して，係数行列生成はより複雑な構造である．

2.2 GPU と有限要素法

本稿では Fermi アーキテクチャの GPU である Tesla C2050 を対象として実装と性能評価を行う．Fermi アーキテクチャの GPU は従来の GPU と比べてキャッシュや倍精度浮動小数点演算性能が強化されており，ECC にも対応している GPU である．

疎行列ソルバー，特に CG 法は，疎行列とベクトルの積や，ベクトル同士の和，積，内積など GPU にとって高速化しやすい計算によって構成されている．そのうえ，CG 法は問題設定を問わず，さらには有限要素法のみならず多くの問題において利用される計算手法である．そのため，CUDA や OpenCL といった容易に GPU を利用できるプログラミング環境が登場する以前から，GPU を用いた CG 法の高速化に関する研究開発は多く行われている．CG 法においては GPU を活用しにくい前処理を用いる場合があり，これを高速化することが現状の大きな課題の 1 つである．

一方で有限要素法における係数行列生成は，やや単純ではないループ構造や依存性のある足し合わせを含んでおり，GPU 向けの実装は CG 法と比べて困難である．また対象問題による実装の違いも大きい．そのため，GPU を用いた係数行列生成の研究報告は少ない．

2.3 対象問題の設定と予備実験

本研究が対象とするプログラムは，GeoFEM プロジェクト[5)] で開発された並列有限要素法アプリケーションを元に整備した性能評価のためのベンチマークプログラム群の 1 つである三次元弾性静解析プログラム[6)] である．このプログラムは CPU 上での性能とメモリ効率を上げるために，係数行列に対して 3x3 のブロック化を行い，さらに対角ブロック・上三角ブロック・下三角ブロックに分けて保持している．行列格納形式は CRS(Compressed Row Storage) 形式である．

ここで，対象プログラムの全てを CPU によって計算した場合および疎行列ソルバーを CPU や GPU によって高速化した場合の実行時間内訳について確認する．実験環境は**表 1** の通りであり，対象問題の大きさは 1,000,000 要素 (接点数は 1,030,301 となる) とする．次章以降の実験においても同様の実験環境と対象問題の大きさを用いる．CG 法における前処理はブロック LU を用いており，これは簡易な前処理であり GPU にとっても容易に高速に計算できる手法である．また CPU と GPU 両方における全ての浮動小数点演算は倍精度で行う．倍精度演算を用いても並列化による演算順序の変更による計算誤差が発生することがあり，CPU による実装と GPU による実装は完全に同一の計算結果が得られるとは限らない．本稿ではこの計算誤差の大きさや，計算誤差がプログラムの実行結果に及ぼす影響の大きさにについての評価は行わない．また，ECC については有効化された状態 (=工場出荷時の設定) で性能測定を行う．なお GPU を用いた実装については我々が文献 4) で行った実装を改良したものを用いている．

実行結果を**図 1** に示す．実行結果から，1CPU による逐次実行においては FEM 全体に占める疎行列ソルバーの実行時間の割合が 91.97%と非常に大きいことが確認できた．また OpenMP や GPU を用いた際の疎行列ソルバーの実行時間もそれぞれ 85.59%および 63.29%と高い割合を占めていることが確認できた．このように，疎行列ソルバーの高速化を行っても全体の実行時間に占める割合は依然として大きい一方で，GPU を用いて疎行列ソルバーを高速化した場合には行列生成処理も 28.36%と無視できない割合を占めていることがわかる．

係数行列生成の計算概要 (プログラムの概形) を**図 2** に示す．図 2 が示すように，係数行列生成は大きく 2 つの多重ループにより構成されている．2 つのループのうち，前半のルー

