Vol.2012-HPC-135 No.40
2012/8/3

重力多体系用Tree Codeの並列GPU化

扇谷　豪[1,2,a)]　三木　洋平[1,2]　朴　泰祐[3]　森　正夫[4]　中里　直人[5]

概要：“ツリー法”は非常に高効率な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ことから，その高速化が精力的に行われている．本研究では，ツリー法の GPU(Graphics Processing Unit) 化を行い，さらにそれを並列化することで，それを高速化した．特にメモリ空間上でのデータの配置に工夫した．また，Warp 内での分岐回数を減少させ，大幅な高速化を可能にする手法も提案する．これらを実装することで，先行研究の提案手法を用いた場合に比べて 200%を越えるカーネル関数の高速化に成功した．

キーワード：GPU，N 体シミュレーション，ツリー法

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]．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，それぞれの粒子に働く重力を計算し，その軌道を追う．例外もあるが，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，用いる粒子数を N とすると，M/N の質量分の星やダークマターに対応させられる．

より高い空間分解能を得るため，離散化に起因する人工的な計算結果の劣化を防ぐため，より大きな N を用いた計算が求められている．しかし，当然ながら計算コストも N に依存して増大する．最も素直で正確な重力計算法は直接法である．これは，N 個の粒子それぞれについて，それ以外の (N-1) 個の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，大粒子数の計算には向かない．そこで，計算コストを $O[N \log(N)]$ に減少させることのできる，高効率な重力計算法 “ツリー法”が計算天文学の分野では広く用いられている [3].

高速な演算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) の黎明期から現在まで盛んに行われ，特に直接法において大きな成果が上げられている [4] [5]．また，近年 NVIDIA 社の提供する CUDA[6] や，クロノス・グループによって策定された OpenCL[7] など，開発環境の整備が進み，ツリー法等のより複雑なアプリケーションについても高速化に成功している [8][9][10][11].

本研究では，Nakasato 2012[9] で提案された手法を元に，それを発展させることで更なる高速化を目指す．MPI を用いた複数 GPU の並列計算のアルゴリズムについても提案し，現状を報告する．また，本研究の開発環境は CUDA で，NVIDIA 社製の GPU を用いて議論を進めるが，多くの内容は他の環境においても応用が可能である．本稿の構成は以下の通りである．まず，§2 でツリー法の概要を説明し、§3 で Nakasato 2012 の手法を紹介する．§4 では本研究の提案手法について説明し，§5 で 1GPU を用いた性能評価を行う．§6 では MPI による並列化アルゴリズムについて提案し，実装と性能の現状も示す．最後に §7 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．

[1] 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
[2] 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
[3] 筑波大学システム情報系
Faculty of Engineering, Information and Systems, University of Tsukuba
[4] 筑波大学数理物質系
Faculty of Pure and Applied Sciences, University of Tsukuba
[5] 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

2.1 ツリー構築 (Tree Construction)

ツリー法の名の通り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していく．基本的な方法では，セル内の粒子が 1 個以下になるまでこれを続ける (図 1)．次に，セルと粒子の群を正しく結びつける．ここで分割前のセルを親セル，自らを分割してできたものを子セルとする．つまり，図 1 の②から見ると，①が親セル，③，④，⑤が子セルとなる．また，ツリーの末端ではセル=粒子となる．各セルの座標はそれに含まれる粒子群の重心とする．

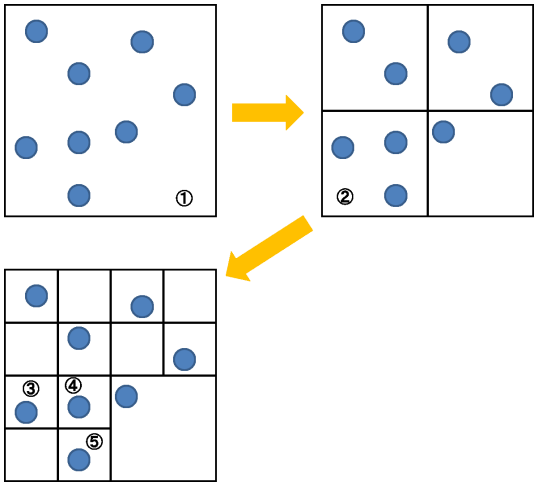

図 1 再帰的なセルの分割.

