Vol.2011-HPC-129 No.10
2011/3/15

CPU/GPU を共用したヘテロジニアス環境における FMM の最適化

福 田 圭 祐[†1] 丸 山 直 也[†1] 松 岡 聡[†1,†2]

本稿は CPU-GPU ヘテロジニアス環境上の Fast Multipole Method(FMM) の最適化手法を提案する。本研究は目標として、計算特性やプロセッサ特性のモデル化に基づくタスク割り当てや最適化を目指しており、現段階では単一 CPU/単一 GPU 向けに実装を行っている。CPU-GPU 混在環境において効率的なプログラム実装を実現するには、並列度の違いや CPU-GPU 間のデータ転送量など、プログラムの特性を考慮する必要がある。FMM においては、各時間ステップは複数のそれぞれ異なる特性を持つ計算フェーズから成り立ち、フェーズ間の依存関係は複雑である。本稿では各計算フェーズの特性を分析し、GPU 上の実行に適したフェーズの特定を行った。そして FMM の派生アルゴリズムである KIFMM の既存実装を OpenMP と CUDA を用いて、CPU/GPU それぞれについて最適化を施した。実装したソフトウェアを、TSUBAME2.0 の単一ノード上の Intel Xeon CPU および NVIDIA Tesla M2050 を用いて評価し、大幅な高速化が達成されたことを示す。

Optimization of FMM on CPU-GPU heterogeneous environment

Keisuke Fukuda,[†1] Naoya Maruyama[†1] and Satoshi Matsuoka[†1,†2]

This paper proposes optimization methods of Fast Multiple Method (FMM) for CPU-GPU heterogeneous environments. Our goal is to propose optimization and task mapping methods based on modeling properties of computations and processors. Currently we focus on an implementation on a single CPU and GPU environment. In order to achieve efficient implementations on heterogeneous environments, properties of computations have to be considered such as parallelism and communication amount between CPUs and GPUs. In FMM, a time step consists of several computation phases, which have different properties and complicated dependency. We have analyzed properties of those phases and identified which phases should be ported to GPUs. Then we have implemented optimization methods for CPUs and GPUs by using OpenMP and CUDA, based on an existing implementation of KIFMM, a derived algorithm of FMM. We have evaluated our implementation on a single node of TSUBAME 2.0 with Intel Xeon CPU and NVIDIA Tesla M2050 GPU, and demonstrate that it achieves a significant speedup over the original one.

1. はじめに

Fast Multipole Method(FMM) は、Greengard ら[4] によって提案された、N 体問題向けの指定精度の $O(N)$ 計算時間アルゴリズムである。N 体問題向けのアルゴリズムとしては他に、直接計算や $O(N \log N)$ 計算量の Barnes-Hut アルゴリズム等が知られている。小規模〜中規模サイズの問題に関しては Barnes-Hut 等の手法の方が効率が良い場合が多いが、大規模問題に関しては計算オーダーが非常に重要となる。近年の計算問題の大規模化に伴い、各種の N 体問題の計算においては FMM の重要性が高まっていくと考えられる。

計算環境に関しては、近年 GPU を搭載したスーパーコンピューターが増加している。東京工業大学学術国際情報センターに設置された TSUBAME2.0 はその最たる例であり、理論性能 2.4PFlops を誇る。また、Linpack ベンチマークによる世界ランキング Top500 では、上位 5 台のうち TSUBAME を含めて 3 台が CPU と GPU を併用したものであった[1]。このような環境では、計算性能の大部分を GPU が占める場合が多い。従来の CPU 向けの分散・並列化だけではなく CPU-GPU 異種混在環境を考慮した最適化を行う必要がある。

しかし、既存の並列化・最適化手法だけでは、CPU-GPU 混在環境においては十分でない。CPU ではコアあたり 1 スレッドを前提とし、単ノード上では数〜数十スレッド程度による並列化がなされることが多い。一方、GPU は数百の SP(Streaming Processor) から構成され、数千〜数万スレッドによる並列化がしばしば行われる。また GPU では分岐処理のコストが大きいなど、アーキテクチャの違いからくる最適化手法の差異も非常に大きい。さらに、GPU 向けのデータ構造への変換やメインメモリから GPU メモリへのデータ転送のコスト等も無視できない場合がある。

