Vol.2012-HPC-136 No.1
2012/10/3

大規模GPUクラスタにおける N体計算コードの演算性能とスケーラビリティの評価

三木 洋平[1,2,a)]　高橋 大介[3,5,b)]　森 正夫[4,5,c)]

概要：我々は，CUDA/OpenMP/MPI を用いて実装した N 体計算コードを最適化し，大規模 GPU クラスタ上で性能評価を行った．本実装では，スケーラビリティを向上させるために，ノード間の通信回数を削減し，またノード間・ノード内の通信を計算と同時に行うことによって通信時間を隠蔽した．筑波大学の HA-PACS（Highly Accelerated Parallel Advanced system for Computational Sciences）上での性能測定の結果，高い演算性能，並列化効率が得られることが確かめられた．GPU 当たりの粒子数が 8192 体未満の場合にはスーパーリニア・スケーリングを示し，8192 体以上の場合には並列化効率はほぼ 100%となった．NVIDIA Tesla M2090 を 256 枚用いた際のピーク性能は単精度 254.0TFLOPS（理論ピーク性能の 74.5%）に達した．

1. はじめに

N 体計算とは，系を構成する恒星などの粒子どうしの間にはたらく自己重力を計算し，粒子軌道の時間発展を調べるという計算手法である．N 体計算は非常に直感的なモデル化に基づくため，宇宙物理学の研究では宇宙の大規模構造や銀河といった無衝突重力多体系の性質やその形成・進化過程を詳細に調べるために広く用いられている．粒子位置の時間発展を計算するためにはその速度情報が，速度の時間発展を計算するためには加速度情報が必要であり，加速度は Newton の運動方程式

$$\boldsymbol{a}_i = \sum_{\substack{j=0 \\ j\neq i}}^{N-1} \frac{G m_j \left(\boldsymbol{x}_j - \boldsymbol{x}_i\right)}{\left(\left|\boldsymbol{x}_j - \boldsymbol{x}_i\right|^2 + \epsilon^2\right)^{3/2}} \qquad (1)$$

によって計算される．ここで，G は重力定数であり，m_i, $\boldsymbol{x}_i$, $\boldsymbol{a}_i$ はそれぞれ i 番目の粒子の質量，位置，加速度である．重力ソフトニング ϵ はゼロ割による発散を防ぐために導入された定数であり，また重力加速度を計算する際に自己相互作用を取り除く役割も果たす．粒子が N 個ある場合の計算の演算量は，重力を感じる粒子（以下，i-粒子）の粒子数 N_i と重力を及ぼす粒子（以下，j-粒子）の粒子数 N_j それぞれに比例するため，$O(N^2)$ となる．

宇宙物理学の研究には膨大な粒子数を用いた N 体計算が必要とされるものが多いが，これには非常に長い計算時間が必要となるために十分な粒子数を用いた計算を遂行することは困難である．そのため，古くから多くの研究者が N 体計算の高速化に取り組んでおり，Particle-Mesh 法 [1] やツリー法 [2] といった高速な手法が提案されている．こうした高速化は演算量の削減によるものであり，例えばツリー法の演算量は，多重極展開を用いて N_j からの寄与を減らしたことで $O(N \log N)$ となっている．

しかし，宇宙物理学の研究で必要とされる計算においては，直接法を用いて N 体計算を行うべきものもある．球状星団の長時間進化がその一例であり，球状星団の寿命が力学的時間に対して非常に長いため，その進化は直接法を用いて調べられてきた．こうした系の進化を計算する際には膨大なステップ数の軌道積分が必要とされるため，近似を用いた不正確な重力計算では星団を構成する恒星の軌道進化を正しく計算できなくなるためである．衝突系である球状星団の進化を探る際に，本研究で扱う無衝突系向けの N 体計算コードを一部改変する必要があるが，本研究の実装方針や結果は衝突系の計算を高速化させる上でも有用な情報となるだろう．

また計算科学の研究という観点においても，N 体計算という単純かつ典型的な問題についてどれだけの性能やスケーリングが得られるかを調べておくことには価値がある．これは，こうした知見が，より複雑なアプリケーションを高速化する上での重要な手がかりとなるからである．

[1] 筑波大学大学院数理物質科学研究科
[2] 筑波大学大学院システム情報工学研究科
[3] 筑波大学システム情報系
[4] 筑波大学数理物質系
[5] 筑波大学計算科学研究センター
[a)] ymiki@ccs.tsukuba.ac.jp
[b)] daisuke@cs.tsukuba.ac.jp
[c)] mmori@ccs.tsukuba.ac.jp

また，$O(N^2)$ という N 体計算の演算量が実行可能な問題サイズに課す制約は非常に強いものである．しかしながら，こうした強い制約があるからこそ，最近の計算機の高い演算性能を十分に引き出すことで実行可能な問題規模を拡大していくことは計算科学が果たすべき重要な役割であると言える．

演算加速器の使用は，直接法に基づく N 体計算の高速化に関する有力な解決策の一つであり，重力多体系向けの専用計算機 GRAPE（“GRAvity PipE”）が特に有名である [3][4]．GRAPE は，重力相互作用の計算を並列パイプライン化することによって内部的に N 体計算を並列化し，高速化することに成功した．

近年，GPGPU（General Purpose computing on Graphics Processing Unit）の発展により，GPU が演算加速器として注目されている．最近の GPU の急速な性能向上，及び Tianhe-1A, Jaguar, Nebulae, TSUBAME 2.0, HA-PACS といった TOP 500 リスト [5] に名を連ねる GPU クラスタの登場によって，演算加速器の性能を引き出せるコードの重要性は増している．そのため，典型的な問題である N 体計算を GPU を用いてどこまで高速化できるか詳細に調べておくことは，計算科学として非常に重要な研究課題である．GPU を用いた N 体計算の高速化は今までも精力的に取り組まれてきた研究課題であり，無衝突系向けの直接計算法 [6][7][8][9][10]，無衝突系向けのツリー法 [8][9][11][12][13][14]，衝突系向けの直接計算法 [15][16][17][18] といった多くの先行研究がある．

本研究では，直接法を用いた無衝突系向け N 体計算コードを CUDA/OpenMP/MPI を用いて実装し，その性能評価を行う．第 2 節においては 1GPU 向けコードの実装と最適化について，第 3, 4 節ではそれぞれ OpenMP，MPI を用いた並列化について記述する．第 5 節において性能測定の結果を示し，第 6 節で議論する．

2. 1GPU 向け N 体計算コードの実装と性能最適化

多くの先行研究によって，i-粒子について大規模に並列化して計算することで GPU の高い演算性能を発揮できるということが示されている [6][7][8][9][10]．本研究における実装は，CUDA SDK[7] と Miki et al. (2012) [10] を基とし，さらなる最適化を施したものとなっている．

