重力多体系用Tree Codeの並列GPU化による計算加速

扇谷　豪[1,2,a)]　三木　洋平[1,2]　朴　泰祐[1,3]　森　正夫[2,3]　中里　直人[4,3]

概要：「ツリー法」は高効率に粒子間の重力計算を行うことのできるアルゴリズムであり，計算天文学の分野で広く用いられる．本研究では，GPU (Graphics Processing Unit) によってツリーコードを加速する方法を検討する．本論文では，Warp 内での分岐回数を減少させ，メモリ空間上での粒子データの配置に工夫をすることで高速化を可能にする手法を提案する．これらを実装し，先行研究の提案手法を用いた場合に比べてカーネル関数を 4 倍高速にした．さらに，これを MPI を用いて並列化することで，10^8 個以上の粒子を用いた大規模計算の実行を可能とする計算性能を達成した．

キーワード：GPU，N 体シミュレーション，ツリー法

Speed-up of a Tree Code for Gravitational Systems by Multi-GPU Parallelization

Go Ogiya[1,2,a)]　Yohei Miki[1,2]　Taisuke Boku[1,3]　Masao Mori[2,3]　Naohito Nakasato[4,3]

Abstract: Since the Tree Method is an efficient algorithm to compute gravity among particles, it is widely utilized in computational astrophysics. In this paper, we speed up it by using GPUs (Graphics Processing Units). We propose expedients to reduce the frequency of warp branching and to achieve high cache hit rate. We implement them and succeed to make the kernel function 4 times faster compared to the methods proposed in the previous work. Furthermore, we parallelize our code using MPI and achieve the suitable performance for the large scale problems in which the number of particles is larger than 10^8.

Keywords: GPU, N-body simulation, Tree method

1. はじめに

天文学において重力は最も重要な物理過程の一つである．自己重力系を成す銀河等の天体の形成や進化を数値的に調べる方法として，N 体シミュレーションが一般的である [1] [2] [3]．N 体シミュレーションでは，系を N 個の粒子によって離散化して表現する．そして，各粒子に働く重力を計算し，その軌道を追う．多くの場合には 1 つの星と 1 粒子を対応させることは要求される莫大な計算量・記憶容量のため不可能である．1 粒子は系の総質量を M，粒子数を N とすると，(M/N) の質量をもつ仮想的な天体や物質として扱われる．

より高い空間分解能を得るため，および，離散化に起因する人工的な計算結果の劣化を防ぐため，可能な限り大きな N を用いた計算が望まれる．一方，計算コストは N に依存して増大する．最も基本的で正確な重力計算法は直接法である．これは，各粒子ついて，$(N$-1) 個の他粒子から及ぼされる重力を足し合わせる方法で，計算コストは $O[N^2]$ と膨大になり，大規模計算に向かない．そこで，計算コストを $O[N \log(N)]$ に減少させる，高効率な重力計算法「ツ

1　筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
2　筑波大学大学院数理物質科学研究科
Graduate School of Pure and Applied Sciences, University of Tsukuba
3　筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba
4　会津大学　コンピュータ理工学部
School of Computer Science and Engineering, University of Aizu
a)　ogiya@ccs.tsukuba.ac.jp

リー法」が計算天文学の分野で広く用いられる [4].

高速な演算加速機として注目を集める GPU (Graphics Processing Unit) を用いた N 体計算の高速化は GPGPU (General-Purpose computation on GPU) の黎明期から盛んに行われ，特に直接法において成果が上げられている [5] [6]．また，近年 NVIDIA 社提供の CUDA[7] や，クロノス・グループにより策定された OpenCL[8] など，開発環境の整備が進み，ツリー法等のより複雑なアルゴリズムについても高速化に成功している [9] [10] [11] [12] [13].

本研究では Nakasato (2012)[10] (以降 N12 とする) の提案手法を発展させ，ツリー法の更なる高速化を目指す．MPI を用いた並列化の方針も提案し，実装の進捗状況を報告する．本研究の開発環境は CUDA で，NVIDIA 社製 GPU を用いて議論するが，多くの内容は他の環境においても適用可能である．

本論文の構成は次の通りである．まず §2 でツリー法の概要を説明し、§3 で N12 の提案手法を紹介する．§4 ではカーネル関数の加速手法を提案し，§5 で 1GPU を用いたカーネル関数の計算性能評価を行う．§6 では CPU 担当部分を高速化する．§7 では MPI による並列化手法を説明し，§8 でそれを実装したコードの計算性能評価をする．また，$ 9 では他の研究との関連性を議論する．最後に §10 で全体を総括し，今後の予定・展望について述べる．

2. ツリー法の概要

ツリー法による重力計算は次の 2 つのプロセスから成る．

2.1 ツリー構築 (Tree Construction)

粒子データからその木構造 (一般的には 8 分木構造) を構築する．まず全粒子が収まる大きさの立方体 (根セル) を用意し，それを再帰的に等分割する．基本的には，セル内の粒子が 1 個以下になるまでこれを続ける (最下層セル=粒子となる)．各セルの座標は含まれる粒子群の重心とする．

次に，セルと粒子を正しく結びつける．ここで分割前のセルを親セル，自らを分割したものを子セルとする．各セルを自らの子セルの内一つ (More ポインタ) と，同じ親セルを持つセル (兄弟セル) の内の一つ (Next ポインタ) と結びつけることで粒子のツリー構造が完成する．他の兄弟セル同士が既にリンクしている場合には，Next ポインタは親セルの兄弟セルにつなげる．

2.2 ツリー走査 (Tree Traversal)

それぞれの重力計算をされる粒子 (i-粒子) について，根セルから順にポインタをたどり，ツリー全体を走査する．ツリー走査の際，たどり着いたセル (j-セル) と i-粒子の距離をその都度測定し，j-セルが i-粒子から見て遠いか否かの判定 (以降「ツリー判定」，または，単に「判定」と呼ぶ) をする．この時，近いならば i-粒子は次に More ポインタをたどり，遠ければ現在のセルから i-粒子に働く重力を計算した後，Next ポインタの指すセルに進む．つまり，ある j-セルが遠いと判定され，重力が計算されることは，その子セル，孫セル等である複数の粒子から i-粒子へ働く重力を，1 つの重い粒子からの寄与に置き換えたことになる．ツリーはおよそ $\log(N)$ 段の階層を持ち，これを N 個の i-粒子に対して行うので，計算コストは $O[N \log(N)]$ となる．