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) とポインタで結びつけることで粒子のツリー構造が完成する (図 2)．ここで，セルが末っ子の場合には Next ポインタを親セルの兄弟セルにつなげる．

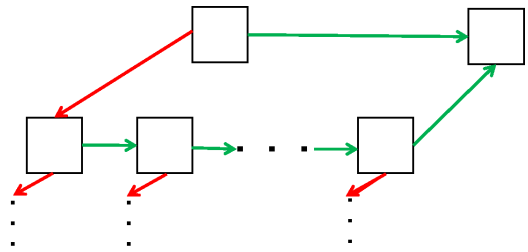

図 2 セル同士の結合．赤は More ポインタ，緑は Next ポインタを表す．また，矢印は始点となるセルが終点のセルを指すポインタであることを示す.

2.2 ツリー走査 (Tree Traversal)

そろぞれの重力計算をされる粒子 (i-粒子) について，根セルからポインタを用いてツリー全体を渡り歩く．ツリーを歩く際，その都度たどり着いたセル (j-セル) と i-粒子の距離を測定し，j-セルが i-粒子から見て十分に “遠いか否かの判定”をする．この時，近ければ 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 ポインタの指す先に進む．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，孫セル等に相当する複数の粒子から 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．ツリーの階層はおよそ $\log(N)$ 段であり，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．

ツリーの判定には様々なやり方があるが，本研究では図 3 に示すような，パラメータ θ を用いた一般的な方法を採用する．この方法では，i-粒子と各セルの重心の間の距離 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，

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

を満たした場合に “遠い”と判定する．この式から，パラメータ θ はセルの見込み角に対応し，θ を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，直接法からのずれは小さくなる) ことがわかる．また，θ は $0 \leq \theta \leq 1$ 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させ，記憶しておく．

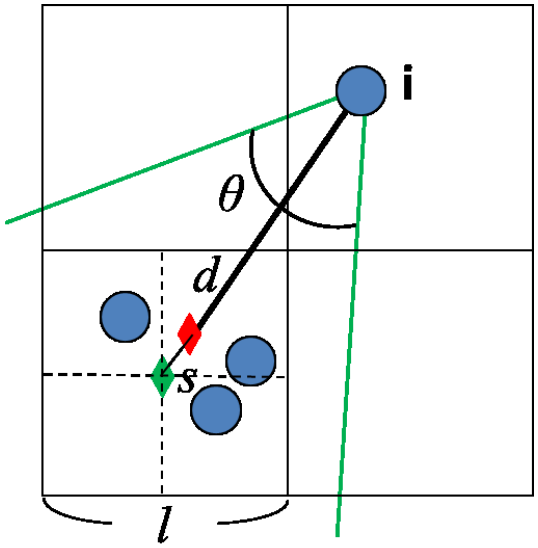

図 3 パラメータ θ を用いた判定の仕方の概念図．赤点，緑点はそれぞれセルの重心，中心を表す.

3. Nakasato 2012 の概説

ここでは Nakasato 2012(N12) によって提案された, GPU を用いたツリー法の高速化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 が担当している．後者の計算量は前者に比べ，圧倒的に大きいことが知られている．また，CPU-GPU 間の通信等は並列化の際の

工夫によって GPU での計算 (カーネル関数) とオーバラップすることも可能である (§6).

GPU のメモリは，いくつかの階層構造を持っており，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，非常に高速な 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ということである．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プリケーションであれば，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，ツリー法の場合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 では切り替えが可能) ので，高い L1 cache ヒット率が達成できれば，Shared memory を有効活用したのと同程度の性能が得られると期待される．

N12 で提案されたのは，各スレッドがそれぞれ 1 つの i-粒子を担当し，ツリー走査を行う方法である．L1 cache のヒット率を上げるには，同一ブロック内のスレッドに同じよう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，ツリー上で同じような判定を繰り返し，同じような経路をたどると期待される (同様の空間座標を持つ)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，Morton 曲線という (Z を繰り返し描くような曲線のため，Z–curve とも呼ばれる) 空間充填曲線 (図 4) を利用し，各粒子に 1 次元的に番号を与え，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ってこれを達成している．実際に N12 では，これによってカーネル関数を約 2 倍高速化することに成功している．また，CPU が担当するツリー構築に関しても，同様に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．

4. 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，その性能を著しく低下させる原因の 1 つとして，Warp の分裂 (Warp divergent) が挙げられる．現在の GPU は 32 スレッドが Warp という単位でまとまり，同時に動作する仕様になっている．Warp 内の 32 スレッドが条件分岐により，別々の処理を行う A 個のフローに分裂した場合には，Warp は A 種類のフローを順に処理する．つまり全ての処理が済み，フローが合体するまでの Warp の実行時間は，A 種類の命令実行時間の和となる．そのため，Warp 分裂の発生中には，有効に仕事ができるスレッド数が減少し，大きな性能低下をまねく．しかがって，性能の向上にはできる限り Warp 分裂を防ぐ必要がある．

