5 アクセラレータによる粒子法シミュレーションの加速

成見 哲（慶應義塾大学理工学部） 濱田 剛（長崎大学工学部）
小西史一（東京工業大学グローバル COE）

はじめに

粒子法は重力多体問題，分子動力学シミュレーション，渦法，境界要素法など多くの分野で使われている計算手法です．最も計算が重い部分が比較的単純でメモリアクセスが少ないなどの理由から，これまでも数値アクセラレータによって加速した例が多くあります．重力多体問題では GRAPE[1]，分子動力学シミュレーションでは MDGRAPE[2] や MDEngine[3]，渦法・境界要素法では MDGRAPE が使われた例があります．最近は GPU，PS3, ClearSpeed などのアクセラレータも使われるようになっています．

GPU や PS3 の最大の特徴は，低価格ながら PC の CPU よりも速いピーク速度を持つことです．その性能を粒子法によるシミュレーションに利用した場合，高い価格性能比を達成できる可能性があります．ただし，これらのコンピュータを使いこなすにはそれなりの労力が必要となります．ここではいくつかのアクセラレータを重力多体問題や分子動力学シミュレーションで使用したときの性能を比較します．また，GPU で重力多体問題を加速する際のチューニング技法，GPGPU を手軽に始める方法などについても解説します．

GPU や PS3 の概要はこれまでに述べられていると思われますので，以下では MDGRAPE に関し若干説明しておきます．GRAPE はもともと重力多体問題用に約 20 年前に開発された計算機です．図 -1 に示すように，ホストとなる汎用の計算機に，専用の計算機を接続して使います．専用計算機側では粒子に働く力の計算のみ行い，その他の計算はホストコンピュータで行います．このように分担することで，専用計算機は単純な機能だけでよくなります．この結果高速なハードウェアを比較的簡単に作ることができるのです．MDGRAPE は分子動力学シミュレーション用にいくつか拡張がなされています．最大の特徴は粒子間のポテンシャルの形を変えられることです．重力だけでなく，クーロン力，分子間力など次の式で表されるような中心力を扱えます．

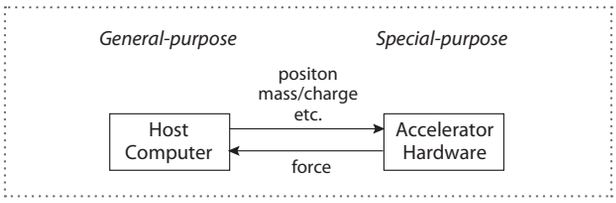

図 -1 GRAPE の基本アーキテクチャ．アクセラレータでは 2 体力の計算だけ高速に行う．ホストコンピュータではその他すべての計算を行う．

$$\vec{F}_i = \sum_{j=1}^{N} A_{ij} g(B_{ij} r_{ij}^2) \vec{r}_{ij} \qquad (1)$$

ここで $\vec{r}_{ij}$ は粒子 i と j の相対位置ベクトル，A_{ij}, B_{ij} は粒子ごとに違うパラメータ，$g(x)$ はポテンシャルの形を表す関数，N は系内の粒子数です．r_{ij}^2 は以下の式で表されます．

$$r_{ij}^2 = \left|\vec{r}_{ij}\right|^2 + \eta^2$$

ここで，η は粒子が近づいて力が発散することを防ぐためのものでソフトニングパラメータと呼ばれます．$g(x)$ はチップ内で 1,024 区間の 4 次補間を行うため，単精度相当の計算精度で任意の関数を与えられます．力を足し合わせる部分は 80-bit の固定小数点を使用しているため，足し合わせによって桁落ちすることは通常ありません．GPU や PS3 を使用する場合でも精度に関して同様の注意が必要です．

GPU, PS3, MDGRAPE-3 の性能

GPU，PS3，MDGRAPE-3 を使ったときの性能について比較します．はじめに重力多体問題での計算速度，次に AMBER を用いた分子動力学シミュレーションでの計算速度を示します．