ツリー判定には様々な手法があるが，本研究ではパラメータ θ を用いた一般的なものを採用する．この方法では，i-粒子と各セルの重心間の距離 d とセルの大きさ l，各セルの中心と重心の距離 s を用いて，

$$l/\theta + s \equiv D_{\mathrm{crit}} < d \tag{1}$$

を満たした場合に「遠い」と判定する．これより，θ はセルの見込み角に対応し，小さくとるとより厳しい判定をする (計算量は大きくなるが，精度が向上する) ことがわかる．また，θ は $0 < \theta \leq 1$ を満たす定数とする．ここで，各セルの D_{crit} はツリー構築時に計算し，記憶しておく．

3. Nakasato (2012) の概説

ここでは N12 により提案された，GPU を用いたツリー法の加速法について簡単に述べる．N12 ではツリー構築を CPU，ツリー走査を GPU が担当する方針がとられた．後者の計算量は前者に比べ大きい．

GPU のメモリは，数層から成る階層構造をしており，それぞれ通信速度と容量のトレードオフがある．N12 で注目されたのは，小容量であるが高速な L1 cache memory の有効活用である．つまり，いかにしてキャッシュヒット率を上げるかが重要視された．直接法など，メモリへのアクセスパターンがあらかじめわかるアルゴリズムであれば，高速でかつ当該ブロック内で共有の Shared memory の利用が有効である．しかし，ツリー法ではそれを知ることはできず，Shared memory の活用が難しい．ただし，例えば NVIDIA 社製の Fermi アーキテクチャでは，64KB の高速な RAM を L1 cache と Shared memory で分けて使用する (48KB+16KB に分割．デフォルトでは Shared memory 48KB だが，CUDA を用いる場合切り替えが可能) ので，高い L1 cache ヒット率が達成できれば，Shared memory を有効活用した場合と同程度の計算性能が得られると期待できる．

N12 で提案されたのは，各スレッドが 1 つずつの i-粒子のツリー走査を担当する方法である．L1 cache ヒット率を上げるには，同一ブロック内のスレッドに同様なツリーデータへのアクセスパターンを持たせれば良い．つまり，ツリー上で同様な判定をし，同様なツリーのたどり方をすると期待される，近傍に位置する i-粒子をブロック内に集めることが重要である．そこで N12 では，空間充填曲線の一種である Morton 曲線 (図 1) を利用し，各粒子に 1 次元

的にインデックスを与え (本研究で後に用いるインデックス計算法は [12] と同様)，それに従ってメモリ空間上で粒子の並び替えをすることにより，これを達成した．N12 ではこの並び替えによってカーネル関数が約 2 倍高速化された．また，CPU が担当するツリー構築に関しても，キャッシュヒット率向上のため，約 2 倍高速化されている．

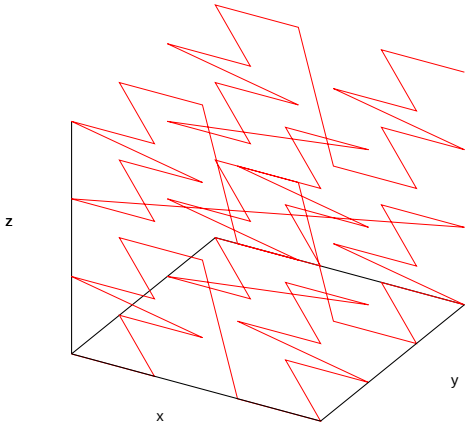

図 1 Morton 曲線.

4. カーネル関数高速化の提案手法

本研究では，§3 で紹介した N12 の手法を発展させ，さらなるツリー法の高速化を目指す．

4.1 ベクトル化・グループ化

GPU による計算において，その計算性能を著しく低下させる原因の一つとして，Warp 分岐 (Warp Branch) が挙げられる．本研究で用いる Fermi アーキテクチャは，32 スレッドが Warp という単位を成し，同時に動作する．Warp 内で条件分岐により A 種類の処理が発生したとすると，32 スレッドは A 種全ての処理を行い，自らに該当する結果のみを採用する．つまり，他の (A-1) 種の不要な処理をも行うため，性能低下の要因となる．そのため，カーネル関数の計算性能向上には，可能な限り Warp 分岐を防ぐ必要がある．

ツリー走査中の Warp 分岐は，主に「More ポインタと Next ポインタのどちらに進むか」で発生する．次に進む先が More ポインタの場合には、各スレッドは More ポインタの指すセルを新たに j-セルとするだけで良い．一方，Next ポインタの場合には，各スレッドは現在の j-セルから担当する i-粒子に働く重力を計算した後に，新たな j-セルに進む．この Warp 分岐が頻繁に起こると，More ポインタを指すスレッドは不要でかつ高コストな重力計算を行うこととなり，計算性能が著しく低下すると考えられる．

また，GPU 上で最も大容量ではあるが，最も低速なグローバルメモリへのアクセス回数を必要最低限にとどめることも，性能向上のためには重要である．

そこで本研究では，複数の i-粒子のツリー判定を統一し，Warp 分岐・メモリアクセス回数を減少させることでカーネル関数の加速を図る．

4.1.1 ベクトル化

N12 では各スレッドが担当する i-粒子数を 1 としていた．本研究では，1 スレッドあたりの i-粒子数を V 個とし，V 個の i-粒子間でツリー判定を統一する．具体的には次のように行う．まず，各スレッドは V 個の担当 i-粒子と j-セル間の距離をそれぞれの i-粒子について測定する．次に，V 個の i-粒子の中で最も j-セルに近い粒子と j-セル間の距離を用いて判定をする．つまり，V 個の粒子の間で最も厳しくなるように判定をする．以降これをベクトル化と呼ぶ．

V を大きくすると，重力計算を完了するまでの総 Warp 分岐数は減少すると期待される．また，j-セルのデータをレジスタに保持することで，同一スレッド内の複数の i-粒子が重力計算をする際にそれを共有できるため，メモリアクセス回数が減少することも期待できる．理想的な場合には，これらは $1/V$ になると予想できる．一方で，V を大きくすることは 1 スレッドがより多くの i-粒子を担当することであり，また，より厳しい判定をする (兄弟セルに移動しにくい) ため，スレッドあたりの計算量は増大する．加えて，Warp 分裂が発生した場合に重力計算をしないスレッドに対しては V 個の i-粒子の重力計算はオーバーヘッドとなるが，それも V に比例して増大する．したがって，最高性能が得られる V の最適値が存在すると考えられる．

4.1.2 グループ化

Warp 分岐数・メモリアクセス回数を減少させるには，多くの i-粒子のツリー判定を統一したい．しかし先に述べたように，ベクトル化だけでは判定を統一される i-粒子数をそれほど大きくできないであろう．そこで，スレッド内だけでなく，同一ブロック内の G 個のスレッドの i-粒子の判定も統一することで，その数を増加させる．本研究ではこの方法をグループ化と呼ぶ．ベクトル化・グループ化は単体でも使用可能であるが，以降これらを組み合わせて用いることを前提に議論を進める．