とりわけ、本研究で取り上げる FMM は複数の計算フェーズからなるアルゴリズムであり、それぞれのフェーズがデータ構造・計算方法・並列性・メモリアクセス等の点から異

†1 東京工業大学
Tokyo Institute of Technology
†2 国立情報学研究所
National Institute of Informatics

© 2011 Information Processing Society of Japan

Vol.2011-HPC-129 No.10
2011/3/15

なった特徴を持っている。また、一部のフェーズでは、計算時間が入力データの性質に依存して増減する。CPU-GPU 混在環境で FMM を効率よく実行するためには、このような計算フェーズの特徴と CPU-GPU 間のデータ変換・転送のコストを考慮して CPU-GPU 間での計算の割り当てを決定する必要があるが、その方法論は明らかではない。

また、CPU と GPU はしばしば比較され、特に FMM に関しては CPU と GPU は同程度の実行性能であるであるといったような報告もなされているが[9]、しかし各種プロセッサは速いスピードで開発が進められており、将来的にも同じ傾向が続くとは限らない。現に、CPU と同じダイ上に両方の種類のプロセッサを搭載した Intel 社の Sandy Bridge 等の製品も登場してきており、異種プロセッサ混在環境を取り巻く環境は大きく変化し続けている。CPU ／ GPU に限らず、プロセッサ特性、性能、ノード構成、ノード数に適応して、自動的に処理を割り当てて最良の性能を得ることができるような手法が必要である。

そのような計算タスク割り当て手法を研究するためには、FMM の完全な CPU 実装および GPU 実装が必要である。本研究では前段階として、FMM の派生アルゴリズムである KIFMM を対象とし、フェーズごとの計算特性に着目して計算の一部を GPU で実行することにより、CPU 実装に比して計算の大幅な高速化が可能であることを示した。KIFMM においては近傍粒子同士の直接計算 (U-list phase) が全計算時間の 50%程度を占めており、この部分は従来の直接計算と同様に十分な並列性を持ち計算密度が高い。また、粒子を Multipole へと集約する際 (後述する Upward フェーズ中の S2U フェーズ) に相互作用カーネルの直接計算を用いており、同様な並列性が見られる。さらに、CPU-GPU 間のデータ変換及び転送のコストについては性能向上幅と比較して十分に小さいことがわかった。

実装としては、KIFMM の提案者である Ying ら[10] による実装である kifmm3d を変更し、一部の計算フェーズを GPU 上で行うようにした。計算カーネルは single layer laplacian カーネルを対象とし、粒子数 1,000,000 の均一分散の入力データを用いて TSUBAME2.0 の単ノード上で実行し、実行時間が大幅に削減されることを確かめた。

2. Fast Multipole Method

本節では、FMM の概要と、並列処理の観点から見た FMM の特徴、そして各フェーズの計算特性の違いについて述べる。本項で述べるアルゴリズムの数学的詳細、計算フェーズについての詳細についてはいずれも文献[4),10)] を参照されたい。

2.1 FMM の概要

FMM の基本的な考え方は、空間を再帰的に木構造に分割し、遠くの粒子を”まとめて”計算することで、全粒子同士の相互作用を近似的に $O(N)$ 時間で計算するということである。計算は、木構築・Upward・U-list・V-list・X-list・W-list・Downward の各フェーズからなる。

まず木構築フェーズでは、中心点を用いて空間を 8 等分[⋆1] し、分割されたそれぞれの空間について、粒子の個数が q 個を超える場合は再帰的に 8 等分を繰り返す。こうして分割された空間は、それぞれの葉 (または box と呼ばれる) がたかだか q 個の粒子を保持する 8 分木として表現される (図 **2.1** [⋆2])。