NVIDIA Tesla M2090 を用いた場合の単精度性能は，[7] は 930GFLOPS，[10] は 991GFLOPS である．どちらの実装においても，ブロックあたりのスレッド数は 256 であり，1 スレッドが 1 つの i-粒子に関する計算を受け持つ．また 256 個の j-粒子の位置情報を高速なシェアードメモリにロードした上で計算することで，最内側ループにおける低速なグローバルメモリへのアクセス回数を削減している．細かい違いとしては，最内側ループのアンローリングの段

```
/* CUDA SDK における実装 */
float r2 = rji.x * rji.x + rji.y * rji.y + rji
    .z * rji.z;
r2 += eps2;
/* Miki et al. (2012) における実装 */
float r2 = eps2 + rji.x * rji.x + rji.y * rji.
    y + rji.z * rji.z;
```

図 1 CUDA SDK との実装の違い

数，キャッシュの使用構成，重力相互作用を計算するために発行される命令数が挙げられる．先行研究である [7] においては 32 段のループアンローリングを施し，また “shared memory preferred” というキャッシュ構成となっている．これに対して，[10] においては予備実験の結果として多くの場合においてより高い性能を示した構成である，128 段のループアンローリングを施し，“L1 cache preferred” とする構成を採用した．

最後の違いは $\boldsymbol{r}_{ji}^2 + \epsilon^2$ の計算方法に関するもの（**図 1**）であり，この違いが性能に対して一番影響のある点である．図 1 において，`float3` 型の変数 `rji`，`float` 型の変数 `eps2`，`r2` にはそれぞれ変位ベクトル $\boldsymbol{r}_{ji} \equiv \boldsymbol{x}_j - \boldsymbol{x}_i$，重力ソフトニングの二乗 ϵ^2，$\boldsymbol{r}_{ji}^2 + \epsilon^2$ の計算結果を格納している．両者の計算方法はほぼ同じに見えるが，実行される命令は全く異なるものとなる．[7] においては，乗算 1 回と FMA（Fused Multiply-Add）命令が 2 回実行された後にさらに加算が 1 回実行されることになる．NVIDIA CUDA C Programming Guide [19] によれば，この計算を完了するためには 4 サイクルが必要である．これに対して [10] では，3 回の FMA 命令が 3 サイクルの間に実行されるだけで同じ計算を完了できるため，[7] よりも高速である．また本実装においては，性能低下の要因となりうる `if` 文をカーネル関数の中から排除した．これに伴い，任意の N_i, N_j に対して N 体計算を正しく取り扱うためには追加の工夫が必要となり，N_i や N_j がブロックあたりのスレッド数 256 の倍数でない時には質量ゼロの仮想粒子を適切な数だけ追加して計算することとした．

本実装では，グローバルメモリへのアクセス時間がより隠蔽されやすいように，Miki et al. (2012) に追加の最適化を施した．シェアードメモリに j-粒子の位置情報を保持するため，j-粒子の位置情報を更新する前後にはブロック内の全スレッドの同期をとる必要がある．複数のブロックが 1 つのストリーミング・マルチプロセッサ（以下 SM）に割り当てられている場合，あるブロックのグローバルメモリへのアクセス時間は他のブロックの計算と同時に行うことによって隠蔽されうる．しかしながら，CUDA スケジューラが計算を実行可能なウォープを見つけられない限りは，このような隠蔽処理は実行されない．そのため，こうしたメモリアクセスと計算の同時実行が起こる確率を高めておくことは性能向上の一助となることが期待できる．

```
__global__ void calc_gravity(int Ni, float4 *
    ipos, float4 *acc, int Nj, float4 *jpos)
{
  __shared__ float4 body[256];
  int  i = blockIdx.x * blockDim.x + threadIdx
      .x;
  int ii =                              threadIdx
      .x;

  /* set i-particles */
  /* x, y, z, and mass */
  float4 pi = ipos[i];
  /* ax, ay, az, and potential */
  float4 ai = {0.0f, 0.0f, 0.0f, 0.0f};

  int nj = blockDim.x;
  for(int jh = 0; jh < Nj; jh += nj){
    /* set j-particles */
    float4 pj = jpos[jh + ii];
    __syncthreads();
    body[ii] = pj;
    __syncthreads();

#pragma unroll 128
    for(int j = 0; j < nj; j++){
      pj = body[j];

      float3 rji;
      rji.x = pj.x - pi.x;
      rji.y = pj.y - pi.y;
      rji.z = pj.z - pi.z;
      float rinv = rsqrtf(eps2 + rji.x * rji.x
           + rji.y * rji.y + rji.z * rji.z);

      /* ai.w += rinv * pj.w; */
      rinv = rinv * rinv * rinv * pj.w;
      ai.x += rji.x * rinv;
      ai.y += rji.y * rinv;
      ai.z += rji.z * rinv;
    }
  }
  /* acc[i] = ai; */
  atomicAdd(&(acc[i].x), ai.x);
  atomicAdd(&(acc[i].y), ai.y);
  atomicAdd(&(acc[i].z), ai.z);
  atomicAdd(&(acc[i].w), ai.w);
}
```

図 2 カーネル関数の実装

そこで本実装では，グローバルメモリからのロード命令を 2 度の__syncthreads() の実行で挟まれた部分の外側に記述することとした（詳細は**図 2** 参照のこと）．こうした注意深い取り扱いによってグローバルメモリへのアクセス時間が隠蔽されやすくなったことで性能が向上し，ピーク性能は単精度 1004GFLOPS に達した．

本実装では，図 2 の最後の部分に示したように，計算が完了した i-粒子の加速度情報をグローバルメモリへストアする際にアトミック演算を用いた．この処理は，1GPU 用のコードを記述する際には必要ないが，複数枚の GPU を用いて計算する際に正しく計算を行うために必要となる処理である．

また，重力ポテンシャル $-\sum_{j\neq i} Gm_j/\sqrt{|\boldsymbol{x}_j - \boldsymbol{x}_i|^2 + \epsilon^2}$ の計算も必要な場合には，図 2 中でコメントアウトしてある ai.w += rinv * pj.w を計算すれば良い．本実装においては，カーネル関数の中から if 文を排除したため，このままでは $j = i$ の場合である自己相互作用の重力ポテンシャルへの寄与を取り除けていない．そこで，カーネル関数の外側において自己相互作用からの寄与である $-Gm_i/\epsilon$ を計算された重力ポテンシャルから差し引くことで自己相互作用を取り除いた．こうした単純な取り扱いは，$\epsilon \neq 0$ である限り有効であり，これは無衝突 N 体計算においては一般的な状況設定である．