グループ化は以下のように行う. まずベクトル化と同様に各スレッドがそれぞれ V 個の担当 i-粒子と j-セル間の距離を求める．次にその中で最小の値を求め，それを Shared memory にコピーする．最後に Shared memory を参照し，G 個のグループの中での最小の距離の値 $R_{\min}$ を共有し，その値を用いてツリー判定をする．ここで注意したいのは，グループ化される G 個のスレッドは同時に同じ j-セルに対しての判定を行うということである．

ベクトル化と同様に，G を大きく設定すると，ブロック内での Warp 分岐数は減少すると期待される．さらに，CUDA ではメモリへのアクセスが 16 スレッド (half Warp)

単位で行われるため，グループ化を行い，同一の j-セルのデータへアクセスすることにより，その命令回数を減らすことができる．しかし，$R_{\min}$ を求める際の計算量は G に比例して大きくなる．また，G を大きくすることはより厳しい判定をし，計算量を増加させる．したがって，G にも高性能を得るための最適値が存在すると考えられる．

ここまでの議論をまとめると，ベクトル化・グループ化を組み合わせた場合，$(V \times G)$ 個の i-粒子のツリー判定を統一するため，Warp 分岐による計算性能の低下を妨げることが可能になる．さらに，メモリアクセス回数も減少すると期待される．一方で，より厳しい判定をするため，計算量は増加する．また，Warp 分岐が生じた場合のオーバーヘッドが大きくなる．このようなトレード・オフのため，高性能が得られる適切な (V, G) のペアが存在すると考えられる．

4.2 Peano-Hilbert 曲線

N12 で行われた，Morton 曲線に従って近い座値標を持つ粒子をメモリ空間上でも近傍に配置し直す手法は，キャッシュヒット率を上げるだけでなく，似たツリーのたどり方をする粒子を Warp 内に集め，Warp 分岐数を減らすという点でも有効であると考えられる．

3 次元分布する粒子を 1 次元的に管理するために利用される空間充填曲線として，Morton 曲線の他に Peano-Hilbert (PH) 曲線が挙げられる (図 2)．Morton 曲線と比較したとき，PH 曲線の利点は図 1，2 を見比べてわかるように，Morton 曲線にある空間的な「飛び」がない点である．Morton 曲線は Z を繰り返し書くように描かれるので，直交空間上で必ず対角線上へのジャンプが生じる．つまり，このときメモリ上で近傍に位置する粒子が，空間座標ではそうでなくなる．一方で，PH 曲線は必ず接する領域へとつながるので，メモリ空間と空間座標でのギャップはより小さくなると考えられる．PH 曲線のインデックス計算には，Lam & Shapiro (1994)[14] の提案手法を 3 次元空間に拡張したものを用いた．

したがって，PH 曲線を用いることで，キャッシュヒット率向上，Warp 分岐数の減少の両方が期待される．その効果は特にベクトル化・グループ化をしたときに大きくなると考えられる．Morton 曲線を用いた場合，1 つのスレッド，またはグループ内でのツリー走査パターンが、先述の「飛び」部分を跨いでしまう可能性がある。そのような場合には必要以上に厳しいツリー判定をし，計算量を増加させる．PH 曲線には飛びがないため，より効率的にツリー走査をすることができる．また同じ理由で，計算天文学で扱われる不均一な粒子分布の問題には PH 曲線の方が適している．なお，PH 曲線は Morton 曲線に比べインデックス計算アルゴリズムが複雑で，その計算のコストは 3 倍程度高い．ただし，後に述べるようにこれはコード全体の性能に対して問題とならず，恩恵の方が大きい．

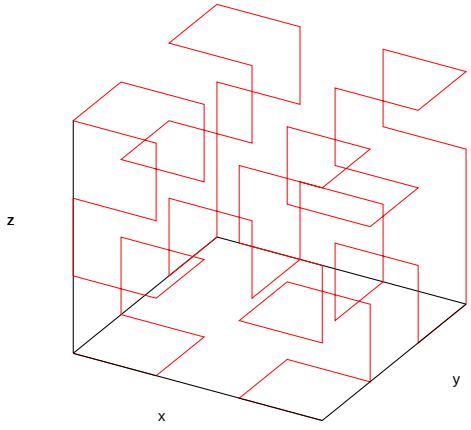

図 2 Peano-Hilbert 曲線.

5. カーネル関数の計算性能評価

ベクトル化・グループ化，および，PH 曲線に基づくアルゴリズムを GPU 上にカーネル関数として実装し、その性能を評価する．

5.1 測定環境・問題設定

実験は筑波大学に 2012 年に導入された大規模 GPU クラスタ，HA-PACS を用いて行う．測定に用いた環境の概要を表 1 にまとめた．なお，HA-PACS には 1 ノード内に 4 枚の GPU が搭載されているが，ここでは MPI 等を使用した並列化は行わず，1CPU core と 1GPU を用いて測定する．並列化については §7 以降で述べる．

表 1 測定環境 (HA-PACS) の概要

CPU	Intel(R) Xeon(R) CPU E5-2670 2.60GHz (8 cores/socket × 2 sockets = 16 cores/node)
GPU	NVIDIA Tesla M2090 (4 GPUs / node)
Main Memory	128 GB, DDR3 1600MHz, 4 channel / socket, 102.8 GB/s/node
OS	CentOS release 6.1 (Final)
CPU Compiler	GCC 4.4.5-6
GPU Toolkit	CUDA 4.0.17
Interconnection	Infiniband QDR × 2 rails
MPI	MVAPICH2 1.7

粒子分布は天文学で広く用いられる Navarro–Frenk–White (NFW) モデル [15] とする．これは球対称モデルで，中心部に近づくにつれ，密度が急激に上昇する分布である．本論文を通してブロックあたりのスレッド数を 256 とし，高速な 64KB の RAM を L1 cache 48 KB : Shared memory 16KB に割り振る，L1 cache prefer オプションを

用いる．ブロック数は $N/(256 \times V)$ とする．また，ツリー法の精度を決定するパラメータ $\theta = 0.6$ に設定する．この値は銀河形成等のシミュレーションで用いられる典型的な値である．

5.2 測定結果

5.2.1 ベクトル化・グループ化の効果

ここでは粒子数 $N = 2^{23}$，粒子は Morton 曲線に従い並び替えをされているとする．$(V, G) = (1, 1) \sim (8, 8)$ の測定を行った．

図 3 には，ベクトル化・グループ化を実装し，(V, G) の組み合わせによってカーネル関数の実行時間がどのように変化するかを調べた結果を示した．予想通り，最高性能が得られる (V, G) の値が存在する．今回の測定では $(V, G) = (4, 4)$ が最速で，N12 の提案手法である $(V, G) = (1, 1)$ に比べておよそ 3.6 倍の高速化が達成できた．