▶重力多体問題での計算速度

図 -2 は重力多体問題の計算に使用した場合の計算速度を示しています．ここでは次の 4 つのプラットフォームを比較しています．

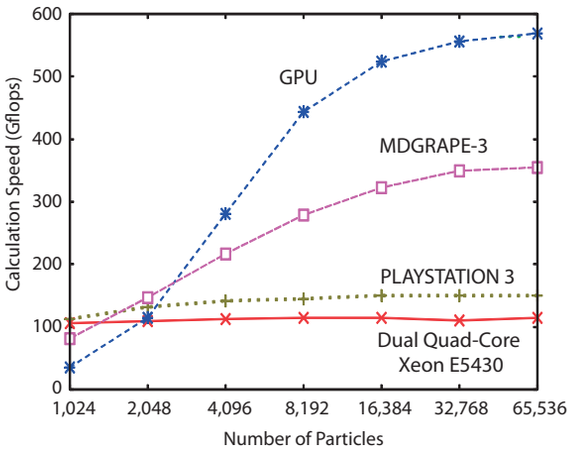

図 -2　重力多体問題での計算速度．GPU は粒子数が少ないときに計算速度が急激に遅くなるが PS3 はそうならない．最大速度は GPU が最もよい．2 粒子間に働く重力計算の演算を 38 演算相当として計算速度を算出している．数値は文献 5) のもの．

1. Dual Quad-Core Xeon E5430 (CPU)
2. PLAYSTATION 3 (PS3)
3. GeForce9800GTX (GPU)
4. MDGRAPE-3 PCI-X (MD3)

グラフから分かるように，GPU の場合粒子数が増えるに従って性能が向上しているのが分かります．PS3 や CPU では粒子数によらず性能が一定しています．GPU では必要とする並列度が数千程度と大きいため，粒子数が少ない場合に休む演算器が多くなって効率が低下するためです．本稿では CPU での計算において 8 コアすべてを使用しています．また，重力多体問題の CPU 用コードには似鳥氏によるアセンブラでチューニングしたコードを使用しています [4]．

重力多体問題の中でも銀河団などの宇宙の大規模な構造を対象とするような無衝突系と呼ばれる系では，式 (1) で，$A_{ij}=m_j$, $B_{ij}=1$, $g(x)=x^{-3/2}$ とすることで各粒子に働く重力を計算します．ここで m_j は粒子 j の質量です．クーロン力と同じ形ですが，ソフトニングパラメータを用いることが違います．重力は引力だけのため 2 つの粒子がかなり近づいて大きな力がかかることがあります．無衝突系においてはもともと 1 つの粒子は複数の星や銀河などによる密度分布を代表する点であり，ある程度の広がりを持ちます．このため，粒子が無限に近付いて力が発散することを避けるために，このソフトニングパラメータをある小さな値に取ります．このためにクーロン力の計算では自分自身の粒子との相互作用を排除する計算が必要になりますが，重力計算では後から簡単に差し引けるので力の計算ループの中ではこの排除を行いません．この違いによって無衝突系を扱う重力計算の方が効率を得やすい傾向にあります．ただし実用的な規模の計算を行う場合は複雑な近似アルゴリズムを用いる必要があるので，GPU への実装には技術的なハードルが非常に高くなる傾向にあります．

表 -1 は，65,536 粒子の場合の計算速度において 4 種類のハードウェアをいくつかの側面で比較しています．実効速度 (Effective speed)，コストパフォーマンス (Cost/speed)，電力パフォーマンス (Power/speed)，サイズパフォーマンス (Size/speed) です．4 つのすべての面で GPU の性能が最も高いことが分かります．ここで用いたアルゴリズムは重力多体問題の中でも比較的単純な計算式に限定していますが，ここでの性能はその計算機を使った科学技術計算の性能の上限値（これ以上の性能は期待できないという値．言い方を変えると，問題が GPU に適していればこのぐらい素晴らしい性能が得られる，という値）を示しています．

▶タンパク質の分子動力学シミュレーションでの計算速度

図 -3 は分子動力学シミュレーションの計算時間を比較しています．AMBER8 を用いてタンパク質 ScytaloneDehydratase（2,715 原子）と異なる数の TIP3P 水分子の系を，孤立系でカットオフなし，*NVT*（粒子数，