3. OpenMP を用いた並列化

ここでは，2 枚の GPU を用いた並列化について議論する．N 体粒子のデータを 2 枚の GPU に分散させた場合には，重力計算を進めるためには 2GPU 間の通信が必要となる．この際，2 枚の GPU が PCI レーンを共有している場合には，CPU 上のメモリを介さずに 2 枚の GPU 上のデータを通信できるピアツーピア・メモリアクセス（以下 P2P アクセス）を用いることができる．P2P アクセスを用いるためには，1 プロセスが対象とする全 GPU 上のグローバルメモリへのポインタを保持している必要があるため，本実装では OpenMP を用いた並列化を施すこととした．

重力相互作用の計算と軌道積分は，**図 3** に示したアルゴリズムに則って実行した．本実装では，1 つの OpenMP スレッドが 1 枚の GPU を，そしてそれぞれの GPU が全体の半分の N 体粒子を担当することとしたため，各 GPU は $N/2$ 個の i-粒子の軌道積分と $N/2$ 個の i-粒子に対する N 個の j-粒子からの重力相互作用を計算することとなる．

この過程において，カーネル関数の実行（図 3 の 4 ステップ目）は GPU 間の通信（図 3 の 3 ステップ目）の完了後に行わなければならないため，より多くの GPU を用いた際に並列化効率を落とさないためには更なる工夫が必要である．図 3 においては，2 行目と 3 行目の処理は完全に独立であるため同時に実行可能である．これは，GPU 間の通信は 2 行目の重力相互作用の計算と同時に行うことで隠蔽されうるということを意味しており，本実装では 2 つの CUDA ストリームを用いることでこれを実現した．具体的には，一つ目の CUDA ストリームが 1 行目と 2 行目の処理を担当し，もう一方の CUDA ストリームが 1 行目

1: 各 GPU が保持する $N/2$ 個の i-粒子の位置情報を更新
2: 各 GPU が保持する $N/2$ 個の i-粒子間の重力相互作用を計算
3: 各 GPU が保持する $N/2$ 個の i-粒子の位置情報を，もう一方の GPU に j-粒子の位置情報として送信
4: 各 GPU が保持する $N/2$ 個の i-粒子が，もう一方の GPU から受け取った $N/2$ 個の j-粒子から受ける重力を計算
5: 各 GPU が保持する $N/2$ 個の i-粒子の速度情報を更新

図 3 OpenMP を用いた並列アルゴリズム

の処理が完了した後に 3 行目，4 行目の処理を実行するように実装した．

4. Message Passing Interface を用いた並列化

使用する GPU 数を 2 枚から増やし，複数ノードからなる GPU クラスタを用いるためには，Message Passing Interface（MPI）による並列化が必須となる．第 3 節において実装したアルゴリズム（図 3）を拡張するのがもっとも単純な実装方針であるが，本研究ではさらに MPI プロセス間の通信回数を削減することで高いスケーラビリティを発揮できるように工夫を施した．本研究において実装した通信アルゴリズムは，以下で説明する輸送段階と蓄積段階の 2 つに大別される．

まず，**図 4** に示した輸送段階のアルゴリズムは，図 3 で示したアルゴリズムの複数ノード向けの自然な拡張である．この輸送段階においては，MPI プロセスを 1 次元リング状に配置しておき，j-粒子のデータを隣のプロセスに対して一方向に転送していくという処理を全ての j-粒子からの寄与を計算し終えるまで繰り返す．このアルゴリズムはリング上の通信パターンとなるため，輸送段階に関わる全プロセス数を n_{t} とすると，その通信回数は $n_{\mathrm{t}} - 1$ 回となる．

1: **for** $s = 0$ to $N_{\mathrm{proc}} - 2$ **do**
2: 　CPU 上の配列 0 のデータを，隣のプロセスの配列 1 に送信
3: 　配列 1 上の新たな j-粒子データを，CPU から GPU へ転送
4: 　i-粒子に対する j-粒子からの重力を計算
5: 　CPU 上で，配列 0 と配列 1 を入れ替え
6: **end for**

図 4　輸送段階のアルゴリズム

1: **while** j-粒子が配列 0，1 の容量に収まる，**or** 蓄積された j-粒子の粒子数が，全 N 体粒子の粒子数の 1/2 を越えない **do**
2: 　j-粒子の位置情報を交換する相手となる MPI プロセスを計算
3: 　CPU 上の配列 0 のデータを，相手プロセスの配列 1 に送信
4: 　配列 1 上の新たな j-粒子データを，CPU から GPU へ転送
5: 　i-粒子に対する j-粒子からの重力を計算
6: 　配列 0 のデータを配列 1 のデータに追加
7: 　CPU 上で，配列 0 と配列 1 を入れ替え
8: **end while**

図 5　蓄積段階のアルゴリズム

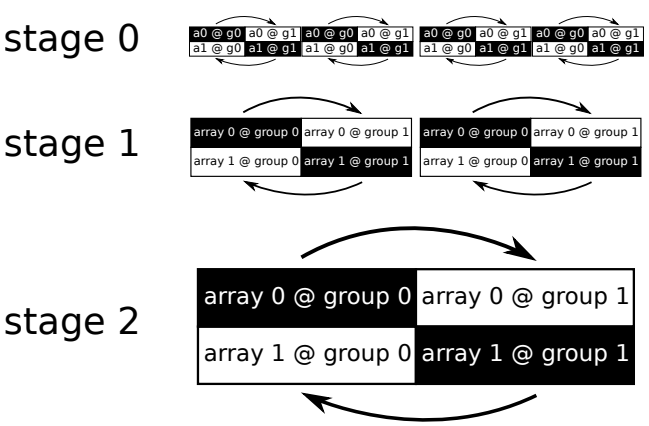

図 6　蓄積段階の模式図

CPU-GPU 間の通信と GPU 上での計算を同時に行うことで通信時間を隠蔽するために，ここでも第 3 節と同様に 2 つの CUDA ストリームを用いる．さらに，`MPI_Isend` や `MPI_Irecv` といったノンブロッキングな MPI 関数を用いて MPI 通信を GPU 上での計算と同時に行うことで，通信によって並列化効率が下がらないようにした．

図 5 に，通信回数を削減するために実装した蓄積段階のアルゴリズムを示した．蓄積段階の基本的な考え方は，前段階までの通信で受け取った全ての j-粒子の位置情報を次の通信時に送信するという，非常に単純なものである．各段階における j-粒子データの通信の様子は，**図 6** に示したようなものであり，グループ 0 または 1 に所属する MPI プロセスに保持されている j-粒子の位置情報を相互に通信し合うことによって共有，合体していくようになっている．通信されるデータサイズは段階ごとに倍々に増えていき，その結果必要となる通信回数は MPI プロセス数を n_{a} とすると二分木の性質を反映して $\log_2 n_{\mathrm{a}}$ となり，輸送段階を採用した場合の $n_{\mathrm{a}} - 1$ 回よりも少ない．