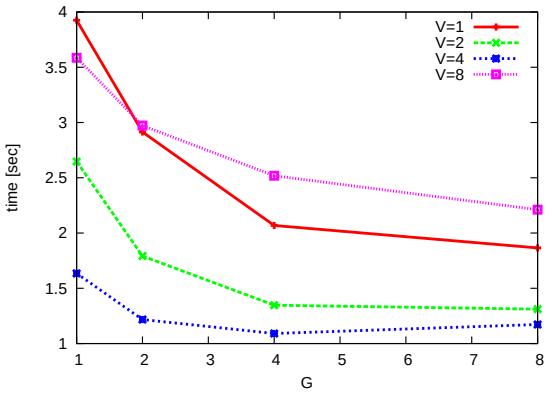

図 3 ベクトル化・グループ化がカーネル関数の実行時間に与える影響．x 軸は G，y 軸は実行時間 (秒)．各線は $V = 1, 2, 4, 8$ を表す

以下では図 3 の結果が得られた理由をいくつかの解析を通して考察する．

まず，各 (V, G) に対する L1 cache ヒット率を調べた結果を図 4 に示す．測定には CUDA Profiler を使用し，L1 cache hit 数と L1 cache miss 数からキャッシュヒット率を算出した．$V \leq 4$ では 90%以上の高いキャッシュヒット率が得られていることがわかる．しかし，$V = 8$ においてキャッシュヒット率は ~75%までに急激に悪化する．本研究の実装では，各スレッドの担当 i-粒子の座標情報等はそれぞれレジスタに記憶している．V を大きくすると，レジスタの圧迫が起こり，足りない容量を低速なローカルメモリに求め，このような振る舞いをする．表 2 に各 $V(G = 1)$ でのレジスタの使用状況をまとめた．Fermi アーキテクチャでは，SM あたりに最大で 48Warp が同時に割り当てられる (Active となる)．Occupancy=1.0 は 48Warp が Active であったことを意味する．これらは CUDA Profiler によって得られた値である．ここで，総使用レジスタ数 (本) は (Warp あたりのスレッド数)×(同時動作した Warp 数)× (1 スレッドあたりの使用レジスタ本数)=32 × (48 × Occupancy)× (1 スレッドあたりの使用レジスタ本数) として算出した．i-粒子 1 つにつき座標など合わせて 11 本のレジスタが必要となる．$V = 1 \rightarrow 2$, $V = 2 \rightarrow 4$ では，スレッドあたりの使用レジスタ数はおおよそそれに沿って増加している．しかし，$V = 4 \rightarrow 8$ においては，1 スレッドあたりの使用レジスタは 2 本しか増加していない．NVIDIA Tesla M2090 に搭載されているレジスタ数は 32768 本であり，$V = 8$ とした場合，レジスタが不足することがわかる．i-粒子データはツリーデータとは異なり，他のスレッドが再利用できないので，キャッシュヒット率の低下をまねく．

表 2 レジスタの使用状況

V	Occupancy	使用レジスタ数/スレッド	総使用レジスタ
1	0.667	27	27648
2	0.5	38	29184
4	0.333	61	31232
8	0.333	63	32256

また，G を大きく設定した方がキャッシュヒット率が向上する．これは，他スレッドにより L1 cache にコピーされたツリーデータを効率よく再利用するためである．

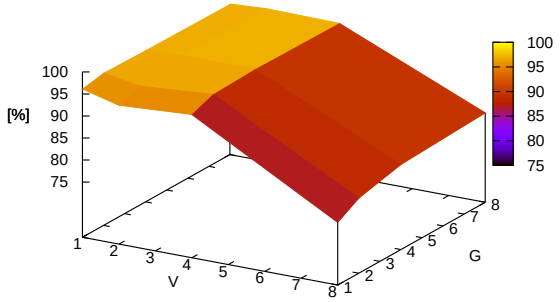

図 4 L1 cache ヒット率．x 軸は V，y 軸は G，z 軸は各 (V, G) に関する L1 cache ヒット率を表す

図 5 には，グローバルメモリと L1 cache へのアクセス回数を L1 cache hit 数と L1 cache miss 数の和として示した．ここで示したのは 1 粒子あたりのメモリアクセス回数である．CUDA Profiler では，Streaming Multiprocessor (SM) あたりの L1 cache ヒット/ミス回数が求められる．M2090 に搭載された SM 数は 16 であるので，1 粒子あたりのメモリアクセス数を (L1 cache ヒット数+ミス数)/SM × 16 / N として算出した．予測された通り，(V, G) が大きな場合にメモリアクセス回数は減少している．

次に，Warp 分岐数の (V, G) 依存性を調べた (図 6)．大きな (V, G) に設定するとより多くの i-粒子のツリー判定を統一するため，Warp 分岐数が減少する．ここでも CUDA

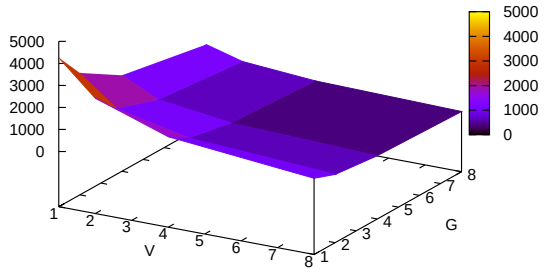

図 5 1 粒子あたりのメモリアクセス回数．x 軸は V，y 軸は G，z 軸は各 (V, G) に関する 1 粒子あたりのメモリアクセス回数を表す

Profiler を測定に用いた．CUDA Profiler では，SM あたりの Warp 分岐数が求められる．1 粒子あたりの Warp 分岐数を (Warp 分岐数/SM) × 16 / N として算出した．

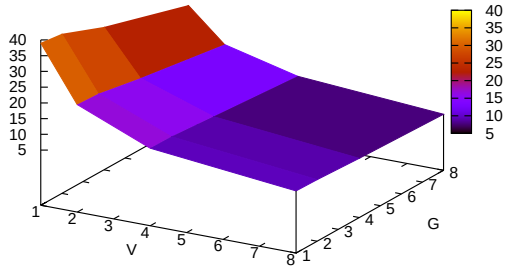

図 6 Warp 分岐数の (V, G) 依存性．x 軸は V，y 軸は G，z 軸は 1 粒子あたりの Warp 分岐数を表す

図 7 は，ある代表 i-粒子が j-セルから受ける重力の計算回数を調べた結果を示す．代表 i-粒子には，系の中心部に位置し，重力計算回数が多いものを選んだ．この粒子の重力計算回数は，$(V, G) = (1, 1)$ で比較すると，外縁部の粒子の 4〜5 倍である．大きな (V, G) に設定するほどより厳しいツリー判定を行い，重力計算回数が増加する．さらに，頻繁に More ポインタを指すので，j-セルとの距離測定等の計算量も増加する．