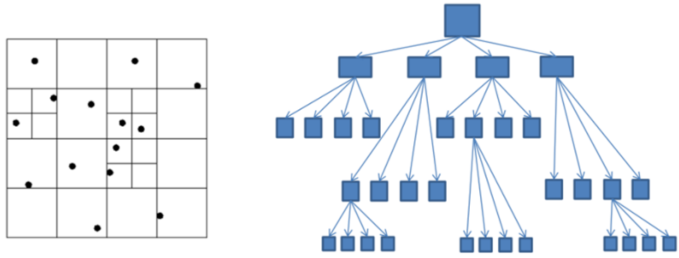

図 1 空間の再帰的分割と木表現
Fig. 1 Division of space and tree representation

次に、それぞれの box に属する粒子の情報を、親の節点 (Multipole と呼ばれる) から節点へと頂点へ向かって再帰的に集約していく。この数学的操作は Multipole Expansion、計算フェーズは Upward Computation と呼ばれる。なお、Greengard らによる FMM と Ying らによる KIFMM の主な違いはこの数学的操作にある。FMM では球面調和関数を用いてカーネル関数を数学的に級数展開することによって粒子をまとめることを可能にしている。一方、KIFMM ではポテンシャル理論の成果を用いて、一定の条件を満たす場合に空間を包み込む曲面上に分布した density による作用と内部の点の density による作用の和が同じになるという定理を用いている。

次に、計算する粒子間の距離に応じて 4 種類からなる評価フェーズがある。

まず、近傍の粒子同士の相互作用については近似計算を用いることができないので、直接計算を行う。このフェーズは U-list フェーズと呼ばれる。元の空間上で隣り合っている box

⋆1 1 次元空間においては 2 等分、2 次元空間においては 4 等分である
⋆2 簡単のため、図は 2 次元の場合を示した

© 2011 Information Processing Society of Japan

同士の全粒子同士の直接計算が行われる (図 **2.1**)。

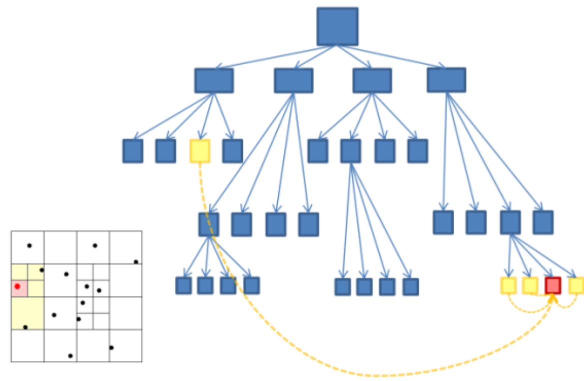

図 2 U-list フェーズの計算例
Fig. 2 Example of U-list phase computation

次に、V-list と呼ばれる計算フェーズでは、親が同じレベルであるような節または box 同士 (ただし互いが U-list である場合をのぞく) の計算が行われる。**図 2.1** は、V-list フェーズによって節から節へ相互作用が計算され、それが後述の Downward フェーズによって粒子へ伝播される様子を示している。このフェーズでは、KIFMM においては小さなサイズの FFT が行われる。

その他に W-list、X-list と呼ばれる計算フェーズが存在する。これらのフェーズは、粒子の分散が不均一であり、木構造が平衡木でない場合の処理を行うフェーズである。本研究では均一分散のみを扱い、これらのフェーズは対象としなかった。

最後に、Downward フェーズと呼ばれるフェーズによって、節同士で行われた相互作用の結果が親ノードから子ノードへと展開され、最終的に末端の個々の粒子の値へと伝播される。

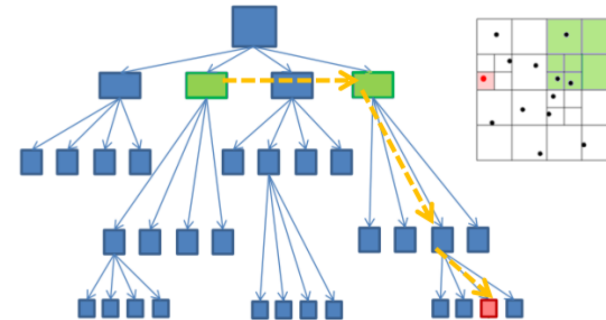

図 3 V-list フェーズの計算例
Fig. 3 Example of V-list phase computation

2.2 並列処理の観点から見た FMM の特徴

上述の各フェーズについて、並列性の観点から特徴を記述する。

まず、空間の分割と木の構築に関しては、木構造を上から辿っていく処理に相当する。つまり、1 回目の分割では並列度 1, 2 回目の分割では並列度 8... のようになる。