© 2011 Information Processing Society of Japan

表 1 実験環境

CPU	Xeon W3520, 2.67GHz 4 コア (1 ソケット)
メインメモリ容量	12GB
GPU	Tesla C2050, 1.15GHz 448 CUDA コア (2 台)
GPU メモリ容量	1GPU あたり 3GB
GPU 接続	PCI Express Gen.2 x16
コンパイラ	gcc version 4.4.0
CUDA 開発/実行環境	CUDA toolkit 4.0
GPU ドライバ	270.41.19

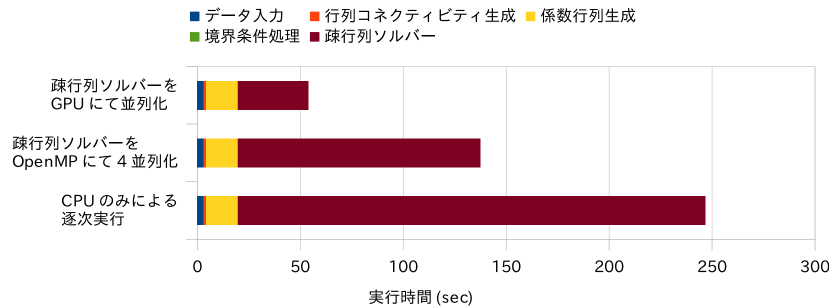

図 1 予備実験：FEM プログラムの実行時間内訳の確認

プは計算量と実行時間が非常に小さく構造も単純であるため，本稿では後半のループのみを高速化の対象とする．

係数行列生成においては要素をベースとして各接点の持つデータを生成する．各接点は周囲の複数の要素の影響を受けるため，全要素についての計算を並列に行うことはできない．そのため，並列化を行うためには同じ接点を共有しない要素を抽出して同じ色を割り当てる処理 (本稿では “塗り分け処理” と呼ぶ) を行った後で並列計算を行う必要がある．塗り分け処理は GPU に向いている計算とは考えにくいため CPU を用いて実行し，塗り分け後の並列計算 (本稿では “要素計算処理” と呼ぶ) を GPU で行うこととする．

```
do k = 1,2
 do j = 1,2
  do i = 1,2
    ガウス積分点(8点)における形状関数とその自然座標系における微分の算出
enddo  enddo  enddo

do icel = 1, 要素数
 8節点の座標から，ガウス積分点における形状関数の全体座標系における微分とヤコビアンを算出
 do ie = 1,8
  do je= 1,8
    全体接点番号ip,jpを元に係数行列における位置を算出（小規模探索処理）
   do k = 1,2
    do j = 1,2
     do i = 1,2
       要素積分，要素行列成分計算，全体行列への足し込み（完全に独立ではない）
   enddo  enddo  enddo
 enddo  enddo
enddo
```

図 2 係数行列生成の手順 (プログラムの概形)

3. 1GPU を用いた係数行列生成処理

3.1 問題設定と実装

本章では，1GPU を用いた係数行列生成処理の実装について述べる．我々は文献 4) において，一連の処理を一度の GPU カーネル呼び出しにて行う実装 (データ分割，本稿では “一括ループ分割実装” と呼ぶ) と，階層化されたループごとに GPU カーネル呼び出しにて行う実装 (データ＋タスク分割，本稿では “個別ループ分割実装” と呼ぶ) を行った．ここでは並列化のためにカラーリング (マルチカラー法) を用いた．本稿でもこの実装を元にさらなる実装と性能評価を行う．

一括ループ分割実装は GPU カーネル呼び出しが一度だけのため，一度 GlobalMemory から読み出したデータを最後に書き戻すまで高速な SharedMemory 上で扱うことができる点がメリットとなる．一方で，この実装は並列度の異なる計算部分を一度の GPU カーネル呼び出しで処理する．今回のプログラムでは GPU カーネルの先頭部分は要素数分の並列度を持つのに対して，中盤部では要素数 $\times 64$ の並列度に，末尾部では要素数 $\times 64 \times 8$ の並列度になるため，ThreadBlock や Thread の数を適切な数に調整する必要がある．今回の問題では，最内の三重ループが並列度 8 で実行できることを利用し，1ThreadBlock 内でこの三重ループを 4 セット実行することで良い性能が得られた．なお，中盤部には小規模探索があり，その結果が最内のループ内にある分岐処理に影響するため，最大で 1WARP(32Thread) が 4 つのフローに分岐する可能性がある．