ベクトル化・グループ化の効果について次のようにまとめられる．図 3 に示したように，ベクトル化・グループ化によりカーネル関数の計算性能を向上させられる．また，最高性能を得るための (V, G) の最適値が存在する．この結果は，以下の複数の要因が複合的に絡んだためである．

(1) L1 cache ヒット率は V を大きくし過ぎると低下する (カーネル関数の計算性能に対して不利)．これは i-粒子データを保持するレジスタが圧迫され，他スレッドが再利用することのできない，それぞれの i-粒子データが低速なローカルメモリへと溢れ出るためである．

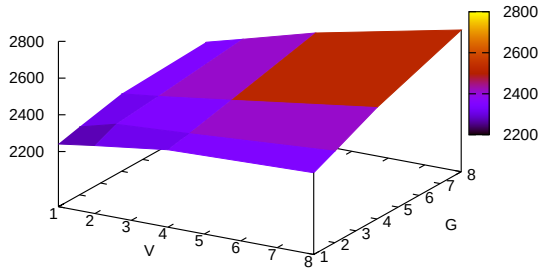

図 7 重力計算回数の (V, G) 依存性．x 軸は V，y 軸は G，z 軸は代表 i-粒子の重力計算回数を表す

(2) Warp 分岐数はより大きな (V, G) に設定すると減少する (有利)．これはより多くの i-粒子のツリー判定を統一するためである．

(3) メモリへのアクセス回数はより大きな (V, G) に設定すると減少する (有利)．これは複数の i-粒子が共通の j-セル対して同時に計算を行うためである．

(4) 大きな (V, G) にした場合，より厳しいツリー判定をするため，計算量が増加する (不利)．

また，現状では (V, G) の最適値を見つけるにはいくつかの組み合わせで試すほかない．カーネル関数内でのツリー走査の仕方を解析し，(V, G) を変化させたときに上に挙げたメリット・デメリットがどのように変化するかをモデル化することで，最適値を解析的に予測することは原理的には可能であると考えている．ただし，それは粒子分布等に強く依存すると考えられ，時々刻々と粒子分布が変動する計算天文学が対象とする問題では実用上困難である．現実的には，計算実行中に一定の間隔で (V, G) の最適値を実験的に探査する方法が有効であろう．

5.2.2 Morton 曲線 vs Peano-Hilbert 曲線

ここでは Morton 曲線，または，PH 曲線に従って粒子の並び替えをして測定を行い，カーネル関数の計算性能が受ける影響を調べた．$N = 2^{17} \sim 2^{23}$，$(V, G) = (4, 4)$ とした．

図 8 に，Morton 曲線を用いた場合に対する PH 曲線を用いた場合のカーネル関数の計算速度向上率を示した．PH 曲線を用いることで，10%程度の速度向上が得られた．ベクトル化・グループ化による高速化と併せて，$3.6 \times 1.1 \sim 4$ 倍の高速化が達成された．ただしこの値は粒子分布により大きく変化しうるものである．今回の測定には球対称モデルを用いたが，実際に計算天文学の分野で扱われるより不均一な粒子分布では，PH 曲線の優位性がさらに大きくなると期待される．

図 9 に示したのは粒子あたりの Warp 分岐数である．PH 曲線の方が Warp 分岐数が小さくなっている．これは Morton 曲線にある「飛び」が PH 曲線には存在しないた

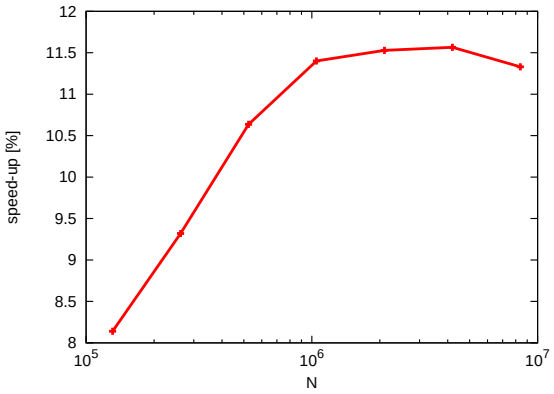

図 8 Morton 曲線を用いた場合に対する PH 曲線を用いた場合の速度向上率．x 軸は粒子数, y 軸は速度向上率を表す

めと考えられる．Warp 分岐数が N と共に増加するのは，走査するツリーが深くなるためである．ここでは粒子あたりの Warp 分岐回数を示したが，N の増加と共に系全体での Warp 分岐回数も増大するため，図 8 のように速度向上率は N に依存する (ただし，$N > 10^6$ で見られる速度向上率の収束の原因は定かでなく，究明には更なる解析を要する)．また，キャッシュヒット率についても調べたが，Morton/PH 曲線でほとんど差はなかった．このように Warp 分岐数がより小さくなるために，PH 曲線を用いることでカーネル関数の計算が加速された．

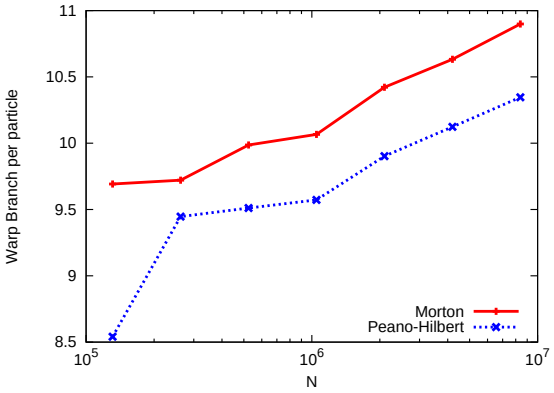

図 9 Morton/PH 曲線で粒子の並び替えをした場合の Warp 分岐数．x 軸は粒子数, y 軸は 1 粒子あたりの Warp 分岐数を表す

この様にカーネル関数を高速化する PH 曲線であるが，そのインデックス計算のコストが Morton 曲線より高いことを先に述べた．インデックス計算・粒子データの並び替えを合わせると，その計算時間はツリー構築と同程度となる．しかし，これらは毎回の時間積分 (重力計算と粒子の位置・速度更新) に必要な工程ではない．これらを毎回行った場合と 100 回の時間積分に 1 度行った場合を比較すると，100 回に 1 度にした場合でもカーネル関数の性能低下は 1%弱にとどまった．これは 100 回程度の時間積分では粒子分布にさほど大きな変化が生じないためである．したがって，インデックス計算のコストが高いことはコード全体の性能を低下させるものではない．