こうした通信パターンは，自分自身の MPI ランクと `1 << (stage id)` の排他的論理和が MPI ランクとなるプロセスを通信相手として j-粒子の位置情報を送りあうことで実現できる．また，この蓄積段階は木構造的なデータ通信を伴うため，ツリー状のネットワーク・トポロジのクラスタには適しているが，遠方の MPI プロセスとの通信を伴うためメッシュ・トーラスには向いていない．

全体の通信アルゴリズムは，j-粒子のデータサイズが確保した配列の容量に収まっている間は蓄積段階を繰り返し，その後輸送段階に移行し全ての j-粒子からの重力を計算し終えるまでループを回すというものになっている．

5. 性能測定

本研究における性能測定には，筑波大学に導入された GPU クラスタ HA-PACS（Highly Accelerated Parallel Advanced system for Computational Sciences）を用いた [20]．HA-PACS の計算ノードは，2 ソケットの Intel Sandy Bridge-EP と 4 枚の NVIDIA Tesla M2090 で構成されており，CPU と GPU は PCI-express generation 3.0 で接続されている．ネットワーク構成は Fat-Tree であり，各計算ノードは 2 系統の Infiniband QDR で結合されている．HA-PACS の理論ピーク性能は単精度 1604TFLOPS であり，これには GPU の 1427TFLOPS という高い演算性能の寄与が大きい．その他の HA-PACS に関する情報は，**表 1** にまとめたとおりである．HA-PACS の各ソケット内の 2 枚の GPU は PCI レーンを共有しており，P2P アクセスを用いた GPU 間の直接通信が可能である．そのため，P2P アクセスを用いた本実装のテスト環境として適し

表 1 測定環境

Number of nodes	268
CPU	Intel Xeon E5-2670
	16 cores per node, 2.6 GHz
RAM	128 GB (DDR 3, 1600 MHz)
GPU	NVIDIA Tesla M2090
	512 CUDA cores, 1.3 GHz
	4 boards per node
Video RAM	6 GB (GDDR 5, ECC on) per GPU
C Compiler	icc 12.1.0
MPI Library	Intel MPI 4.0.3.008
CUDA toolkit	nvcc 4.1, CUDA SDK 4.1.28
Interconnection	Infiniband QDR ×2 rails
Network topology	Fat-Tree

た GPU クラスタであると言える．さらに，ネットワーク構成が Fat-Tree となっているため，本実装において採用した蓄積段階の MPI 通信に適した構成となっている．

性能測定の際には，性能を正確に測定するための工夫が必要である．測定精度を向上するため，我々はカーネル関数を繰り返し実行し，その全実行時間を測定することとした．しかし，本実装においては計算と通信を同時に行うために 2 つの CUDA ストリームを用いている（第 3，4 節）ため，カーネル関数の最後で `cudaStreamSynchronize()` を実行することによって，2 つのストリームを同期しなければならない．仮にこのストリーム同期を怠れば，1 つ前のステップでのカーネル関数と別のストリームに属する，現在のステップでのカーネル関数が同時に実行されうることになり，性能を過大評価することとなってしまう．

しかし，ここで追加した同期処理の実行時間は本来 N 体計算の実行時間には含まれないものであり，特に，粒子数 N が小さい時にはこの影響が無視できない．そこで，本実装の演算性能を正確に測定するためには，追加の同期処理に要する時間を独立に測定しておき，N 体計算コードの実行時間から差し引くようにしなければならない．

このため，N_{sub} 回の空カーネルの非同期実行と同期処理（1GPU の場合は `cudaStreamSynchronize()`，2GPU の場合は `cudaStreamSynchronize()` と `omp barrier`，4GPU 以上の場合には `cudaStreamSynchronize()`, `omp barrier` と `MPI_Barrier()`）の全実行時間 t_{tot} を測定することで，同期処理に要する時間 t_{sync} を求めた．**図 7** に，測定した実行時間 t_{tot} を空カーネルの呼び出し回数 N_{sub} の関数としてプロットした．黒丸，赤の正方形，その他のシンボルはそれぞれ 1GPU，2GPU，4GPU 以上の場合の測定結果であり，どの場合においても 1 次関数として振る舞うことが分かる．カーネル関数の起動時間を t_{kernel} とすれば，ここで測定した全実行時間 t_{tot} は $t_{\mathrm{sync}} + N_{\mathrm{sub}} t_{\mathrm{kernel}}$ と表現できるので，測定結果を最小二乗法を用いてフィットすることで t_{sync}，t_{kernel} が求まる．図 7 における線は最小二乗法に基づくフィットの結果であり，求まった t_{sync}，t_{kernel}

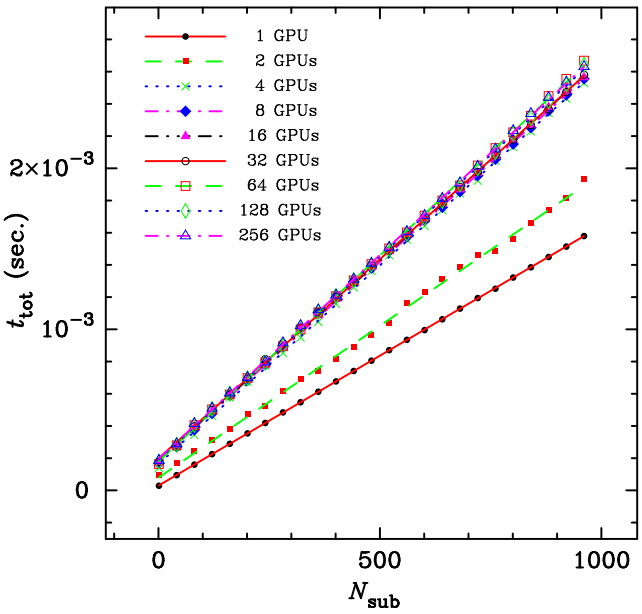

図 7 同期処理とカーネル関数起動に要する時間

表 2 同期処理，カーネル関数起動の実行時間

Number of GPUs	t_{sync} (sec.)	t_{kernel} (sec.)
1	2.97×10^{-5}	1.61×10^{-6}
2	7.84×10^{-5}	1.89×10^{-6}
4	1.64×10^{-4}	2.47×10^{-6}
8	1.85×10^{-4}	2.47×10^{-6}
16	1.89×10^{-4}	2.48×10^{-6}
32	1.99×10^{-4}	2.47×10^{-6}
64	1.79×10^{-4}	2.56×10^{-6}
128	1.89×10^{-4}	2.52×10^{-6}
256	1.96×10^{-4}	2.53×10^{-6}