次に、Upward フェーズは 2 段階に大別される。個別の粒子から box へと集約する S2U フェーズ、木の節から節 (つまり Multipole から Multipole) へと集約する M2M フェーズである。KIFMM における S2U フェーズは、box ごとに用意した数百個 (精度に依存する) の仮想的な粒子と box 内に存在する実在の粒子との直接作用として計算される。通常、q、仮想粒子数ともに数百〜個程度であり、同時にすべての box の S2U は互いに独立であるので、U-list に準ずる十分な並列性があると言える。一方、M2M フェーズについては box および節の数は粒子数ほど多くはなく、しかも木を上方向に進むに従って並列性が減少していく。

U-list フェーズについては、すでに部分的に述べたように、(1) それぞれの各 box 組の計算は全て互いに独立であり (2) 各 box 組の計算の中で、粒子同士の相互作用はすべて独立である。つまり、U-list フェーズは、従来の直接計算と全く同様に細粒度の十分な並列性を持っている。さらに、木の構築が終了後、Upward フェーズを初めとする各フェーズとも独立である。

© 2011 Information Processing Society of Japan

次に、V-list について述べる。V-list は、節から節への水平方向の計算である。V-list 同士の計算はすべて独立であるものの、それぞれの節での粒子の情報が必要であることから、Upward フェーズの計算進捗に依存する。つまり、Upward フェーズがレベル l に到達したとき、Upward フェーズのレベル l - 1 以降と V-list のレベル l は並列に計算することが可能である。

Downward フェーズについても、Upward フェーズと同様 (ただし方向が逆) の並列性を持っている。

2.3 各フェーズの計算特性

前述のように、各フェーズはそれぞれ異なった計算特性を持っている。

空間の分割と木の構築操作は、粒子の並べ替えを伴うため、多量のメモリアクセスと分岐処理を含む。また、Upward,Downward フェーズは、木構造をたどる処理であり、各フェーズはカーネルの評価及び小規模の行列-ベクトル積である。

もっとも計算特性に違いがあるのは U-list と V-list である。前述のとおり U-list は部分的な直接計算であり、高い並列性と高い計算密度を持つ。一方、V-list は小規模な FFT と小規模な行列-ベクトル積からなる。

本研究では対象としなかったが、W-list と X-list の計算量は、入力データである粒子の配置に依存する。入力粒子が均一分散であるような場合には、W-list と X-list の計算量は 0 である。

具体的な計算量の詳細を**表 1** に示した。N は粒子数、q は box 内の最大粒子数、M は box 数で $M \approx N/q$ である。p はユーザーが指定する許容誤差を示す値で、4 で低精度、6 で中精度、8 で高精度となっている。

表 1 各フェーズの計算量
Table 1 Computational complexity of each phase

Phase	Complexity
Upward	$O(Np + Mp^2)$
U-list	$O(27Nq)$
V-list	$O(Mp^{3/2} \log p + 189Mp^{3/2})$
X-list	very small for uniform distribution $O(Nq)$ for otherwirse
W-list	very small for uniform distribution $O(Nq)$ for otherwirse
Downward	$O(Np + Mp^2)$

3. 最適化と実装の方針

3.1 方　　針

本研究の目的は、複数の性質の異なる計算フェーズからなる FMM を、CPU-GPU 混在環境において最適なタスク割り当てによって実行することである。そのためには、まず CPU、GPU それぞれにおける効率的な実装を用意する必要がある。

本稿では、Ying ら[10] による CPU 上での逐次実装に変更を加え、CPU での OpenMP を用いた並列化及び GPU 上での並列化により、逐次実装と比較して大幅な高速化を達成できることを示す。具体的には、Upward フェーズのうちの Upward(S2U)、U-list を対象とした。これらのフェーズは、逐次実装において実行時間の大きな割合を占めているからである (**図 3.1**)。