6. ツリー構築の拡張

前節まではカーネル関数に注目し，その計算を高速化することに成功した．これにより，CPU が担当するツリー構築部分の計算時間が支配的となった (§5 の測定では，カーネル関数が最速の $(V, G) = (4, 4)$ の場合にカーネル関数の 2 倍弱の計算時間を要した)．そのため，ツリーコードの更なる高速化には，ツリー構築の計算時間を短縮する必要がある．本研究では，セル内の粒子数が N_{crit} 個以下となった場合にツリー構築を打ち切ることでそれを図る．§2 で述べた基本的な方法は $N_{\mathrm{crit}} = 1$ に対応し，先の測定も $N_{\mathrm{crit}} = 1$ として行った．この N_{crit} を任意の数とする方法はツリー構築の拡張であり，広く用いられている．

この方法を用いた場合，$\log(N_{\mathrm{crit}})$ 段程度のツリー構築をしないため，N_{crit} を大きくするほどツリー構築の計算量は減少する．一方で，ツリー走査中に最下層のセルに達した場合，それに含まれる複数個の粒子が i-粒子に及ぼす重力を，直接法を用いて計算することとなる．つまり，N_{crit} を大きくした場合，ツリー走査の計算量は増大する．これらは計算量のトレードオフ関係にあり，N_{crit} の最適値があると予想される．

図 10 には，N_{crit} を変化させた場合のツリー構築，ツリー走査 (カーネル関数) の実行時間の変化を示した．$N = 2^{25}$, $(V, G) = (4, 4)$ とし，その他の設定は §5 と同一にした．また，粒子は PH 曲線に従って並び替えた．予想通り，N_{crit} を大きくするほど，ツリー構築とカーネル関数の実行時間は減少/増大した．また，総計算時間 (赤線) には CPU–GPU 間の粒子・ツリーデータ通信時間も含まれるが，それらはツリー構築・カーネル関数の実行時間に比べると十分小さい．最速となったのは $N_{\mathrm{crit}} = 4$ で，$N_{\mathrm{crit}} = 1$ に比べて ~12%高速になった．N_{crit} についても最適値は粒子分布等により変化すると考えられる．

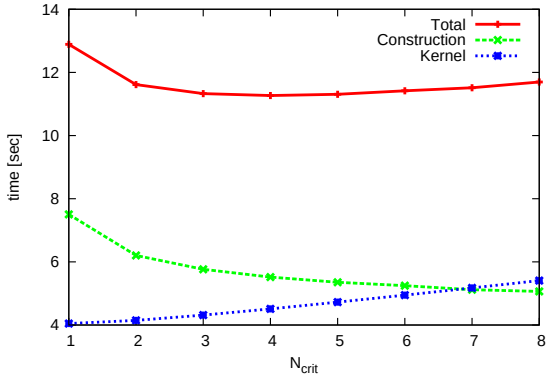

図 10 さまざまな N_{crit} に対するツリー構築 (緑)，カーネル関数 (青) の実行時間．赤線は CPU–GPU 間のデータ通信を含めた総実行時間を表す

7. 並列 GPU 化の実装方針

ここまでは 1CPU+1GPU を用いてコードの計算性能評価をした．本節では現在進めている MPI を利用した並列 GPU 化の方針を述べる．以下では MPI プロセス数 (CPU core 数)=GPU 数であるとする．つまり，各 CPU core をそれぞれ 1GPU のホストとする．HA-PACS はノードあたり 4GPU を搭載しているので，4MPI プロセス/ノードとして使用する．

おおまかな計算の手順は以下の通りである．

(1) 各プロセスに担当する i-粒子を割り当てる．必要ならばプロセス間で粒子データを通信する (領域分割).
(2) 各プロセスが割り当てられた i-粒子からなる部分ツリーを構築する (部分ツリー構築).
(3) プロセス間で必要なツリーデータを通信する (ツリーデータのプロセス間通信).
(4) 得られたツリーデータを用いて，各担当 i-粒子に働く重力を計算し，粒子の位置・速度を更新する．

7.1 領域分割

並列化にあたり「領域分割」が重要になる．領域分割は各 MPI プロセス (GPU) が i-粒子として担当する粒子を決める工程である．空間充填曲線を用いてメモリ空間上での粒子の配置を決める方法は，ツリー構築・ツリー走査を高速化する．領域分割でもそれを利用する．具体的には，空間充填曲線により与えられたインデックスに従って $(N/N_{\rm p})$ 個ずつ i-粒子として各プロセスに割り当てる．ここで $N_{\rm p}$ はプロセス数である．つまり，各 CPU のメインメモリには，$(N/N_{\rm p})$ 個の粒子データが収まれば良い．使用するメモリ容量を小さくする工夫は，大規模問題を解く上で必須である．また粒子データの MPI 通信についても，全粒子データを共有する場合に比べ，通信量を削減できる．

7.2 部分ツリー構築

本研究では，各 CPU がそれぞれ保持する i-粒子データを用いて部分ツリーを構築する方法を採用する．他にも各プロセスにツリー構築を担当する空間領域を割り当てる方法等が考えられるが，この方法ではツリー構築のためにプロセス間通信が必要となる．採用方法の利点として，この通信が不要な点や，並列化に際してツリー構築のアルゴリズムを変更しなくてよい点が挙げられる．一方で，空間充填曲線に従い粒子を等分し，それからなる部分ツリーを作る採用方法では，粒子分布によっては担当プロセスが切り替わる空間領域で部分ツリー同士の幾何学的な重なりが生じる．その領域でツリー構造は分断され，それを用いた重力計算は直接法に近くなる．このように，$N_{\rm p}$ によって全体のツリー構造が変化し，その結果得られる重力が変化するが，実用上問題はない [16].

7.3 ツリーデータのプロセス間通信

各プロセスはそれぞれが構築した部分ツリーデータしか持たず，系全体から働く重力を計算するには他プロセスとの通信が必要である．最も簡単な方法は，全プロセスが全部分ツリーデータを共有する方法である．しかし，ツリーのデータ容量は粒子データと同程度になるため，この方法では大規模問題を解くことができない．また，ツリーデータの MPI 通信時間が無視できなくなることが予想される．

そこで本研究では，以下のようにツリーデータの通信量そのものを削減する工夫をする．これは Warren & Salmon 1993[17] で提唱された方法で, Locally Essential Tree (LET) と呼ばれる．ここで，LET を供給される (i-粒子を保持する) プロセスを I プロセス，LET を供給する (部分ツリーを保持する) プロセスを J プロセスとする．全てのプロセスが $(N_{\rm p}-1)$ 個の他プロセスの I プロセス，および，J プロセスとなる．J プロセスの作った部分ツリー J の内，I プロセスが保持する i-粒子がツリー走査するのに必要なデータはごく一部だけである．そこで，不要なツリーのデータは間引いて (LET を作成して) 通信する．LET を作成するには，部分ツリー J を 1 つの擬似的な i-粒子にたどらせれば良い．その擬似 i-粒子の座標は，I プロセスの粒子がとりうる座標の中で，各 j-セルに最も近くなるようにする．