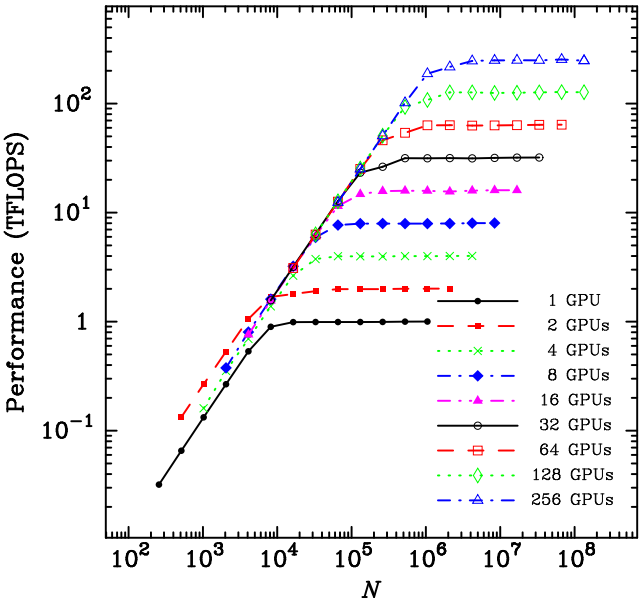

図 8 単精度演算性能の粒子数 N と GPU 数に対する依存性

の値は**表 2** にまとめたとおりである．

本実装の性能評価は，使用する GPU 数を 1, 2, 4, ⋯, 256 枚として行った．また最大の粒子数は 134217728 と

し，GPU 当たりの粒子数が 256 体から 1048576 体の範囲における性能評価を行った．**図 8** に，測定結果を横軸に全粒子数 N をとり，縦軸を単精度での演算性能として用いた GPU 数ごとにプロットした．また測定時間から演算性能へと変換する際には，一相互作用を計算するのに単精度で 26FLOP を要するという仮定を用いた．この仮定は，compute capability 2.0 の GPU については最も妥当な仮定である [10][19]．ピーク性能は粒子数 $N = 67108864$，GPU 数 256 のときに 254.0TFLOPS に達した．

6. 演算性能のモデル化

6.1 1GPU 向けコードに対する性能モデル化

ここでは，1GPU のみを用いた場合の性能モデル化を行う．しかし，本実装は性能向上のために複数の命令が可能な限り同時に実行されるように実装したため，解析的なモデル化は困難である．そこで，複数の未知パラメータを用いたモデル式を構築しておき，性能評価の結果を用いて未知パラメータの値を見積もることによって性能のモデル化を行う．

まず，カーネル関数を起動する際には C_{kernel} クロックサイクルだけの起動コストが必要である．また，計算を開始する前には i-粒子の位置情報をグローバルメモリからロードし，計算を終了する際には i-粒子の加速度情報をグローバルメモリへとストアしなければならない．グローバルメモリにアクセスする際には，$400 - 800$ クロックサイクルに及ぶレイテンシ L が生じる [19]．このレイテンシは非常に大きいため，i-粒子のデータサイズ（位置情報，加速度情報共に 1 粒子当たり 16 バイト）やメモリバンド幅（M2090 では 177.6GB/s）のデータ転送時間への寄与は無視でき，グローバルメモリに対するロード・ストアに要する時間はほぼ L だけで決まってしまう．このため，カーネル関数内の相互作用計算以外の寄与は $C_{\mathrm{kernel}} + 2L$ と見積もられる．

次に，カーネル関数内部において相互作用を計算するループの実行に要するクロックサイクル数であるが，図 2 にソースコードを示したとおり，j-粒子の位置情報をグローバルメモリからシェアードメモリにコピーし，シェアードメモリ上の全 j-粒子から受ける重力を計算するというループを，グローバルメモリ上にある全 j-粒子からの寄与を計算し終えるまで繰り返すという手続きになっている．そのため，この部分はグローバルメモリからデータを取得する際のレイテンシ L，シェアードメモリに蓄える j-粒子の粒子数 T_{all}（この値は，ブロックあたりのスレッド数と揃えてある），一相互作用の計算に要するクロックサイクル数 C_{calc}，全 j-粒子数 N_j を用いて定式化できる．

ループ 1 回の計算に要するクロックサイクル数は $L + T_{\mathrm{all}}C_{\mathrm{calc}}$ となるが，1 つの SM に同時に複数のブロックが含まれている場合（本実装では，1 つの SM に同時に割り当てられるブロック数は 2 である）には，レイテンシ L と計算 $T_{\mathrm{all}}C_{\mathrm{calc}}$ のうち時間の短い方の実行時間が隠蔽されることになる．また，compute capability 2.0 の GPU については，SM 当たりの CUDA コアの数は 32 であるので，T_{all} スレッドが担当する計算は $T_{\mathrm{all}}/32$ 回に分けて実行されることとなる．そのため，$T_{\mathrm{all}} \max(L, T_{\mathrm{all}}C_{\mathrm{calc}})/16$ が，1 つの SM に割り当てられた 2 つのブロックがループ 1 回分の計算に要するクロックサイクル数である．

このループが 1 回処理されるごとに T_{all} 個の j-粒子からの寄与が計算されるので，全 j-粒子数 N_j からの寄与を計算し終えるためには N_j/T_{all} 回だけループが実行されることになる．その結果，T_{all} 個の i-粒子に対する重力の計算は $N_j \max(L, T_{\mathrm{all}}C_{\mathrm{calc}})/16$ クロックサイクルで実行されると見積もられる．

以上の議論によって，1 つの SM に割り当てられた 2 つのスレッドブロックが担当する計算を完了するのに要するクロックサイクル数が見積もられたので，全ブロック数と，SM の数から全体のクロックサイクル数が求まる．全ブロック数は，全 i-粒子数 N_i とブロック当たりの i-粒子数 T_{all} から N_i/T_{all} と計算できる．本研究で用いた GPU である NVIDIA Tesla M2090 では SM の数は 16 であるので，1 つの SM には同時に 2 つのスレッドブロックが割り当てられることに注意すると，スレッドブロックの投入は $\mathrm{ceil}(N_i/(32T_{\mathrm{all}}))$ 回だけ行われる．したがって，N_i 個の i-粒子，N_j 個の j-粒子系における N 体計算を行うためには，$T_{\mathrm{all}} = 256$ を代入して，

$$\mathrm{ceil}\left(\frac{N_i}{8192}\right)\left[\frac{N_j}{16}\max(L, 256C_{\mathrm{calc}}) + C_{\mathrm{kernel}} + 2L\right] \tag{2}$$

クロックサイクルを要することが分かった（より詳細，かつ一般的な導出については，[10] を参照されたい）．

以下では，式 (2) における未知パラメータである C_{calc}, L, C_{kernel} を性能測定の結果を用いて見積もっていく．N_j が十分に大きい場合には，カーネル関数の起動・終了コストを表す $C_{\mathrm{kernel}} + 2L$ の項が無視できる．これより，$N_i = N_j = 1048576$ の時のカーネル関数の実行時間 28.9 秒という結果から，$\max(L, 256C_{\mathrm{calc}})$ の項の寄与が見積もられる．式 (2) を M2090 のクロック周波数 1.3GHz で割ったものが理論的に見積もられる実行時間であるので，$\max(L, 256C_{\mathrm{calc}})$ の項の寄与は

$$\begin{aligned}\max(L, 256C_{\mathrm{calc}}) &= \frac{28.9 \times 1.3 \times 10^9}{(1048576/8192) \times (1048576/16)} \\ &\sim 4479\end{aligned} \tag{3}$$