System	Effective speed (Peak speed) (Gflops)	Cost/speed ($/Gflops)	Power/speed (Watt/Gflops)	Size/speed (liter/Gflops)
Dual Quad-Core Xeon E5430	115 (170)[1)]	21.0	3.7	0.39
PLAYSTATION 3	157 (179)[2)]	2.8	1.3	0.06
GeForce 9800GTX	569 (432)[3)]	2.6	0.5	0.05
MDGRAPE-3 PCI-X	355 (380)	32.8	0.7	0.07

Best　Second best

1) Nitadori, K. (http://grape.mtk.nao.ac.jp/~nitadori/phantom/).
2) Narumi, T., Kameoka, S., Taiji, M. and Yasuoka, K. : *SIAM J. Sci. Comp.*, **30**, pp.3108-3125 (2008).
3) Narumi, T., Sakamaki, R., Kameoka, S. and Yasuoka, K. : Proceedings of PDCAT'08, pp.143-150 (2008).

表 -1　重力多体問題での計算速度．コストパフォーマンス，電力パフォーマンス，サイズパフォーマンスの比較．GPU がすべてに勝る．GPU の速度がピーク速度を超えているのは，1 相互作用 38 演算という換算が GPU においては演算数を多く見積もっていることになるため．

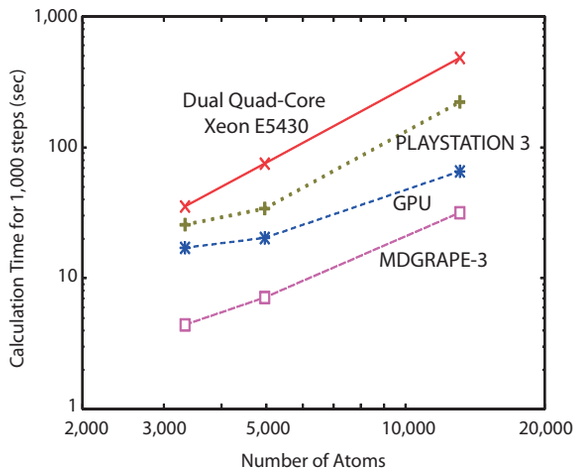

図 -3　分子動力学シミュレーションの計算時間．横軸は系内の原子数，縦軸は 1,000 ステップにかかる計算時間．GPU と PS3 は粒子数が少ないときにパフォーマンスが落ちている．数値は過去の文献 5) のもの．

体積，温度一定）アンサンブルで計算しています．カットオフとは，粒子間の相互作用計算を特定の粒子間距離以内の粒子同士のみに限定することで計算量を減らす手法です．GPU と PS3 では，粒子数が少ないときに計算効率が悪くなっています．GPU の場合は必要とする並列度が足りないためですが，PS3 の場合は汎用の PowerPC コアの計算速度が遅いため 2 体力（クーロン力や分子間力）以外の計算がボトルネックとなっています．表 -2 は，4,953 原子の系で計算速度以外に，コストパフォーマンスや電力パフォーマンスを比較しています．GPU はすべてにわたり 2 番目の性能となっています．PS3 はコストパフォーマンスでは最も優れています．

無衝突系の重力多体問題と分子動力学シミュレーションでの計算の違いの 1 つは，2 粒子間の力の積算に倍精度相当の精度が要求されるかどうかです．単純な分子であればすべて単精度でも大丈夫な場合もありますが，たんぱく質などの複雑な分子の場合は積算の精度を上げないと桁落ちにより精度が悪くなることが多いのです．ちなみに重力多体問題でも球状星団や銀河中心核などの非常に密度の高い天体現象を扱う系（衝突系と呼ばれる）ではやはり積算に倍精度相当の精度が要求されます．

タンパク質の分子動力学シミュレーションでは，結合力計算（2 体力以外の計算）もそれなりの時間を要します．GPU や PS3 で 2 体力の計算が速くなったために相対的に計算時間の割合が多くなるためです．特に PS3 の場合は PowerPC コアが遅いので，もう少し単純な分子の計算であれば若干計算速度が速いと期待されます．