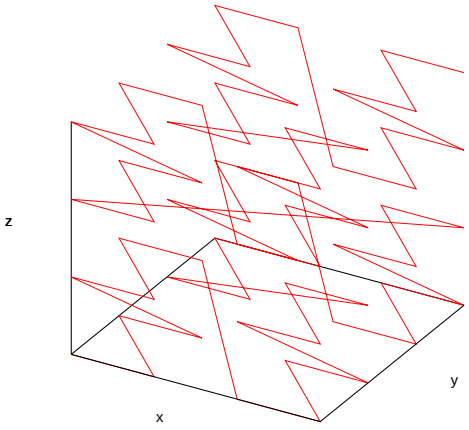

図 4 Morton 曲線.

ツリー走査での Warp 分裂は，“More ポインタと Next ポインタのどちらをたどるか”で主に発生する．次にたどるのが 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に移動するだけで良い．一方，Next ポインタの場合には，各スレッドは現在のセルからそれぞれが担当する i-粒子に働く重力を計算した後に，セルの移動をする．この Warp 分裂が頻繁に起こると，More ポインタを指すスレッドは不要でかつ大きな計算量の重力計算を何度も行うことになり，性能が著しく低下すると考えられる．

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 分裂を減らすことでカーネル関数の高速化を行う．

4.1.1 ベクトル化

N12 では 1 スレッドが担当する i-粒子数を 1 としていた．ここでは，1 スレッドあたりの i-粒子数を V 個とし，V 個の i-粒子も間でツリーの判定を統一する．具体的には次のようにする．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-粒子について計算する．次に，ツリー判定をその中で最も厳しい判定となるよう，つまり，V 個の i-粒子の中で最も j-セルに近い距離を判定に用いる．本研究ではこの方法をベクトル化と呼ぶ．

V を大きくとるほどに，全粒子の重力を計算するまでの総 Warp 分裂数は減少すると期待される．一方で，V を大きくすることはより厳しい判定をすることになるので，1 スレッドあたりの計算量は V に依存して増加する．したがって，最も高性能が得られる V の値が存在すると考えられる．

4.1.2 グループ化

Warp 分裂数を減少させるには，できるだけ多くの i-粒子のツリー判定を統一したい．しかし，ベクトル化のみで

は先に述べた理由により，それほど大きな数の V をとることはできないと考えられる．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-粒子のツリー判定も統一することで，さらに Warp 分裂を減少させる．本研究ではこの方法をグループ化と呼ぶ．もちろんベクトル化・グループ化は単体でも使用可能であるが，以下ではこれらを組み合わせることを前提として議論を進める．

グループ化は次のように行う．ここで注意したいのが，グループ化でまとめられる G 個のスレッドは同時に同じ j-セルに対して判定を行うということである．まず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距離の値を求め，それを Shared memory にコピーする．最後に Shared memory を参照して G 個のグループの中での最小の距離の値 $R_{\min}$ を共有し，その値を使ってツリー判定をする．

ベクトル化の時と同様に，G を大きくとるほどに，ブロック内での Warp 分裂数は減少すると期待される．しかし，Shared memory の参照から $R_{\min}$ を求める際の作業量は 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適切な値が存在すると考えられる．

ここまでの議論を簡単にまとめると，ベクトル化・グループ化を組み合わせた場合，より厳しい判定をするため，計算量は増加する．一方で，$V \times G$ 個の i-粒子のツリー判定を統一するため，Warp 分裂による性能の低下を妨げることが可能になる．このようなトレード・オフがあるため，高性能を得るための V, G のペアが存在すると考えられる．

4.2 Morton 曲線 vs Peano-Hilbert 曲線

N12 で行われた，Morton 曲線に従って似通った空間座標を持つ粒子をメモリ空間上でも近く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，似通ったツリーのたどり方をする粒子を Warp 内に集め，Warp 分裂の回数を減らすという観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．Morton 曲線と比較して PH 曲線の持つ利点は，図 4，5 を見比べてわかるように，Morton 曲線にある空間的な “飛び”が PH 曲線にはない点である．Morton 曲線は Z を繰り返し書くように描かれるので，必ず対角線上へのジャンプが起こる．つまり，このときメモリ空間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っていくので，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．