個別ループ分割実装は，一括ループ分割実装と比べて各計算部分単位の最適化が行いや

© 2011 Information Processing Society of Japan

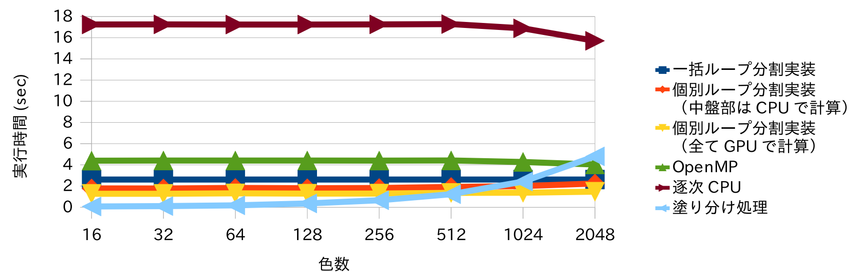

図 3 色数と性能の関係 (1GPU)

すいことや，カーネルサイズが小さいため GPU 上で同時に実行可能な WARP 数を増やしやすい点がメリットである．一方で GPU カーネルをまたいでデータを維持するには GlobalMemory を利用する必要があり，一括ループ分割実装と比べて GlobalMemory アクセス数が増加する点がデメリットとなりえる．

ところで，今回の対象プログラムには小規模な探索処理が含まれている．一般的には規模の小さな問題や探索のような分岐回数の多い (特に 1WARP 内での分岐処理が多い) 計算は GPU に適しない問題である．そこで，個別ループ分割実装については中盤部 (探索処理) を CPU で行うプログラムと GPU で行うプログラムを作成して性能を比較することにした．次節では CPU との性能比較も含めた性能評価について示す．

3.2 性能評価

本節では以下の 5 つの実装について性能の比較を行う．

(1) 一括ループ分割実装
(2) 中盤部 (小規模探索) を CPU で計算する個別ループ分割実装
(3) 全て GPU で計算する個別ループ分割実装
(4) OpenMP による並列化実装
(5) 元々の 1CPU 逐次実装

各実装について，それぞれ色数を 16 色から 2048 色まで 2 のべき乗の色数に設定して要素計算処理の実行時間を測定した．また，CPU で行った塗り分け処理の実行時間についても測定した．測定結果を**図 3** に示す．

まず塗り分け処理に注目すると，色数が多い場合には塗り分けに多くの実行時間が必要であり，色数が非常に多い場合には要素計算処理より多くの時間がかかってしまうことが確認できた．多くの色を用いた塗り分け処理を行う場合には，塗り分け処理についても高速化が必要である．

要素計算処理に注目すると，個別ループ分割実装 (全て GPU で計算) が最も良い性能が得られた．一般的に小規模探索処理は GPU に適していない処理であるが，計算量自体も少ないため並列度の高さによるメリットが分岐によるデメリットを上回ったと考えられる．

色数と要素計算処理の実行時間の関係については，色数による実行時間の逆転は起こらなかったものの，CPU による計算処理は色数が増えた際に実行時間がわずかに減少した一方，GPU による計算処理は色数が増えた際に実行時間がわずかに増加した．色数が増えた際に CPU の実行時間が減少している点については，色数が増えると一度に計算する量が減少するためにキャッシュの効果が高まったことが原因であると考えられる．逆に GPU の実行時間が増加している点については，今回用いている GPU には CPU 同様にキャッシュが搭載されてはいるものの，実行時間が短いこともあり，色ごとに GPU カーネルの起動・終了を行うコストがキャッシュの効果を上回ったことが原因であると考えられる．