ここではカットオフなしの計算で計算速度を比較しましたが，カットオフした場合や PME（Particle Mesh Ewald）による計算の場合の計算速度に関し興味がある読者の方もいると思います．残念ながら今回は詳細なデータはありませんが，GPU の場合おおざっぱにはシングルコアのホスト計算の 10 倍前後の計算速度というのが予想されます．たとえば NAMD のグループの GPU による結果 [6] でもそのくらいの速度です．カットオフなしの計算ではシングルコア CPU の数 10 倍の計算速度なのに対し PME やカットオフでは 10 倍程度にとどまるのは，GPU の場合カットオフ計算によるオーバヘッドが大きいのが理由です．もともと高い並列度を必要とするため，カットオフして計算量を減らそうとしてもなかなか効率よく計算量を減らせないという事情があります．

GPU のプログラミングと処理の最適化

実際に重力多体シミュレーションを GPU で加速する場合の GPU のプログラミング方法とその最適化について説明したいと思います．ここで用いるプログラミング方法は NVIDIA GPU が GeForce8800 以降から採用している CUDA に限定して説明することにします．CUDA では C＋＋ 言語を若干拡張した言語仕様を用いることになります．

重力多体シミュレーションでは粒子間に働く重力相互

System	Simulation time for 1,000 steps (sec)	Relative Acceleration	Acceleration / Cost (Million JPY^{-1})	Acceleration / Power (kW^{-1})
Dual Quad-Core Xeon E5430	75.1	1	3.3	2.5
PLAYSTATION 3	34.2	2.2	44	11
GeForce 9800GTX	20.2	3.7	19	12
MDGRAPE-3 PCI-X	7.1	10.6	7.6	42

Best　Second best

表 -2　分子動力学シミュレーションにおける計算速度，コストパフォーマンス，電力パフォーマンスの比較．GPU はすべてにおいて 2 番目の性能．

```
 1  __device__ float4
 2  inter(float4 xj,  float4 xi,  float4 apot)
 3  {
 4    float mj    = xj.w;         // Mass Mj
 5    float ieps2 = xi.w;         // epsilon^2
 6    float dx = xj.x - xi.x;     // Coordinates Xj - Xi
 7    float dy = xj.y - xi.y;     // Coordinates Yj - Yi
 8    float dz = xj.z - xi.z;     // Coordinates Zj - Zi
 9    float r2 = dx*dx + dy*dy + dz*dz + ieps2;
10    float r1i = 1/sqrt(r2);
11    float r2i = r1i * r1i;
12    float mr3i = mj * r2i * r1i;
13    apot.x += dx * mr3i;        // Accel AXi
14    apot.y += dy * mr3i;        // Accel AYi
15    apot.z += dz * mr3i;        // Accel AZi
16    return (apot);
17  }

18  __global__ void
19  kernel(float4* g_xj,
20         float* g_xi,
21         float* g_fi,
22         int ni,
23         int nj)
24  {
25    int i = blockIdx.x*blockDim.x+threadIdx.x;
26    float4 ai = make_float4(0.0, 0.0, 0.0, 0.0);
27    float4 xi;
28    xi.x = g_xi[i];        // Coordinates Xi
29    xi.y = g_xi[i+ni];     // Coordinates Yi
30    xi.z = g_xi[i+ni*2];   // Coordinates Zi
31    xi.w = g_xi[i+ni*3];   // epsilon ^ 2
32    for(int j = 0; j<nj; j++)  ai =  inter (g_xj[j], xi, ai);
33    if(i<ni){
34      g_fi[i]      = ai.x;
35      g_fi[i+ni]   = ai.y;
36      g_fi[i+ni*2] = ai.z;
37    }
38  }
```

図 -4　CUDA で重力加速度を求めるコード（最適化なし）

作用を加速することが最も重要になるというのは先ほど説明した通りです．そのため，GPU でシミュレーション全体を加速するためには重力相互作用をいかに高速に計算させるかを考えることになります．