ツリーデータのプロセス間通信は以下のように行う．まず各プロセスが保持する粒子の座標の最大・最小値を共有する．次にそれを用いて自プロセス以外の $(N_{\rm p}-1)$ 個の J プロセス向けの LET を作成する．最後にそれらを他プロセスと MPI_Alltoallv を用いて交換する．このように全体通信を使用した場合，1 対 1 通信に比べてプロセス間の同期回数を減らすことができる．

8. 並列 GPU ツリーコードの計算性能評価

以降の測定では，$N_{\rm p}=1$ で最速となる $(V,G)=(4,4)$, $N_{\rm crit}=4$ とする．また，ブロック数は $N/(256\times V\times N_{\rm p})$ とし，粒子は PH 曲線に従って並び替えされているとする．

8.1 強スケーリング

粒子数 $N=2^{25}$ の強スケーリング測定の結果を図 11 に示す．測定は $N_{\rm p}=1\sim 64$ で行った．

$N_{\rm p}\leq 10$ ではツリー構築 (緑) とカーネル関数 (青) が支配的で，それらが良スケールのため，総計算時間 (赤) も良スケールとなる．LET 作成 (紫) は $N_{\rm p}$ によらずほぼ一定値である．LET 作成の際に擬似 i-粒子がたどる部分ツリーの容量は $\propto N_{\rm p}^{-1}$ であるが，それを他プロセス数 $(N_{\rm p}-1)$ 回たどる必要があり，$N_{\rm p}$ への依存性が弱いためと考えられる．$N_{\rm p}>10$ ではこの LET 作成が無視できなくなる．カーネル関数の計算性能も飽和傾向となり，全体のスケールが悪化していく．カーネル関数のこの振る舞いは，プロセス間で担当 i-粒子の計算量に偏りが生じ，全体のロード

バランスが崩れている (系中心部の計算量が多い粒子群を担当するプロセスが全体を律速する) ためと推察している．これに対し，単純に各プロセスに $(N/N_{\rm p})$ 個の粒子を割り当てるのではなく，計算量で重み付けした上で割り当てるという方法が有効であると考えている．LET 作成については，自ら構築した部分ツリーによるカーネル計算とオーバーラップして隠蔽を図れば改善されるであろう．また，HA-PACS には 1 ノードあたり 16 core の CPU が搭載されているが，今回の実装ではそのうち 4 core のみを使用している．残りの CPU core を利用することもさらなる高速化に寄与すると考えている．

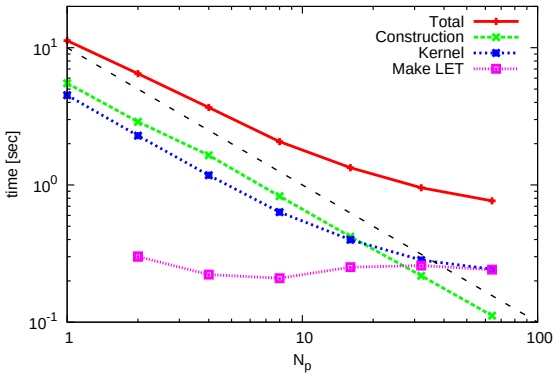

図 11 $N = 2^{25}$ の強スケーリング測定．横軸はプロセス数 ($N_{\rm p}$=CPU core 数=GPU 数)．縦軸は計算時間．各線は総計算時間 (赤)，ツリー構築 (緑)，カーネル関数 (青)，LET 作成 (紫)，理想のスケール則 $\propto N_{\rm p}^{-1}$(黒) を表す

8.2 弱スケーリング

各プロセスの担当 i-粒子数 $N_1 = 2^{24}$ とした弱スケーリング測定を行った (図 12)．このとき総粒子数 $N = N_1 \times N_{\rm p}$ である．測定は $N_{\rm p} = 1 \sim 16$ で行った ($N_{\rm p} = 16$ の時 $N = 2^{28}$)．

総計算時間はおおよそ $\log N_{\rm p}$ に比例する．これはツリー法の計算コストの粒子数依存性から理解できる．ツリー法の計算コストは $O[N \log(N)] = O[(N_1 \times N_{\rm p}) \log(N_1 \times N_{\rm p})]$ である．これを $N_{\rm p}$ プロセスで並列処理するとし，定数部分を除くと，$\log N_{\rm p}$ の依存性が残る．つまり，ツリーの階層数がこのスケーリングを決めている．

また，$N > 10^8 (N_{\rm p} \geq 8)$ の問題について 10 秒程度で一回重力計算ができる．この規模の計算を実行するに十分な計算性能が達成されたと言える．HA-PACS の豊富な計算資源をもってすれば，複数のパラメータ計算も可能であろう．

9. 関連研究

Bedorf et al.[12] は，本研究と同様に GPU を用いてツリー法の計算加速を行った．この研究の実装方針が本研究

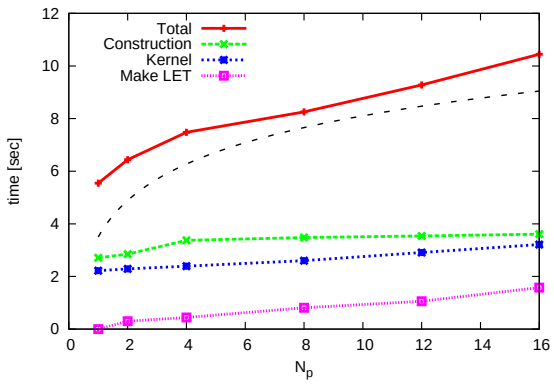

図 12 $N_1 = 2^{24}$ の弱スケーリング測定．横軸はプロセス数 ($N_{\rm p}$=CPU core 数=GPU 数)．縦軸は計算時間．各線は総計算時間 (赤)，ツリー構築 (緑)，カーネル関数 (青)，LET 作成 (紫)，予想スケール則 $\propto \log N_{\rm p}$(黒) を表す

のものと異なるのは，ツリー走査だけでなく，ツリー構築部分も GPU が担当する点である．ツリー走査においては，Bedorf et al. でもベクトル化と同様に 1GPU コアが複数の i–粒子を担当しているが，その上限値が設定されているのみで，実際の担当 i–粒子数はコアによって異なる．コア間のロードバランスの面から，担当 i–粒子数は同程度の数が望ましい．また，複数のコアを束ねるグループ化のような操作は行われていない．Warp 分岐数やメモリアクセス回数の観点から，グループ化はこの研究の実装においても有用であると考えられる．