4. 複数 GPU や CPU を用いた係数行列生成処理

4.1 2GPU を用いた実装と性能評価

前章の性能評価結果を踏まえて，本節では 2GPU を用いた実装について，さらに次節では 2GPU に加えて CPU を活用した実装について述べる．

今回は 2GPU 向けの実装として，要素計算処理における計算を色ごとに別々の GPU に割り当てて同時に実行することを考える．本来，色分け処理が行われていた理由は同じ接点に対する計算 (書き込み) を同時に行うことがないようにするためであるが，接点に関するデータを多重化して計算処理を行い，最後にデータの統合を行えば正しい結果を得ることができる．GPU はそれぞれが独立したデバイスメモリを有しているため，それぞれのデバイスメモリ上で計算処理を行い最後にデータの統合を行えばよい．

本実装の動作イメージを**図 4** に示す．本実装では，計算処理が終了した時点では生成された行列データは各 GPU に分散して配置されている．そのため，(生成された行列を CPU 上で処理するならば) 分散配置されている行列データをメインメモリに統合する処理が必要となる．統合処理自体は単純に配列を足し合わせるのみで良いため，FEM 全体の実行時間に与える影響は軽微となることが予想される．しかし，CPU-GPU 間で転送する必要があるデータが増加することによる性能低下が考えられる．これについては後述とする．

ⓒ 2011 Information Processing Society of Japan

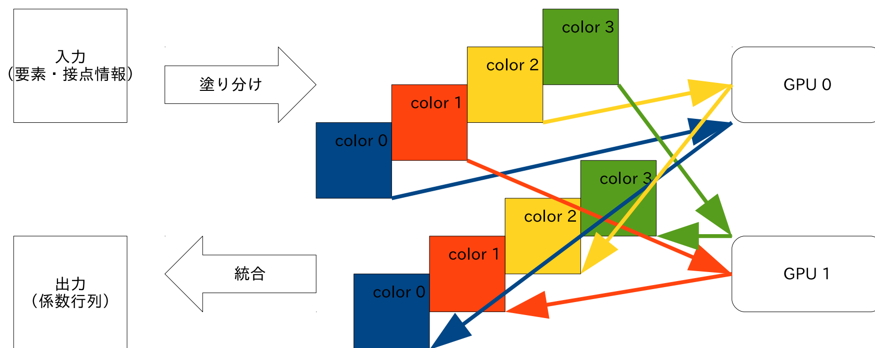

図 4　2GPU 実装の動作イメージ

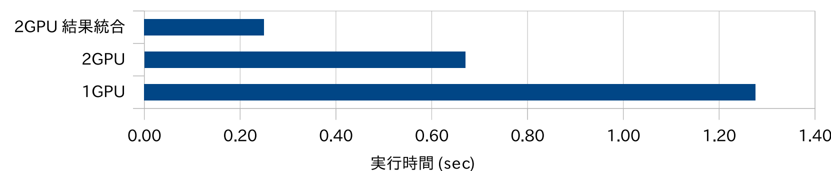

図 5　2GPU 使用時の実行時間

以上の提案に基づいて 2GPU 向けの実装を行った．実行時間測定の範囲は，塗り分け処理が終了した時点から各 GPU による要素計算処理が全て終了するまでとする．塗り分け処理を行う前に GPU へ転送可能なデータは全て転送しておき，今回の測定範囲からは除外する．また，CPU-GPU 間のデータ転送時間およびメインメモリ上で結果を統合する時間については別途測定して評価する．CUDA がバージョン 4.0 から 1CPU スレッドによる複数 GPU の制御に対応したことから，CPU による各種の処理は基本的に 1 スレッドのみを用いて実行し，結果の統合 (配列の足し合わせ) のみ OpenMP により並列化 (4 スレッド実行) した．