▶ GPU プログラム作成の最初の一歩

まず最初に最も簡単なプログラム例（図 -4）を説明し，順を追ってプログラムを変形させながら最適化の方法を説明していきたいと思います．

図 -4 は重力相互作用を計算するプログラムです．これはほとんど最適化を考慮することなしに記述した CUDA プログラムの例です．

図 -4 のプログラムは，大きく分けて 2 つのサブルーチンから構成されます．18 行目から 38 行目まではカーネルと呼ばれる特殊なサブルーチンを表しています．これはサブルーチンの最初に `__global__` という CUDA 予約語を付けることで区別します．この `__global__` で区別したサブルーチンは GPU 上で動作するプログラムのメイン関数に相当するものです．カーネルはホスト PC から起動されて，GPU 上で動作を開始します．また 1 行目から 17 行目まではカーネルから呼び出されるサブルーチンを表します．これはサブルーチンの最初に `__device__` という CUDA の予約語を付けて区別します．`__device__` で区別されたサブルーチンは，GPU で動作中のサブルーチンから起動されます．ちなみに，`__device__` ルーチンはユーザがカーネルルーチンを読み書きしやすくするために使うもので，必要がなければカーネルルーチン 1 つだけで CUDA プログラムを記述してもまったく構いません．また図 -4 の 1, 2, 19, 26, 27 行目に現れる `float4` という型は 4 個の `float` 型変数を持った CUDA 独自の変数を宣言するためのものです．`float4` の各要素へのアクセスには，図 -4 の 4 ～ 8 行目のように，`x`，`y`，`z`，`w` を使ってアクセスします．`float4` もプログラムを読み書きしやすくするために使用しています．

図 -4 で示したプログラムの処理内容の説明に進みます．カーネルルーチンではホスト PC から GPU ボード上のメモリに送られた粒子データ `g_xj`, `g_xi` を読み出し（図 -4 の 28 ～ 32 行），それらを使って重力加速度 `ai` を計算します（図 -4 の 1 ～ 17 行）．求めた重力加速度 `ai`

はGPUボードメモリにある配列g_fiに書き戻します（図-4の34～36行）．配列g_fiの内容はホストPCへ転送され，時間積分等のホストPCにおける次の処理に使われます．サブルーチンinterはカーネルルーチンの32行目で呼ばれていますがこれは2粒子間に働く重力加速度を求める部分のみを専門に担当しています．

CUDAプログラム理解へ重要なポイント ― カーネルは多数のスレッドで並列実行

ここで図-4のCUDAプログラムを理解する上での非常に重要なポイントを説明します．それはカーネルルーチンはGPU上で同時に起動する多くのスレッドという処理単位で実行される，ということです．別々のスレッドでまったく同じカーネルルーチンが処理されるというのは一見無意味なことをしているように思われるでしょう．しかしながら，個々のスレッドで一意に決められたスレッド番号・ブロック番号というものを用いることでそれぞれのカーネルの動作やカーネルが担当するデータを区別することができるため問題はありません．ここでは25行目に現れるblockIdx.xとthreadIdx.xという特殊変数がスレッドごとに一意に決まるスレッド番号・ブロック番号をそれぞれ保持しています． またblockDim.xは各ブロックに含まれるスレッドの総数を表す特殊変数です．たとえば最初のブロックの最初のスレッドの場合，それぞれ0と0が自動的に割り当てられ，25行目のiの値は0になります．blockDim.x，すなわち各ブロックに含まれるスレッドの数が128であるとすると，2番目のブロックの2番目のスレッドの場合，blockIdx.xには1，threadIdx.xには1が自動的に割り当てられ，25行目のiは129になります．このように多数のスレッドで同じカーネルルーチンが実行されますが，スレッドで一意に決まるスレッド識別番号iを用いることで，処理の内容や処理するデータを区別することが可能になります．計算効率を考えなければ，スレッド識別番号とif文などの条件分岐を用いることで，たとえば数千以上のスレッドすべてが別々の動作を行うようなプログラムを作成することも可能です．

メモリレイテンシの最適化 ― コアレスアクセス

これまで図-4のCUDAプログラムはまったく最適化していませんと説明しましたが，後のコードの説明を簡単にするために，実は1つだけ最適化を行っている部分があります．それは図-4の28行目から31行目および34行目から36行目です．ここではコアレスアクセス（coalesce access）というメモリの読み書きを高速にするためのテクニックを用いています．