また，Bedorf et al. では並列化については述べられていない．その方針を採用した場合，単 GPU のみを用いるのであれば，計算に必要な粒子・ツリーデータは常に GPU に存在しており，CPU–GPU 間の通信頻度を下げられるため，通信時間の削減が期待できる．しかし並列化を行う場合には，粒子・ツリーデータは一般に CPU を一旦介して他 GPU へと送受信されるため，その恩恵を受けにくくなる．ただし，CUDA 4.0[7] 以降で使用可能な GPUDirect v2.0 や，筑波大学で開発が進められている Tightly Coupled Accelerators (TCA)[18] など，GPU 間で直接データ通信をする技術の利用により，今後この問題は緩和・解決されるかもしれない．

加えて，高速多重極展開法 (Fast Multipole Method：FMM) を GPU クラスタ向けに実装した Yokota et al.[13] も重要な先行研究の一つとして挙げる．FMM は $O[N]$ の計算コストで N 粒子間の相互作用を計算可能なアルゴリズムである．論文中では 1~4096MPI プロセス (=GPU 数) の弱スケールについて調べられており，4096 プロセスにおいて 74%の並列化効率が得られている．このときプロセス (GPU) あたりの粒子数は 2^{24} である．Yokota et al. では重力相互作用について計算されていないため，本研究との比較はおおまかにしかできないが，$N = 2^{24}$ の場合 (1 プロセス)，計算時間は ~80 秒となっている．一方で我々の

実装では，同じ粒子数では ∼5 秒で一度重力計算が可能である (図 12)．§8 で示したように，弱スケール測定においてツリー法の計算時間は $\log N_{\rm p}$ に比例する．一方で FMM は $O[N]$ のアルゴリズムであるため，理想的には計算時間は変化しない．このようにスケール則に関して不利な面はあるものの，$N_{\rm p} = 1$ で我々の実装の方が 10 倍以上高速であるため，ある並列数までは我々の実装が優位に立つであろう．

10. まとめと今後の予定・展望

本研究では GPU を使用し，計算天文学で広く用いられる重力計算法，ツリー法を高速化した．先行研究である Nakasato (2012) の提案手法を発展させることで更なる計算加速を目指した．本研究で特に留意したのは，L1 cache ヒット率を高くすることと，Warp 分岐数・メモリアクセス回数を削減することである．そのために，重力を計算される複数粒子のツリー判定をスレッド内でまとめるベクトル化と，複数のスレッド単位でまとめるグループ化を実装した．これらにより，およそ 3.6 倍のカーネル関数の高速化に成功した．また，Nakasato (2012) で採用された Morton 曲線より性質の良い Peano-Hilbert 曲線を利用し，カーネル関数の計算をさらに 1 割程度加速した．これらを併せ，4 倍程度の高速化に成功したこととなる．

MPI を用いた並列 GPU 化も進めている．並列化の基本方針は，領域分割や LET によって通信量を必要最低限に抑えるなど，CPU の並列化でも用いられる一般的なものである．$N_{\rm p} = 16$ を用いたとき，$N = 2^{28} > 10^8$ の重力計算を 1 回あたり 10 秒程度で行うことが可能 (図 12) であり，この規模の計算が実行可能な計算性能が達成された．しかし，自らの構築した部分ツリーを用いたカーネル計算と LET 作成のオーバーラップ等，改善すべき点がいくつか残されている．今後はこれらの問題を一つずつ解決し，$N > 10^9$ の大規模計算に挑戦する．

謝辞 本研究を進めるにあたり，有益なご意見を多数いただきました，筑波大学大学院システム情報工学研究科の児玉 祐悦教授，高橋 大介教授，計算科学研究センターの塙 敏博准教授，ならびに，HA-PACS を利用する機会を頂いた筑波大学計算科学研究センターに深く感謝致します．本研究の一部は筑波大学計算科学研究センター学際共同利用プログラム採択課題「近傍銀河の進化の探究」によるものです．

参考文献

[1] M., Mori, & R. M., Rich, The Once and Future Andromeda Stream, Astrophysical Journal, 674, L77 (2008).
[2] G., Ogiya, & M., Mori, The Core-Cusp Problem in Cold Dark Matter Halos and Supernova Feedback: Effects of Mass Loss, Astrophysical Journal, 736, L2 (2011).
[3] G., Ogiya, & M., Mori, The Core-Cusp Problem in Cold Dark Matter Halos and Supernova Feedback: Effects of Oscillation, arXiv:1206.5412 (2012).
[4] J. Barnes, P. Hut, A hierarchical $O[N \log(N)]$ force-calculation algorithm, Nature 324 446449 (1986).
[5] L. Nyland, M. Harris, J. Prins, Fast N-Body Simulation with CUDA (2007).
[6] Y., Miki, D., Takahashi, & M., Mori, A Fast Implementation and Performance Analysis of Collisionless N-body Code Based on GPGPU, Procedia Computer Science, 9, 96 (2012).
[7] Nvidia, NVIDIA CUDA C Programming Guide Version 4.0 (2011).
[8] Khronos, The OpenCL Specification Version 1.2 (2011).
[9] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence, in SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. New York, NY, USA: ACM, pp. 1-12 (2009).
[10] N., Nakasato, Implementation of a Parallel Tree Method on a GPU, Journal of Computational Science, Volume 3, Issue 3, Pages 132141 (2012).
[11] N., Nakasato, G., Ogiya, Y., Miki, M., Mori, & K., Nomoto, Astrophysical Particle Simulations on Heterogeneous CPU-GPU Systems, arXiv:1206.1199 (2012).
[12] J., Bédorf, E., Gaburov, & S., Portegies Zwart, A sparse octree gravitational N-body code that runs entirely on the GPU processor, Journal of Computational Physics, 231, 2825 (2012).
[13] R., Yokota, L. A., Barba, T., Narumi, & K., Yasuoka, Petascale turbulence simulation using a highly parallel fast multipole method on GPUs, arXiv:1106.5273 (2011).
[14] W. M., Lam, & J. M., Shapiro, A Class of Fast Algorithms for the Peano-Hilbert Space-Filling Curve, Proceedings ICIP 94, 1, 638-641 (1994).
[15] J. F., Navarro, C. S., Frenk, & S. D. M., White, A Universal Density Profile from Hierarchical Clustering, Astrophysical Journal, 490, 493 (1997).
[16] J., Makino, A Fast Parallel Treecode with GRAPE, Publications of the Astronomical Society of Japan, 56, 521 (2004).
[17] M., Warren, & J., Salmon, A parallel hashed oct-tree N-body algorithm, In Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (pp. 1221). New York: ACM. (1993).
[18] 塙 敏博, 児玉祐悦, 朴 泰祐, 佐藤三久：Tightly Coupled Accelerators アーキテクチャのための通信機構，情報処理学会研究報告 (アーキテクチャ)， Vol. 2012-ARC-201, No. 26, pp. 1–8 (2012).