クロックサイクルになることが分かる．ところで，[19] によれば，$L \sim 400 - 800$ クロックサイクル程度であり，ここで得られた値とは大きく異なる．したがって，ここで求まったクロックサイクル数は C_{calc} の寄与によるものと考えられ，$C_{\mathrm{calc}} \sim 17.5$ と求まる．また C_{kernel} については，表 2 に示したようにカーネル起動に要する時間がすでに求

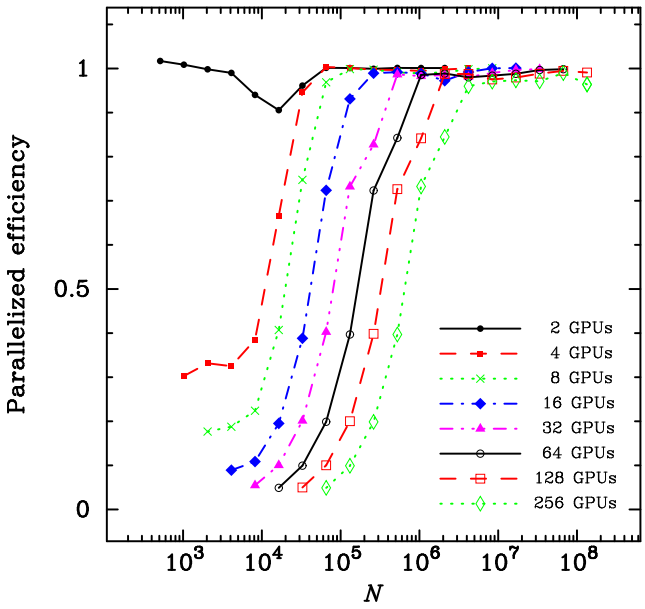

図 9 並列化効率 $P(N;N_{\rm GPU})/(P(N;1)\times N_{\rm GPU})$ の全粒子数 N 依存性（ストロング・スケーリングに相当）

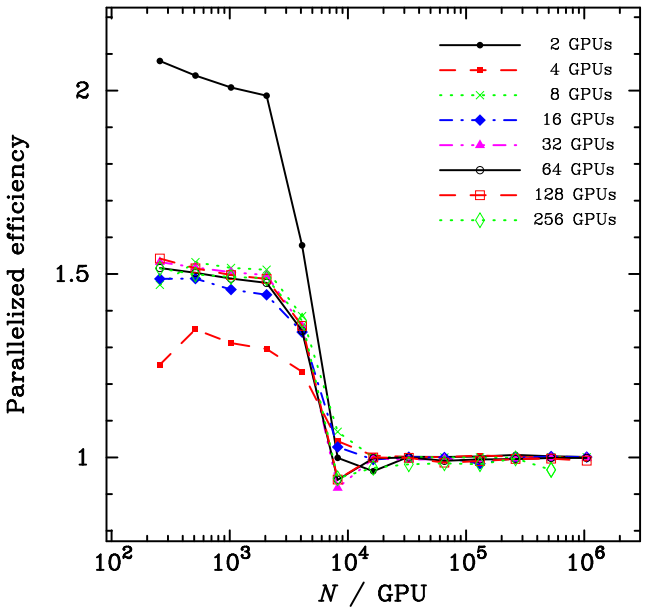

図 10 並列化効率 $P(N/N_{\rm GPU};N_{\rm GPU})/(P(N/N_{\rm GPU};1)\times N_{\rm GPU})$ の GPU 当たりの粒子数（$N/N_{\rm GPU}$）依存性（ウィーク・スケーリングに相当）

まっているので，これに M2090 のクロック周波数を乗じることで $C_{\rm kernel}\sim 2100$ クロックサイクルとなることが分かる．

最後に，図 8 における 1GPU の演算性能の振る舞いを，式 (2) を用いて簡単に説明しておく．粒子数 $N(=N_i=N_j)$ が $N\le 8192$ の時には，$\mathrm{ceil}(N_i/8192)=1$ となるため，式 (2) より計算に必要なクロックサイクル数は N_j のみに比例する．これに対して総演算量は N_iN_j に比例するため，粒子数 N の増加と共に演算性能は N_i に比例して向上していくことが分かる．これを言い換えると，粒子数 N が少ない時には計算に用いられていなかった SM が，粒子数 N の増加によって用いられるようになったことによる性能向上である．つまり，$N\le 8192$ の領域において粒子数 N と共に演算性能が低下するのは，GPU の全ての SM を有効に使うことができていないためである．

一方，$N\ge 8192$ の時には，$\mathrm{ceil}(N_i/8192)\propto N_i$ となり，計算に必要なクロックサイクル数も N_iN_j に比例することとなり，演算性能の増加は止まる．以上のような式 (2) の N_i に対する依存性によって，図 8 に示した 1GPU の演算性能の振る舞いが $N=8192$ を境に変化する理由が説明できる．

6.2 複数 GPU 向けコードに対する性能モデル化

ここからは，複数枚の GPU を用いた際の演算性能についての解析を進めていく．以下では，粒子数 N，$N_{\rm GPU}$ 枚の GPU を用いた際の演算性能を $P(N;\,N_{\rm GPU})$ と書く．

まず，ストロング・スケーリングを調べるために，並列化効率 $P(N;N_{\rm GPU})/(P(N;1)\times N_{\rm GPU})$ を**図 9** に示した．横軸を全粒子数 N として，用いた GPU 数ごとに並列化効率をプロットした．OpenMP のみによって 2GPU を用いた場合の並列化効率は幅広い範囲の N にわたって 1 となっており，2GPU 間の通信の隠蔽がうまく働いていることを示唆している．一方で，OpenMP/MPI によって 4GPU 以上を用いた場合の並列化効率は，OpenMP のみを用いた場合とはまったく異なる振る舞いを示している．

この 4GPU 以上の振る舞いを理解するため，ウィーク・スケーリングに相当する量である，並列化効率 $P(N/N_{\rm GPU};N_{\rm GPU})/(P(N/N_{\rm GPU};1)\times N_{\rm GPU})$ を GPU あたりの粒子数 $N/N_{\rm GPU}$ に対してプロットした（**図 10**）．図 10 においては，$P(N/N_{\rm GPU};N_{\rm GPU})$ の計算に用いられているカーネル関数が $P(N/N_{\rm GPU};1)$ とまったく同じであるため，図 9 に比べて並列化効率の振る舞いについて議論しやすい．図 10 に示した並列化効率を見ると，$N/N_{\rm GPU}\ge 8192$ においては並列化効率が 1 に収束していることが分かる．CPU-GPU 間や MPI プロセス間の通信に要する時間が計算時間に比べて十分に短い，あるいは計算との同時実行によって完全に隠蔽されている場合においては，各 GPU において並列化前のカーネル関数の演算性能がそのまま発揮される．そのため，GPU あたりの粒子数が多い時に並列化効率が 1 に収束するという結果は，当然の結果であると言える．