2GPU を用いた場合の実行時間 (色数 32) を**図 5** に示す．図 5 からわかるように，1CPU と 2GPU を比較すると実行時間はほぼ半分に削減されている．しかし，2GPU 実行時の結果統合にも無視できない時間がかかっている．今回の実装は全 GPU が本来生成する行列と同じ容量の行列データを保持しており，結果の統合には全ての行列データについて GPU から CPU へ書き戻して足し合わせる必要がある．そのため，特に問題サイズが大きい場合やより多くの GPU を用いる場合には，要素計算処理の時間は複数 GPU 化により短くなる一方で，CPU-GPU 間の転送時間と結果の足し合わせ時間は GPU 数倍に増加することが予想できる．この時間増加を抑制する方法としては，塗り分け処理の時点で各色が行列データ全体にまたがらないようにする方法が考えられる．要素と接点の構成上，色ごとに全く重複しないように塗り分けを行うことはできないが，各 GPU が更新するデータの範囲を限定できれば各 GPU が担当・保持する必要がある行列データが小さくなる．また，係数行列生成処理の前後で行われる処理を踏まえてデータ転送のタイミングを調整することも考えられる．これらの実装と性能評価は本稿の執筆時点ではまだ行えておらず，今後の課題である．

4.2　複数 GPU 複数 CPU 環境向けの実装と性能評価

本節では，複数 GPU に加えて CPU も利用して高速に要素計算処理を行うことについて述べる．

本節では具体的な問題設定として，塗り分け処理は 1CPU のみで行い，計算処理をいくつかの GPU と CPU(CPU コア) で行うことを考える．各 GPU や CPU への問題割り当てについては，前節で述べたように，単純に色ごとに担当すれば要素計算処理自体は高速化が可能であり，結果の統合も含めて性能向上させるためには塗り分け処理の工夫が必要であると考えられる．

一方，今回の問題設定では各 CPU や GPU の性能 (同じ数と構成の接点に対して要素計算処理を行うのに要する時間) が均一ではないため，最大性能を得るためには性能バランスに見合った計算量 (色数) を担当する必要がある．そのための実装としては，あらかじめ各 CPU と GPU の性能バランスを確認しておき静的に問題割り当てを行う方法や，実行時に計算していない (割り当てられた計算を終えた) CPU や GPU に対して動的に問題割り当てを行う方法が考えられる．

本節では前者の案に基づいて実装を行った．実装の概要を**図 6** に示す．利用する CPU コア数と GPU コア数の合計数分だけのスレッドを作成し，各スレッドは担当する CPU や GPU を用いて指定された割合分だけの計算を行う．本稿の実験環境には CPU が 4 コアと GPU が 2 台搭載されているが，3 スレッドを生成し，2 スレッドをそれぞれ 1 台ずつの GPU を制御するためのスレッドに割り当て，1 スレッドを CPU による計算に用いて実行時間を測定した．測定結果を**図 7** に示す．今回は逐次 CPU と GPU の性能差が大きいことから全体の 2%というわずかな色数の計算を CPU に割り当てた．実験の結果，2GPU+1CPU の実行時間は 2GPU 実装の実行時間を下回っており，GPU に加えて CPU も利用すること