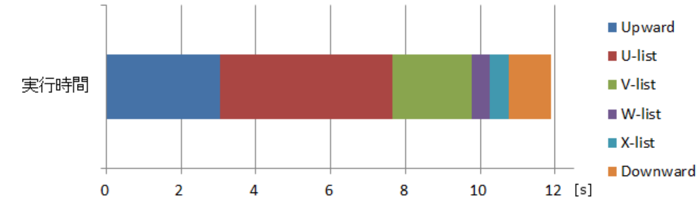

図 4 逐次実装における各フェーズの実行時間比
Fig. 4 Time of each phase in serial implementation

また、U-list フェーズの GPU 実装については、直接計算における計算規模は大きいほうが効率が良い。よって、前述したパラメーター q は、大きいほうが CPU に対しての高速化の幅が大きくなることが期待される。さらに、q を大きくすることは空間の分割数を減らし木の高さを低くすることを意味する。これは U-list 以外の計算フェーズ、すなわち Upward,V-list, X-list,W-list,Downward の計算量が減ることを意味する。ここでは、予備評価として**図 3.1** を挙げる。GPU によって高速化した実装において、q の値を変化させて U-list とその他のフェーズの計算時間を測定した結果である。q を大きくしたとき、U-list フェーズの計算時間の上昇幅と比して、他フェーズの計算時間は大きく減少していることがわかる。これにより、全体の実行時間も大きく減少する。これは、CPU から GPU への負荷の移動と捉えることもできる。本研究ではこの q の値を手動で設定したが、このパラメーターが、タスク割り当ての最適化と計算時間の短縮において大きな役割を果たすことがわかり、設定の自動化が要求される。

© 2011 Information Processing Society of Japan

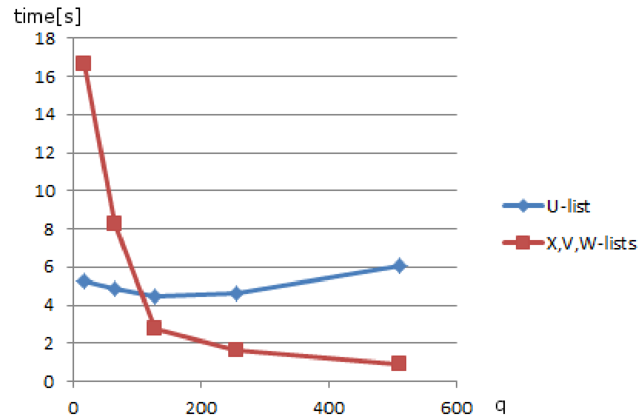

図 5　様々な q での U-list と他フェーズの計算時間の対比
Fig. 5　U-phase vs. Other phases on various q

また、今回 GPU による実装を見送った V-list については、FFT のライブラリである FFTW の利用法に関する最適化を行なった。具体的には、計算ごとに FFTW の plan を作成していたところ、注意深く plan の再利用を図ることによって OpenMP 実装・GPU 実装の双方に共通して高速化を達成することができた。

3.2　CPU 実装

CPU 上での並列化には、OpenMP を用い、計算フェーズ内の依存性に留意しつつプログラムに存在する for ループを並列化した。本稿では、将来的な研究に向け、CPU,GPU それぞれについてシングルソケットでの効率的な実装を与えることが目的であるため、CPU1 コアに対して 1 スレッドが割り当てられるように制御した。この点についての詳細は評価の項で述べる。

Ying らによる実装では、box 同士の相互作用を計算する際に、DGEMV 関数を用いている。具体的には、target box 内の粒子を t_i、source box 内の粒子を s_j とし、カーネル K を適用した結果の相互作用の行列が $\{I_{ij}\} = \{K(t_i, s_j)\}$ と表現できる。これに各 source 粒子の density [*1]である (d_j^s) ベクトルをかけることによって、target 粒子 t_i の density に加算される d_i^t が計算される。これは確かに (1) 式のように行列-ベクトル積で表すことができる。

$$\boldsymbol{d}^t = \boldsymbol{I}\boldsymbol{d}^s \tag{1}$$

しかし、DGEMV 呼び出しのオーバーヘッド等を考慮すると、カーネル計算時に同時に行なったほうが効率が良いことがわかったので、本実装においてはそのように変更を施している。