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 分裂の減少が期待される．その恩恵は特にベクトル化・グループ化を使用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 “飛び”をまた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，計算量を増やすこととなる．PH 曲線ならばそのような “飛び”がないため，より効率的にツリーを渡り歩くことができる．また，同様の理由により，実際に計算天文学で計算される不均一な粒子分布の問題には PH 曲線の方が向いていると言える．

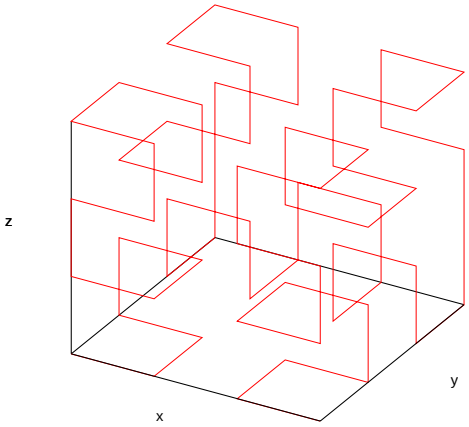

図 5 Peano-Hilbert 曲線.

5. 性能評価

ここでは，先に提案したベクトル化・グループ化，PH 曲線がカーネル関数の性能にどのような影響を与えるのかを，これらを実装して調べる．

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 は GPU クラスタであり，また，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，CPU 1core + 1GPU を用いて測定する．並列化については §6 で述べる．

粒子分布は天文学で広く用いられる Navarro–Frenk–White(NFW) モデル [12] 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．本稿を通してブロックあたりのスレッド数は 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．

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

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 $N = 2^{20} = 1048576$，粒子は Morton 曲線によって並び替えをしているとする．$(V, G) = (1, 1) \sim (8, 8)$ の測定を行った．

図 6 には，§4 で述べたベクトル化・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性能がどのように変化するかを調べた結果を示した．予想された通り，最高性能が得られる (V, G) の値が存在する．また，速度向上率は山型の分布となった．特に $V = 4$ に注目した結果を図 7 に示した．今回の測定では $(V, G) = (4, 4)$ が最速で，$(V, G) = (1, 1)$ に比べ 248%の高速化に成功した．

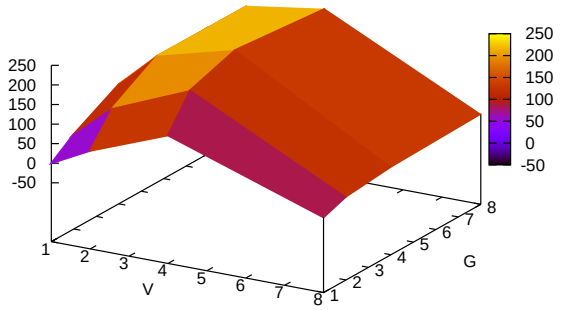

図 6 ベクトル化・グループ化がカーネル関数の性能に与える影響．x 軸は V，y 軸は G，z 軸は $(V, G) = (1, 1)$ に対する速度向上率を表す．

以下では図 6 の結果が得られた理由をいくつかの解析を通して理解する．

まず初めに，各 (V, G) に関する L1 cache ヒット率を調べた (図 8)．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．この図から，$V \leq 4$ では 90%を越える非常に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 ~70%程度に急激に悪化している．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させている．V

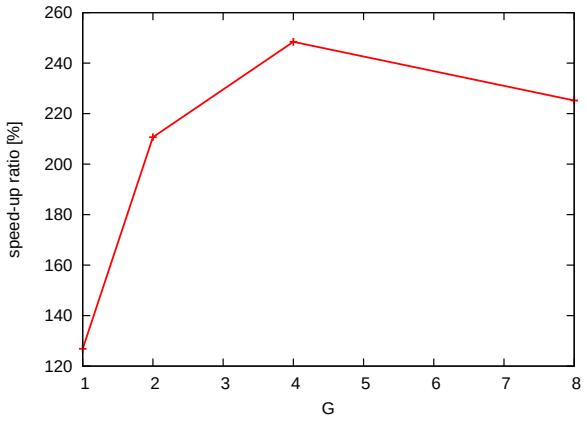

図 7 $V = 4$ に注目した速度向上率．x 軸は G, y 軸は $(V, G) = (1, 1)$ に対する速度向上率を表す．

を大きくとると，レジスタの圧迫が起こり，足りない容量をグローバルメモリ (大容量・低速) に求め，このような振る舞いをする．表 2 に各 V 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，最大で 48Warp を同時に動作できる．Occupancy=1.0 は 48Warp が同時動作したことに対応する．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) として計算した．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，おおよそそれに沿った増加の仕方をしている．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，必要な本数をまかなえず，レジスタが溢れていることがわかる．各スレッドの担当する 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を下げる要因となる．