コアレスアクセスとはベクトル計算機などで用いられるメモリアクセスのレイテンシを短縮するための一般的な技法です．GPUはデータの読み書きにDRAM（Dynamic RAM）と呼ばれるメモリを用います．このDRAMはデータの読み書きを開始する前に外部から要求命令（コマンド）を必ず受け取らなければなりません．DRAMはコマンドを受け取ってからしばらく後にはじめてデータの読み書きを開始します．この間隔のことをDRAMのレイテンシと呼びます．

読み書きするデータのアドレスが連続だった場合には，DRAMは1回のコマンドで複数のデータを連続して読み書きすることができます．一方読み書きするデータが連続した順番となっていない場合には，つまり読み書きするデータのアドレスが飛び飛びになっている場合には，その都度コマンドを受け付ける必要が発生し，レイテンシが大きくなります．結果として連続アクセスの場合よりもメモリの読み書き速度が大変遅くなってしまいます．

このDRAMのレイテンシの短縮を考慮したプログラミング方法がコアレスアクセスと呼ばれるものです．図-4の28行目はほんの1行の記述ですが，実際には数百～数千のスレッドで同時に実行されます．スレッド識別番号がi＝kのスレッドがアクセスするアドレスはg_xi［k］となります．そのとき，隣のスレッドはg_xi［k+1］を読もうとしています．また，その隣のスレッドはg_xi［k+2］，その隣はg_xi［k+3］，といったように，たった1行だけの記述ですが，すべてのスレッドについて1つ1つ動作を考えると，多数のスレッドによって連続したアドレスをDRAMに要求していることが理解できるかと思います．

このようなコアレスアクセスを用いることでDRAM（NVIDIA GPUの場合はデバイスメモリと呼びます）に連続したデータを要求でき，メモリの読み書き速度を向上させることが可能です．そのほかにもGPUで使われるGDDRと呼ばれる特殊なDRAMは16バイト単位の読み書きが高速であるなどの特徴を使ってさらなる最適化をすることも可能ですが，今回の説明では省略します．

ホストPCからのカーネルルーチンの実行―カーネルコール

これまでにも何度かふれましたが，カーネルルーチンはホストPCが直接呼び出します．この呼び出しのことをカーネルコールと呼びます．ホストPCからは次のような記述でカーネルコールを行います．

```
ni = 32768;
kernel<<< ni/128, 128>>>(d_xj, d_xi, d_fo,
ni, nj);
```

一般的なC＋＋言語のサブルーチン呼び出し方法とは異なり，サブルーチンと引数リストの間に3重ブラケットで囲まれた記述を行う必要があります．この3重ブラケットで囲まれた部分では，カーネルルーチンをGPUにいくつのスレッドを用いて処理させるかを指示することになります．この例ではブロック数が256個，各ブロック当たりのスレッド数が128個，つまり合計256×128 ＝ 32768個のスレッドを起動してカーネルルーチ

ンの処理を並列に行うことを意味しています．以降で説明する最適化を施したカーネルルーチンでも同様に上記と同じカーネルコールを用いることとします．

ここでGPUのアーキテクチャについても簡単にふれておきます．G80（GeForce8800シリーズ）やG92（GeForce8800シリーズの一部とGeForce9800シリーズ）世代のGPUでは，物理的には16個のマルチプロセッサと呼ばれる，Intel CPUなどでコアと呼んでいるものと同等の電子回路で構成されています．個々のマルチプロセッサの内部にはストリームプロセッサと呼ばれるスカラプロセッサがそれぞれ8個ずつ内蔵されています．ストリームプロセッサについては，Intel Core2 CPUでのSSE，もしくはIBM Power CPUでのAltiVecとよく似たベクトル演算ユニットだと考えていただくと分かりやすいかと思います．GPU全体では合計128個のストリームプロセッサを使ってたくさんのスレッドを処理します．