3.3　GPU 実装

本節では、特に GPU 実装の詳細について記述する。GPU 実装は、開発環境 CUDA を用いて NVIDIA 社の GPU 上で開発した。

まず、U-list フェーズの計算について述べる。前述のように、U-list は、近傍 box 同士の直接計算である。ここでは、影響を受ける側の box の事を target box、影響を与える側の box のことを source box と呼ぶことにする[*2]。U-list の全計算は、target box についてのループとみなすこともできる。それぞれの target box について、その U-list を構成する source box からの影響を計算して足しあわせる。GPU 上での U-list 計算を実装するにあたって注意すべきは、複数の source box から 1 つの target box に書き込みが発生するので何らかの同期が必要であること、またそれぞれの target box の U-list がどの box であるかを GPU に渡すことが必要となるということである。また、CUDA におけるグリッドサイズ、スレッドブロックサイズの上限も勘案する必要がある。

なお、CPU での並列化と同様にカーネルによる相互作用の計算と別に DGEMV をおこなっているような操作については、相互作用と DGEMV を同時に実行するようにした。

そこで、本稿では、1 つの target box に関する計算を 1 つのスレッドブロックが行う形式を取り、スレッドブロック内では、個々の粒子に 1 つ 1 つのスレッドが割り当てられるような形式を取った。それぞれのスレッドブロックには、対象となる target box の U-list となる source box のリストが渡され、ループを用いて全ての source box 内の粒子について計算を行う。これにより、個々の粒子への計算は 1 つのスレッドが責任を持つので、書き込みに関する同期の必要がない。グリッドサイズは target box の個数とした。これにより、全 target box について計算が行われる。なお、グリッド・スレッドブロックともにサイズは 1 次元である。

次に、Upward フェーズのうちの S2U フェーズについて述べる。これは、box に属する

*1 density は、電荷粒子でいえば電荷、重力問題で言えば質量となる物理量である

*2 全ての box の組について、互いに target box である場合と source box である場合が対応していることに注意されたい

© 2011 Information Processing Society of Japan

粒子から box へと粒子をまとめる処理である。Upward フェーズの中でも、対象とする粒子数、box 数が非常に多く、大きな割合を占めていることが 図 3.1 からわかる。

ここで行われている計算は、数学的には box を取り囲む曲面上に分布する density を離散的に近似する処理であるが、手続きとしては U-list と類似している。box の中心座標から作成した要求精度に依存する個数の仮想的な点に対して box 内の粒子から直接計算と同じ操作を行う。仮想的な target 点群の個数は、低精度で 150 程度、高精度で 450 程度であり、十分な並列性を持っている。よって、全ての box に対して全て長さが 1 の U-list を作成することによって、U-list フェーズで用いたルーチンを利用して計算を行うことができる。

なお、NVIDIA 社が提供している CUDA SDK に含まれるマニュアル GPUGems3 では、N 体問題の GPU 実装の最適化手法として shared メモリを用いる方法が示されている[8)]。Fermi 以前の世代の GPU においては、shared メモリを用いることがほぼ必須であった。しかし、ハードウェアキャッシュを備える Fermi 世代では shared メモリを利用するための分岐命令によってかえって速度が減少する場合もあり、かならずしも用いたほうが高速になるとは限らない。今回我々が用いたのは Fermi 世代の GPU であるため、shared メモリは用いなかった。ただし、注意深く shared メモリを用いることにより、さらなる最適化が達成できる可能性は残っていると考えられる。

4. 評　　価

オリジナルの逐次実装、OpenMP による並列実装、GPU による並列実装のそれぞれについて、TSUBAME2.0 の単一ノードを用いて性能を評価した。具体的な環境は **表 2** に示した。

本稿では、OpenMP を用いた CPU 並列化実装と GPU と OpenMP を組合せた並列化実装を比較しているが、我々は CPU と GPU の比較においてはソケット単位での比較が公平であると考えている。CPU では Hyper Threading が有効とされているため、プロセスからは 12 core の CPU が 2 つあるかのように見える。評価では、1 ソケット上の 1 core あたり 1 スレッドが確実に割り当てられるように制御した。