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高くなる．これは，より多くのスレッドが L1 cache を効率よく使い回したためである．

次に，Warp 分裂数の (V, G) 依存性を調べた (図 9)．より多くの i-粒子のツリー判定をまとめるため，大きな (V, G) に設定するほど Warp 分裂数は減少する．ここでも CUDA Profiler を測定に用いた．CUDA Profiler では，Streaming

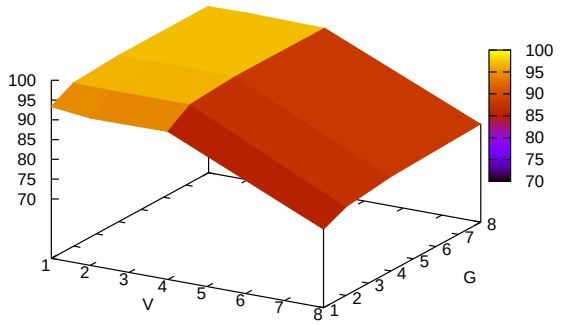

図 8 L1 cache ヒット率．x 軸は V，y 軸は G，z 軸は各 (V, G) に関する L1 cache ヒット率を表す．

Multiprocessor(SM) あたりの Warp 分裂数が求められる．また，M2090 に搭載された SM 数 (16) であるので，1 粒子あたりの Warp 分裂数を (Warp 分裂数/SM) $\times$ 16 / N として求めた．

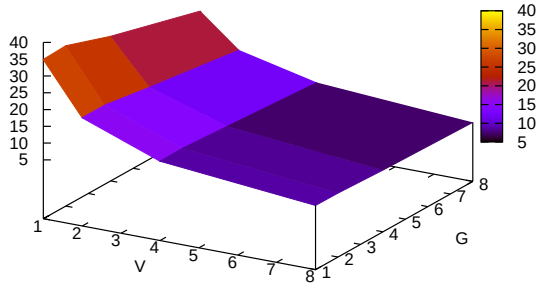

図 9 Warp 分裂数の (V, G) 依存性．x 軸は V，y 軸は G，z 軸は 1 粒子あたりの Warp 分裂数を表す．

図 10 には，ある代表 i-粒子への j-セルから受ける重力の計算がなされた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，重力計算が頻繁に行われる粒子を選んだ．この粒子の重力計算回数は，$(V, G) = (1, 1)$ で比較すると，外縁部の粒子のおよそ 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．これに加え，より深くツリーに潜る (頻繁に More ポインタを指す) ことにもなるので，j-セルとの距離測定など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は次のようにまとめられる．図 6 に示したように，ベクトル化・グループ化によってカーネル関数の性能を飛躍的に向上させることができる．最高性能を得るための (V, G) の値が存在し，速度向上率はそれを頂点とする山型の分布となった．このような速度向上率分布となったのは，以下のようないくつかの要素が絡んだためと考えられる．

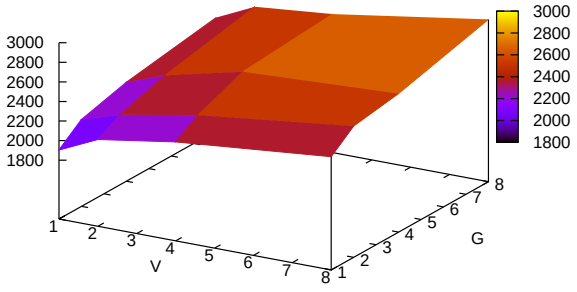

図 10 重力計算回数の (V, G) 依存性．x 軸は V，y 軸は G，z 軸は代表 i-粒子の重力計算回数を表す．

(1) L1 cache ヒット率は V を大きくし過ぎると大きく低下する (カーネル関数の性能にとって不利)．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-粒子データがグローバルメモリへと溢れ出るためである．
(2) Warp 分裂数はより大きな (V, G) に設定すると大きく減少する (有利)．これはより多くの i-粒子のツリー判定をまとめるためである．
(3) 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，現状では最適な (V, G) のペア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詳しく解析し，(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予測できるのではないかと考えている．

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，カーネル関数の性能がどのような影響を受けるかを調べた．また，$N = 2^{17} \sim 2^{23}$，$(V, G) = (4, 4)$ とした．