一方の $N/N_{\rm GPU}<8192$ における並列化効率は，4GPU 以上を使用した場合において 1.5 程度の値に達するという，非常に興味深い結果になっている．このスーパーリニア・スケーリングは，通信時間を隠蔽するために通信と計算を同時実行させたことの副産物であると考えられる．

通信と計算の同時実行を実現するために複数の CUDA ストリームを用いる必要があったため，本実装では重力計算を 2 つの独立な CUDA ストリームに関連付けて計算し

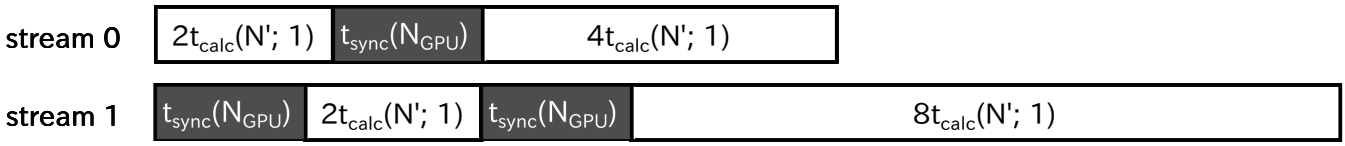

図 11 異なる CUDA ストリームに属するカーネル関数の実行イメージ

ている（第 3，4 節）．GPU あたりの粒子数が 8192 体以上である場合には，2 つの独立な CUDA ストリームを用いる利点は通信と計算の同時実行を可能とする点だけであるが，$N/N_{\mathrm{GPU}} < 8192$ の場合にはさらなる利点がある．第 6.1 節において議論したように，$N/N_{\mathrm{GPU}} < 8192$ の場合においては GPU 内に未使用の SM が残されている．したがって，この未使用の SM を 2 つ目の CUDA ストリームに関連付けられたカーネル関数が使用することによって，演算性能 $P(N/N_{\mathrm{GPU}}; N_{\mathrm{GPU}})$ がリニア・スケーリングの場合に比べて 2 倍まで向上することが可能となる．これが $N/N_{\mathrm{GPU}} < 8192$ において 2 倍までのスーパーリニア・スケーリングが得られる理由である．

以下では，$N/N_{\mathrm{GPU}} < 8192$ における並列化効率 $P(N/N_{\mathrm{GPU}}; N_{\mathrm{GPU}})/(P(N/N_{\mathrm{GPU}}; 1) \times N_{\mathrm{GPU}})$ が上限値である 2 よりも小さくなる理由について考察する．ここでは，GPU 当たりの粒子数が少ない状況における通信パターンである，蓄積段階におけるカーネル関数の実行の様子を簡単にモデル化する．

2 つの CUDA ストリームに属する命令が実行されていく様子を，横軸にとった時間の関数として模式図に示した（**図 11**）．$t_{\mathrm{calc}}(N'; 1)$，$t_{\mathrm{sync}}(N_{\mathrm{GPU}})$ はそれぞれ GPU あたりの粒子数 $N' \equiv N/N_{\mathrm{GPU}}$ の計算を 1GPU で実行した時のカーネル関数の所要時間，N_{GPU} 枚の GPU を同期するのに要する時間である．蓄積段階においては，j-粒子の粒子数 N_j は MPI 通信を完了させる度に 2 倍ずつ増えていくため，カーネル関数の実行時間も 2 倍ずつ増えていく．唯一の例外はストリーム 0 の開始時であるが，これは第 3 節において述べた OpenMP を用いた並列化によるものである．

性能測定の結果から，カーネル関数の実行時間 $t_{\mathrm{calc}}(N; 1)$ は**表 3** にまとめたとおりであり，$t_{\mathrm{sync}}(N_{\mathrm{GPU}})$ も表 2 に示したようにすでに求まっている．また，実際には CPU-GPU 間や MPI プロセス間のデータ通信に要する時間も全体の計算時間に寄与するが，同期処理や計算時間に比べるとその寄与は小さいため，このモデル化においては無視した．

表 3 カーネル実行に要する時間の粒子数 N 依存性

N	$t_{\mathrm{calc}}(N; 1)$ (sec.)
256	5.32×10^{-5}
512	1.04×10^{-4}
1024	2.06×10^{-4}
2048	4.09×10^{-4}
4096	8.16×10^{-4}
8192	1.94×10^{-3}

図 11 に示したカーネル関数の実行モデルから得られる全実行時間は，N_{GPU} が 4 の倍数でない場合（ストリーム 0 に対応）には

$$t_{\mathrm{calc}}(N/N_{\mathrm{GPU}}; 1) \left(1 + \sum_{i=0}^{\log_4(N_{\mathrm{GPU}}/2)} 4^i\right) + t_{\mathrm{sync}}(N_{\mathrm{GPU}}) \log_4(N_{\mathrm{GPU}}), \quad (4)$$

また N_{GPU} が 4 の倍数となる場合（ストリーム 1 に対応）には

$$2t_{\mathrm{calc}}(N/N_{\mathrm{GPU}}; 1) \sum_{i=0}^{\log_4(N_{\mathrm{GPU}}/2)} 4^i + t_{\mathrm{sync}}(N_{\mathrm{GPU}}) \log_4(N_{\mathrm{GPU}}) \quad (5)$$

と表される．したがって，ウィークスケーリングを意味する並列化効率 $P(N/N_{\mathrm{GPU}}; N_{\mathrm{GPU}})/(P(N/N_{\mathrm{GPU}}; 1) \times N_{\mathrm{GPU}})$ は，ストリーム 0 とストリーム 1 それぞれに対して

$$\left[\left(1 + \sum_{i=0}^{\log_4(N_{\mathrm{GPU}}/2)} 4^i\right) / N_{\mathrm{GPU}} + \frac{t_{\mathrm{sync}}(N_{\mathrm{GPU}}) \log_4(N_{\mathrm{GPU}})}{t_{\mathrm{calc}}(N/N_{\mathrm{GPU}}; 1) N_{\mathrm{GPU}}}\right]^{-1}, \quad (6)$$

$$\left[\frac{2}{N_{\mathrm{GPU}}} \sum_{i=0}^{\log_4(N_{\mathrm{GPU}}/2)} 4^i + \frac{t_{\mathrm{sync}}(N_{\mathrm{GPU}}) \log_4(N_{\mathrm{GPU}})}{t_{\mathrm{calc}}(N/N_{\mathrm{GPU}}; 1) N_{\mathrm{GPU}}}\right]^{-1} \quad (7)$$