表 2　評価環境
Table 2　Environment for evaluation

	CPU	GPU
Type	Intel 6 core Xeon X5670 × 2	NVIDIA Tesla M2050 × 3
Freq	2.93GHz	1.15GHz
Cores	6 x 2 (with HT)	14SM/448 cores
Memory	54GB	3GB

最適化実装の比較対象とする逐次実装としては、我々の OpenMP 実装でスレッド数を 1 としたものを用いた。いずれも Ying らによる実装が元となってはいるが、前述したように FTTW の plan の再利用がされていないなど、非常に基礎的な手法での最適化の余地が大きかったため、対象としないこととした。評価した粒子数は 1,000,000、粒子の分散は均一分散を用いた。

まず、逐次実装の、q の値の変化による性能変化を (**図 4**) に示す。

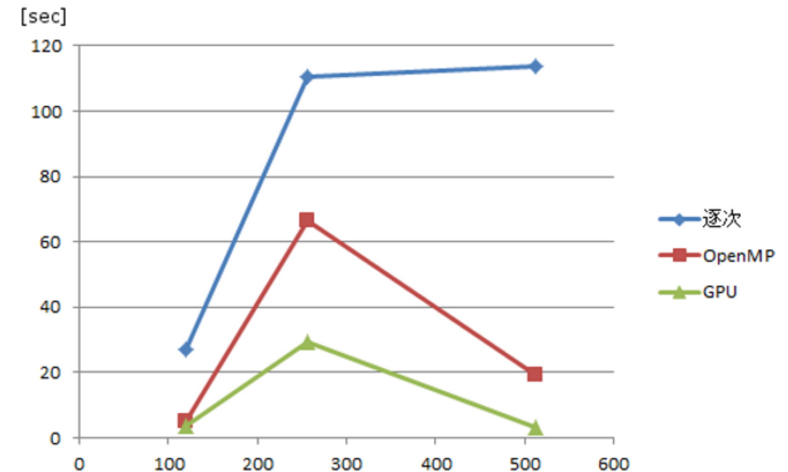

図 6　いくつかの q での総実行時間の変化の比較
Fig. 6　Processing time of the 3 implementation on various q

今回の実装では、$q = 256$ の時において、均一分散においてはほぼ発生しないはずの W-list、X-list の計算時間が大幅に増えており、原因は不明であるが実装上のバグだと思われる。$q = 120$, $q = 512$ についてはこの現象は発生していない。この 2 つの点について見てみると、OpenMP 版、GPU 版の実装が、それぞれ逐次版に対して大幅な高速化を達成していることがわかる。具体的には、$q = 120$ においては、逐次版に対して OpenMP 版が約 5.3 倍、GPU 版が約 7.67 倍、$q = 512$ においては OpenMP 版が約 5.9 倍、GPU 版が約 35 倍の高速化をそれぞれ達成した。また、OpenMP 版と GPU 版を比較すると、今回の比

ⓒ 2011 Information Processing Society of Japan

較の範囲では $q = 120$ の時が両者とも全体時間としては最短となり、その時の計算時間を比較すると、GPU 版が約 1.4 倍高速となった。

最後に、それぞれの q における計算時間の内訳を示す。

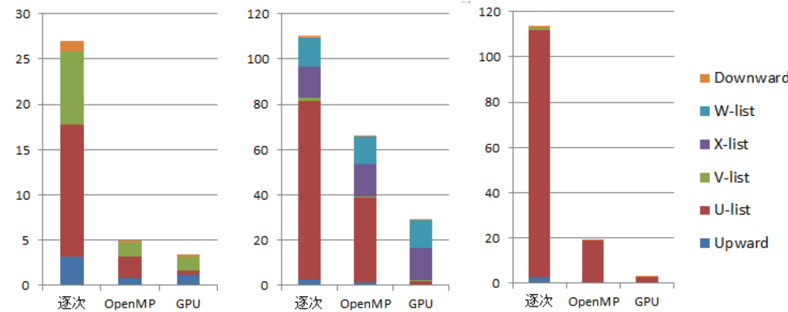

図 7　いくつかの q における実行時間の内訳
Fig. 7　Breakdown of execution time on varying q

q が大きくなるに従って、全体に占める U-list フェーズの割合が支配的になり他のフェーズの計算時間は減少することが確認でき、その場合でも GPU を用いた実装は全体として実行時間が抑えられていることが観察できた。