図 11 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速度向上率を示した．PH 曲線を用いることで，数%から 10%程度の速度向上が得られた．

以下では図 11 のように，Morton 曲線に比べ速度向上が得られた理由を探る．

図 12 には L1 cache ヒット率を示した．キャッシュヒット率は Morton，PH どちらでも大差はない．また，図 13 に示したのは Warp 分裂数である．PH 曲線の方が Warp 分裂数が小さくなっていることがわかる．これは Morton 曲線にある “飛び” が PH 曲線には存在しないためと考えられる．このように Warp 分裂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高速化がされた．

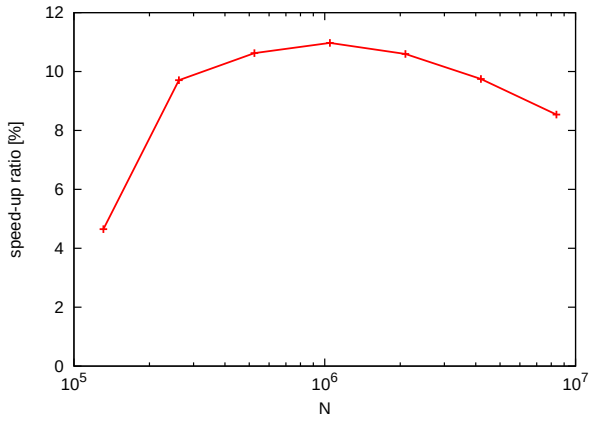

図 11 Morton 曲線を用いた場合に対する PH 曲線を用いた場合の速度向上率．x 軸は粒子数 N,y 軸は速度向上率を表す．

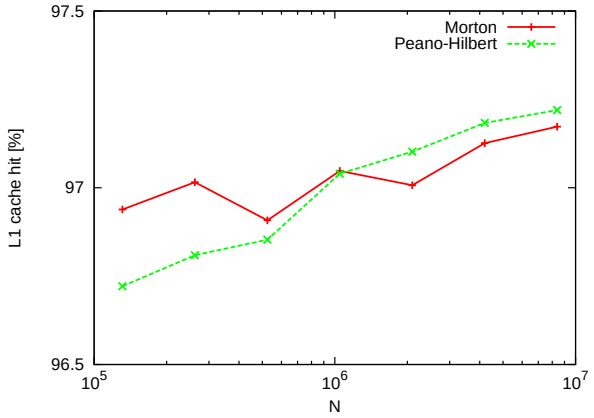

図 12 Morton/PH 曲線で粒子の並び替えをした場合のキャッシュヒット率．x 軸は粒子数 N,y 軸は L1 cache ヒット率を表す．

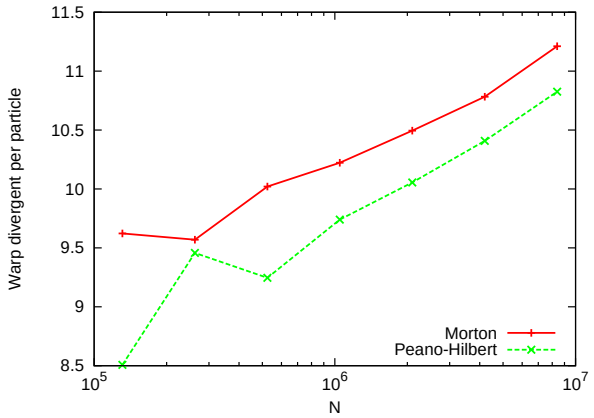

図 13 Morton/PH 曲線で粒子の並び替えをした場合の Warp 分裂数．x 軸は粒子数 N,y 軸は 1 粒子あたりの Warp 分裂数を表す．

6. 並列 GPU 化

§5 では 1GPU のみを用いて，ベクトル化・グループ化，PH 曲線の効果について評価した．ここでは，現在進めている MPI を利用した並列 GPU 化の方針を述べ，現在の実装状況を報告する．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 はそれぞれ 1GPU のホストとする．

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-粒子を割り振る．必要があればプロセス間で粒子データの交換をする (領域分割)．
(2) 各プロセスに構築すべきツリーの占める空間領域を割り振る．必要があればプロセス間で粒子データをやり取りし，各 CPU が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やり取りし，カーネル関数で重力計算をする (並列重力計算)．
(4) 得られた重力加速度を用いて，粒子の位置・速度を更新する．
(5) (1)~(4) を必要なだけ繰り返す．