© 2011 Information Processing Society of Japan

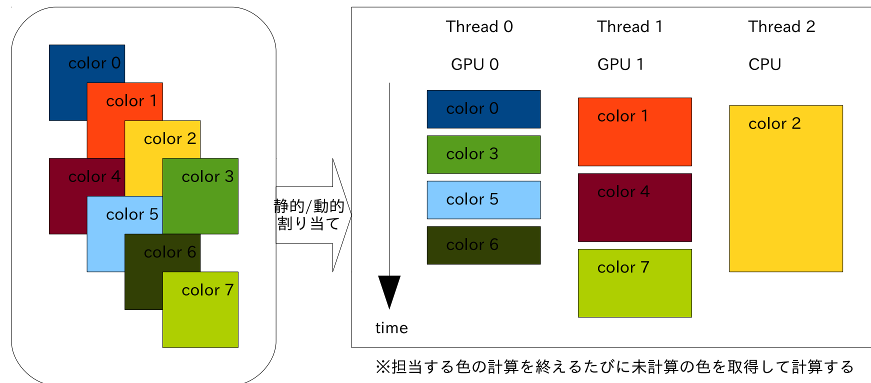

図 6　不均質な複数 CPU 複数 GPU 環境向けの実装の動作イメージ

図 7　GPU2 台と CPU1 コアを用いた際の実行時間

で GPU のみでの性能を上回ることが可能であることが確認できた．同様の手法を用いることでより多くの GPU や CPU を使った場合や，GPU と CPU の性能がそれぞれ異なる場合にも効果が得られることが期待できる．

今回の実装では CPU コアを使い切れておらず，搭載されている CPU と GPU を全て用いて最大性能を得るためには，以下に示すようないくつかの実装の改善や性能比較を行う必要があり，これらは今後の課題である：

- 性能評価においては前節同様に CPU-GPU 間のデータ転送時間を除外しており，データ転送を含めて性能を向上させる必要がある．詳細は前節に述べたとおりである．
- 今回は静的な割り当てを行ったが，動的な割り当てとの性能比較や，理論上の最大性能 (最適な割り当てを行うことができた場合の性能) との比較を行う必要がある．
- 今回は CPU コアを 1 つだけ要素計算処理に用いたが，より多くの CPU コアを活用する余地がある．各 CPU コアごとに別の色を担当するべきか同一の色を OpenMP により並列実行するかや，GPU の制御に CPU コアを占有させるべきか否かは，検討・比較の余地がある．

5. おわりに

本稿では三次元弾性静力学を対象として GPU(CUDA) を用いた有限要素法の実装を行った．特に係数行列生成に注目して実装と性能評価を行った．

1GPU を用いた実装については，小規模探索処理を CPU から GPU へ移動することで我々が文献 4) にて行った実装よりも高い性能を得た．また複数 GPU を用いる実装や，さらに CPU にも計算を割り当てる実装を考案・実装し，さらに性能向上が得られることを確認した．

本研究の今後の課題としては，3 章および 4 章の末尾に詳細を示したように，CPU-GPU 間のデータ転送を含めた高速化 (アプリケーションのより広い範囲についての GPU 化を含む) や，CPU コアの最適な利用とスケジューリングの最適化などがあげられる．

謝辞　本研究は，科学技術振興機構戦略的国際科学技術協力推進事業（共同研究型）「日本－フランス共同研究」「ポストペタスケールコンピューティングのためのフレームワークとプログラミング」の補助を受けている．

参考文献

1) NVIDIA: NVIDIA GPU Computing Developer Home Page, http://developer.nvidia.com/object/gpucomputing.html.
2) Cevahir, A., Nukada, A. and Matsuoka, S.: An Efficient Conjugate Gradient Solver on Double Precision Multi-GPU Systems, 先進的計算基盤システムシンポジウム SACSIS2009，pp.353–360 (2009).
3) Bolz, J., Farmer, I., Grinspun, E. and Schröder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid, *ACM SIGGRAPH 2003*, pp.917–924 (2003).
4) 大島聡史，　林雅江，片桐孝洋，中島研吾：三次元有限要素法アプリケーションの CUDA 向け実装と性能評価，情報処理学会研究報告 2011-HPC-129，pp.1–6 (2011).
5) GeoFEM: http://geofem.tokyo.rist.or.jp/.
6) Nakajima, K.: Parallel Iterative Solvers of GeoFEM with Selective Blocking Preconditioning for Nonlinear Contact Problems on the Earth Simulator, *ACM/IEEE Proceedings of SC2003* (2003).

© 2011 Information Processing Society of Japan