5. 関 連 研 究

FMM は、Greengard ら[4] によって最初に提案された。その後、いくつかの改良や変種が提案されたが、最も広く知られているのは本稿でも取り上げた Ying ら[7] による KIFMM であろう。FMM と KIFMM の違いは、粒子を box からノードへとまとめていく際の手法にある。

Lashuk らは、GPU 上での KIFMM の実装に取り組み、逐次実装に対して 30 倍を超える大幅な高速化を得ている[7]。Chandramowlishwaran らは、KIFMM を複数種類のマルチコア CPU 上で高速化する研究を行った[2),3]。また、それらの成果を総合し、Vuduc らは、KIFMM においては高度に Nehalem 等のプロセッサ上で高度に最適化された CPU 実装は GPU 実装に匹敵するとの見解を示した[9]。本稿での GPU 実装については、Lashuk らの成果を参考にした部分が多い。一方、本稿では Fermi 世代の GPU に適した最適化を行っている点、また本研究の目標が単に CPU および GPU での最適化実装を与えることではなく、ヘテロジニアス環境上でのタスク割り当ての最適化手法の提案を目指している点などが異なる。

横田らは、FMM を応用して GPU クラスタ上での乱流解析を実装した[11]。

FMM の他に N 体問題の高速化法として広く用いられているアルゴリズムに、$O(N \log N)$ 計算量である Barnes-Hut の Tree アルゴリズムがある。Jetley らは、Barnes-Hut アルゴリズムの実装である ChaNGa を GPU 上に移植し、その性能を調べた[6]。Hamada らは、安価に構築した大規模な GPU クラスタ上で Barnes-Hut アルゴリズムを実装し、2009 年の ACM ゴードン・ベル賞を受賞している[5]。

6. まとめと今後

本稿では、CPU-GPU 異種混在環境上での FMM のタスク最適割り当ての方法論を得ることを目標として、その前段階として既存の逐次 FMM 実装に対して GPU および OpenMP による並列化を行い、大幅な性能向上が得られることを確認した。また、FMM の各フェーズにおける並列性・性能向上の可能性および障害について各種の知見を得た。

今後は、関連研究で示されている知見も含めて、可能な限りの CPU・GPU 上での高性能な並列化実装を実装することを目指す。その後、その実装を利用し、CPU-GPU 異種混在環境上での最適なタスク割り当て戦略の提案を目指し研究を進めていく。

参 考 文 献

1) Top500 supercomputing sites, September 2010. `http://www.top500.org/`.
2) A.Chandramowlishwaran, S.Williams, L.Oliker, I.Lashuk, G.Biros, and R.Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, 2010.
3) Aparna Chandramowlishwarany, Kamesh Madduri, and Richard Vuduc. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–12, Washington, DC, USA, 2010. IEEE Computer Society.
4) L.Greengard and V.Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, Vol.73, pp. 325–348, December 1987.
5) Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High*

© 2011 Information Processing Society of Japan

Performance Computing Networking, Storage and Analysis, SC '09, pp. 62:1–62:12, New York, NY, USA, 2009. ACM.

6) Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and ThomasR. Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

7) Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp. 58:1–58:12, New York, NY, USA, 2009. ACM.

8) Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with cuda. In *GPU Gems 3*. 2007.

9) Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pp. 13–13, Berkeley, CA, USA, 2010. USENIX Association.

10) Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, Vol. 196, No.2, pp. 591 – 626, 2004.

11) R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, and K. Yasuoka. Fast multipole methods on a cluster of gpus for the meshless simulation of turbulence. *Computer Physics Communications*, Vol. 180, No.11, pp. 2066 – 2078, 2009.

© 2011 Information Processing Society of Japan