6.1 領域分割

並列化にあたり，“領域分割”が非常に重要になる．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．先に示したように，Morton/PH 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・ツリー走査を大きく高速化する．このことから，領域分割もそれを利用する方式をとる．その方法は非常に簡単で，Morton/PH 曲線によって与えられたインデックスにしたがって (N/p) 個ずつ i-粒子として各プロセスに割り当てれば良い．ここで p はプロセス数である．つまり，各プロセスのメモリには，$(N/p+\alpha)$ 個の粒子データが収まれば良い (α については後述)．このようなメモリの節約は大規模な問題を解く場合には必須になる．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，データの通信量は大きく削減される．

6.2 部分ツリー構築

各 CPU はそれぞれ部分ツリーを構築する．そのために，各々のプロセスに部分ツリーの占める空間領域を割り当てる必要がある．その最も単純な方法は，全体を等分しそれぞれが保持している i-粒子の位置する領域を割り当てるやり方である (粒子の分布の仕方によってはロードバランスを崩す可能性もある)．しかし，先に述べたように領域分割を行った場合，部分ツリーを作るのに必要な粒子データが揃っている保証はない．正しく部分ツリーを構築するには，それが占める領域は立方体，あるいは，直方体として割り振る必要がある．しかし，Morton/PH 曲線を利用した領域分割をした場合，他の担当領域にはみ出す粒子が発生しうる．逆に言えば，部分ツリーを構築するのに必要なデータが揃っていない可能性がある．そのため，データの不足がなくなるように他プロセスと粒子データをやり取りする必要がある (先に示唆した $+\alpha$)．必要なデータが揃えば，各プロセスはそれから部分ツリーを構築することがで

きる．このようにすることで，ツリー構築の並列化が可能になる．

6.3 並列重力計算

各プロセスはそれぞれが構築した部分ツリーしか持たないので，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須となる．最も簡単な方法は，全プロセスが全部分ツリーを共有することである．しかし，ツリーのデータ容量は粒子と同程度になるため，この方法では大規模な問題を解くことができない．また，p を大きくした際には，その通信時間はほぼ横ばいのままで，ツリーデータの MPI 通信時間が無視できないものとなることが予想される．

そこで本研究では，以下のような方法をとる．

(1) 各プロセスは自前の部分ツリー (ツリー 0 とする) 以外に，2 つの部分ツリーを保持できるメモリを確保する (それぞれツリー 1，2 とする)．
(2) 他プロセスのツリー 0 から必要なデータをツリー 1 に取得する．同時にツリー 0 のデータを他プロセスに転送する．
(3) ツリー 1 のデータを使ってカーネル計算をする間に，ツリー 2 に他プロセスから必要なツリーデータを取得し，ツリー 0 からはデータを他プロセスに転送する．
(4) ツリー 1,2 を順番に使い，全プロセスと通信するまで (3) を繰り返す．

この方法では，確保するメモリは全ツリーデータを保持するために必要な容量の $3/p$ 倍以下ですみ，ツリーデータの通信時間の隠蔽を図ることができる．

また，ツリーデータの通信量そのものを大幅に減少させる工夫もする．これは Warren & Salmon 1993[13] で提唱された方法で，Locally Essential Tree(LET) と呼ばれる．他プロセスの作った部分ツリーの内，実際にツリー走査に必要なデータはほんの一部だけである．そこで，不要なツリーのデータは間引いて (LET を構築して) 通信するのである．LET を構築するには，ツリー 0 の部分ツリーを 1 つの擬似的な i-粒子に歩かせれば良い．その擬似 i-粒子の座標は，次に LET を渡す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．LET の構築は上の (2)，(3) において，ツリー 0 から転送する “必要なデータ” を用意することである．

6.4 並列化の進捗状況

先に述べた並列化計画の内，領域分割は未実装であり，全プロセスが全粒子データを共有している状態なので，粒子の通信なしで部分ツリーの構築をしている．また，並列重力計算の部分は実装が済んでいる (ただし，通信の隠蔽は行っていない)．粒子数 $N = 2^{24} = 16777216$ の測定結果を図 14 に示す．粒子の分布は立方体内に一様とした．また，粒子は Morton 曲線に従って並び替えをしている．ブロック数は $N/(256 \times V \times p)$ とする．