と見積もられる．

ここで見積もった理論並列化効率 (6), (7) は非常に複雑で，並列化効率の振る舞いを理解しづらいため，ここでは 2 つの相補的な極限をとって議論する．

一つ目の極限として，並列化効率が最大となる極限である $t_{\mathrm{sync}}(N_{\mathrm{GPU}}) = 0$ の場合を考える．この極限を考えることによって見積もられる並列化効率の上限は，4, 8, ⋯, 256GPU を用いた際にそれぞれ 2.0, 1.33, 1.6, 1.45, 1.52, 1.49, 1.51 となる．

もう一方の極限として，並列化効率の下限値を見積るために，$t_{\mathrm{sync}}(N_{\mathrm{GPU}}) = 2t_{\mathrm{calc}}(N/N_{\mathrm{GPU}}; 1))$ という場合を考える．現在考えている $N/N_{\mathrm{GPU}} < 8192$, $N_{\mathrm{GPU}} \geq 4$ という状況設定の下では，常に $t_{\mathrm{sync}}(N_{\mathrm{GPU}}) < 2t_{\mathrm{calc}}(N/N_{\mathrm{GPU}}; 1))$ が成り立っているため，この極限は $t_{\mathrm{sync}}(N_{\mathrm{GPU}})$ の上限をとったことになっている．この時に見積もられた並列化効率は，4, 8, ⋯, 256GPU を用いた際にそれぞれ 1.0, 1.0,

1.14, 1.23, 1.33, 1.39, 1.44 と見積もられる．

以上の見積もりでは，並列化効率の振る舞いを詳細に説明することまではできていないが，図 10 に表れた大まかな振る舞いについては説明できている．またここでの見積もりからは，使用する GPU 数を増やすに連れて並列化効率が 1.5 に収束することが示唆され，これは測定結果の振る舞いと一致している．

7. まとめ

本研究では，直接計算法に基づく無衝突系向け N 体計算コードを実装し，GPU クラスタ向けの性能最適化を施した．本実装では，特にスケーラビリティを向上させるために，通信回数の削減や通信と計算の同時実行による通信時間の隠蔽といった処理を行った．

本実装のピーク性能は，256 枚の NVIDIA Tesla M2090 を用いて $N = 67108864$ の計算を行った際に，理論ピーク性能の 74.5%である単精度 254.0TFLOPS に達した．GPU 当たりの粒子数が 8192 未満の場合には 1.5 倍程度のスーパーリニア・スケーリングが，8192 以上の場合にはほぼ 100%の並列化効率が実現されることが，性能測定の結果分かった．またこうした振る舞いについては，本研究において構築した性能モデルによってよく説明できることが分かった．

謝辞 詳細な議論に付き合ってもらった筑波大学数理物質科学研究科の扇谷 豪 氏，有益なコメントを下さった筑波大学システム情報系の朴 泰祐 教授に感謝します．また，本研究における性能測定は，筑波大学計算科学研究センターの HA-PACS を用いて行いました．試用期間中であったにも関わらず HA-PACS における大規模計算を認めていただき，また数々の技術的なサポートをしていただいた HA-PACS プロジェクトチームに感謝します．

参考文献

[1] Hockney, R. W. and Eastwood, J. W.: *Computer simulation using particles*, (1988).
[2] Barnes, J. and Hut, P.: A hierarchical O(N log N) force-calculation algorithm, Nature, 324, 446 (1986).
[3] Okumura, S. K., Makino, J., Ebisuzaki, T., Fukushige, T., Ito, T., Sugimoto, D., Hashimoto, E., Tomida, K. and Miyakawa, N.: Highly Parallelized Special-Purpose Computer, GRAPE-3, Publications of the Astronomical Society of Japan, 45, 329 (1993).
[4] Kawai, A., Fukushige, T., Makino, J. and Taiji, M.: GRAPE-5: A Special-Purpose Computer for N-Body Simulations, Publications of the Astronomical Society of Japan, 52, 659 (2000).
[5] TOP500 List, 入手先 ⟨http://www.top500.org/⟩ .
[6] Hamada, T. and Iitaka, T.: The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units, ArXiv Astrophysics e-prints, arXiv:astro-ph/0703100 (2007).
[7] Nyland, L., Harris, M. and Prins, J.: Fast N-Body Simulation with CUDA, GPU Gems 3, 677 (2007).
[8] Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K. and Taiji, M.: 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, 62:1 (2009).
[9] Hamada, T. and Nitadori, K.: 190 TFlops Astrophysical N-body Simulation on a Cluster of GPUs, Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, 1 (2010).
[10] Miki, Y., Takahashi, D. and Mori, M.: A Fast Implementation and Performance Analysis of Collisionless N-body Code Based on GPGPU, Procedia Computer Science, 9, 96 (2012).
[11] Gaburov, E., Bédorf, J. and Portegies Zwart, S.: Gravitational tree-code on graphics processing units: implementation in CUDA, Procedia Computer Science, 1, 1119 (2010).
[12] Bédorf, J., Gaburov, E. and Portegies Zwart, S.: A sparse octree gravitational N-body code that runs entirely on the GPU processor, Journal of Computational Physics, 231, 2825 (2012).
[13] Nakasato, N.: Implementation of a parallel tree method on a GPU, Journal of Computational Science, 3, 132 (2012).
[14] 扇谷 豪, 三木 洋平, 朴 泰祐, 森 正夫, 中里 直人：重力多体系用 Tree Code の並列 GPU 化, 情報処理学会研究報告, Vol. 2012-HPC-135, No. 40, 1 (2012).
[15] Harfst, S., Gualandris, A. Merritt, D., Spurzem, R., Portegies Zwart, S. and Berczik, P.: Performance analysis of direct N-body algorithms on special-purpose supercomputers, New Astronomy, 12, 357 (2007).
[16] Portegies Zwart, S. F., Belleman, R. G. and Geldof, P. M.: High-performance direct gravitational N-body simulations on graphics processing units, New Astronomy, 12, 641 (2007).
[17] Belleman, R. G., Bédorf, J. and Portegies Zwart, S. F.: High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA, New Astronomy, 13, 103 (2008).
[18] Gaburov, E. and Harfst, S. and Portegies Zwart, S.: SAPPORO: A way to turn your graphics cards into a GRAPE-6, New Astronomy, 14, 630 (2009).
[19] NVIDIA Corporation: NVIDIA CUDA C Programming Guide Version 4.1, (2011).
[20] HA-PACS プロジェクト，入手先 ⟨http://www.ccs.tsukuba.ac.jp/CCS/research/project/ha-pacs⟩ .