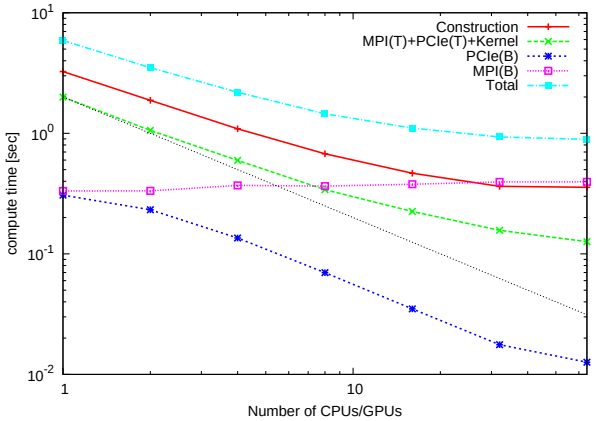

図 14 開発中のコードの並列性能．横軸はプロセス数 (p = CPU core 数 = GPU 数)．縦軸は計算時間．各線が計算時間の内訳を示す．T，B はそれぞれツリー，粒子データの通信であることを表す．また，黒線は $\propto p^{-1}$ の理想スケールを表す．

領域分割が未実装であるため，現在は全プロセスが全粒子データを共有するために，$64 \times N$ という大きなデータ通信が生じている．また，1 プロセスあたりの通信量は並列数を上げても変化しないため，その影響が p を大きくしたときに顕著になる (紫線)．メモリ面の要請だけでなく，計算時間の面からも領域分割は必須であると言える．

ツリー構築 (赤線) とカーネル関数+ツリーデータの通信 (緑) については，p が小さな場合には，$\propto p^{-1}$ のスケールを表す黒線と同程度の傾きをした良いスケールを示すが，p が大きくなるにつれスケールが悪くなっていく．ツリー構築は，必ずしも重力計算の度に毎回行わなくても良いので (その場合は粒子の移動に合わせてセルの座標も変更するなどの措置をとる必要がある)，効率化は可能である．カーネル関数+ツリーデータの通信については，先に説明したように通信の隠蔽を図れば改善が見込めると考えている．

7. 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である “ツリー法” を高速化した．先行研究である Nakasato 2012 の提案手法を元に，それを発展させることで更なる高速化を目指した．

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 分裂数を減らすこと” である．そのために，重力を計算される複数の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，ベクトル化・グループ化なしの場合に比べて 248%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 曲線を利用して，空間座標において近傍の粒子をメモリ空間上でも近隣に位置するよう並び替え，その影響を調べた．その結果，Peano-Hilbert

曲線を利用した方が，カーネル関数は数%から 10% 程度高速化された．

さらに，MPI を用いた並列 GPU 化も進めている．計画している並列化の基本方針は，領域分割や LET によって MPI 通信量を必要最低限に抑え，また，カーネル計算と MPI 通信をオーバーラップするなど，CPU の並列化でも用いられるような一般的な方法である．しかし，現在その全ての実装が済んでいるわけではなく，領域分割やツリー構築の効率化，カーネル計算中の通信隠蔽等，改善をすべき点がいくつか残されている．今後はまずこれらの問題を 1 つずつ解決していく予定である．

現在，$p = 1$(並列化なし) で，$N = 2^{24} = 16777216$ の重力計算を 1 回あたり数秒で行うことが可能 (図 14 左端) であり，この程度の規模の問題に対しては実際に計算天文学の研究に用いることができると言える．つまり，$p = 10 \sim 100$ で強スケールするような並列化ができれば，$N \sim 10^{8\sim9}$ の世界でも有数の大規模計算が可能になると期待している．

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，筑波大学システム情報工学研究科の児玉 祐悦教授，高橋 大介教授，塙 敏博准教授に深く感謝致します．

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] J. Barnes, P. Hut, A hierarchical $O[N \log(N)]$ force-calculation algorithm, Nature 324 446449 (1986).
[4] L. Nyland, M. Harris, J. Prins, Fast N-Body Simulation with CUDA (2007).
[5] 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).
[6] Nvidia, NVIDIA CUDA C Programming Guide Version 4.0 (2011).
[7] Khronos, The OpenCL Specification Version 1.2 (2011).
[8] 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).
[9] N., Nakasato, Implementation of a Parallel Tree Method on a GPU, Journal of Computational Science, Volume 3, Issue 3, Pages 132–141 (2012).
[10] N., Nakasato, G., Ogiya, Y., Miki, M., Mori, & K., Nomoto, Astrophysical Particle Simulations on Heterogeneous CPU-GPU Systems, arXiv:1206.1199 (2012).
[11] 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).
[12] J. F., Navarro, C. S., Frenk, & S. D. M., White, Astrophysical Journal, A Universal Density Profile from Hierarchical Clustering, 490, 493 (1997).
[13] 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).