カーネルコールの3重ブラケットで指定したブロック数とスレッド数は，それぞれマルチプロセッサ，ストリームプロセッサをどれだけ使うかということを示しています．物理的にはマルチプロセッサは16個，その中にストリームプロセッサは8個しかありませんが，それぞれ256個，128個というような実際のハードウェアの数よりも大きな値を指定できるのは，マルチプロセッサ，ストリームプロセッサそれぞれがSMT（Simultaneous Multithreading）アーキテクチャと呼ばれるハードウェア実装方式を用いているためです．

SMTとは複数のスレッドを単一のプロセッサで並列に実行するための技術です．並列実行と言っても物理的には1つのハードウェアを使いまわすことになり，複数のスレッドは時間をずらしながら単一のハードウェア(ここでは浮動小数点演算器)を交互に使うことになります．

Warpとは何か

多少細かな説明になりますが，現在のNVIDIA GPUアーキテクチャは，マルチプロセッサ内の8個のストリームプロセッサが単一の命令供給ユニットから送られてくる同一の命令を受け取る設計になっています．このようなハードウェア設計様式をSIMD fashionと呼びます．また，ストリームプロセッサは命令供給ユニットの4倍の速度で動作する設計になっています．つまり4クロックのあいだ8個のストリームプロセッサは同じ命令を読み続けることになります．そのため，論理的に同時実行するスレッドの最小単位は8×4＝32個となります．この同時実行するスレッドの最小単位のことをCUDAマニュアルではワープ（Warp）と呼んでいます．ハードウェアは1ワープ32スレッドを最小単位として動作する設計になっていますので，カーネルコールで指定するスレッド数（3重ブラケット間の右側の数字）は32の倍数が最も効率良く動作する値であることが分かります．マニュアルを読んだだけではWarpが意味するところを理解しづらいと思いましたので簡単に説明しておきました．

▶シェアードメモリを使った最適化法

多少細かな説明もありましたが大雑把な理解として，1つのカーネルルーチンを非常にたくさんのスレッドで処理する，ということさえ理解できればCUDAプログラミングは難しいという意識がだいぶ薄れてくると思います．それでは次はどのようにプログラムを高速化すればよいかという最適化の話題について説明を進めていきたいと思います．

最適化の話題のまず最初として図-4のCUDAプログラムをシェアードメモリ（Shared Memory）と呼ばれる高速なオンチップメモリを用いることで高速化してみたいと思います．デバイスメモリよりも高速なシェアードメモリを使うことによる最適化はCUDAプログラムの最適化の中でも最も優先順位が高い方法です．つまりプログラミングの労力に見合った高速化が最も得られやすい方法なので第一に説明したいと思います．

ちなみに，シェアードメモリへのアクセスはG80以降のUnified Shaderアーキテクチャを採用したGPUとCUDAプログラミング環境がともに2006年11月（CUDAのLinux対応は2007年2月）に登場して初めて利用可能になった機能です．それ以前のGPUと，CgやHLSLなどのシェーダ言語と呼ばれる古いプログラミング手法ではシェアードメモリを効率良く利用することができませんでした．2007年以降GPGPUの応用研究がそれまで以前と比べて急速に拡大してきた背景には，このUnified ShaderアーキテクチャとCUDAによるシェアードメモリへのアクセス機能が一般のユーザに開放されたことが最も大きな要因となっています．

手前みそで恐縮ですが，私たちは2007年3月にCUDAがリリースされた直後，このシェアードメモリを利用することがN体計算を高速化する最も重要な方法であることを提案しChamomile Scheme[9]という名前をつけました．現在ではごく当たり前の最適化手法として広く認知されていますがパイオニアの特権ということでお許しください．

図-5は図-4のカーネルをシェアードメモリを使って最適化したプログラム例です．シェアードメモリとはマルチプロセッサごとに16KBずつ実装されているローカルメモリ（Cell.BEなどではローカルストアと呼ばれているもの）のことです．シェアードメモリはその名のとおりスレッド間でデータを共有することが可能です．ただし，ブロックをまたがったデータの共有はできません． 図-5の22行目で`float4`型の128個の配列`s_xj`をシェアードメモリとして宣言しています．

図-5の25行目に注目してみましょう．この行ではデバイスメモリからシェアードメモリへデータを移動して

```
 1  __device__ float4
 2  inter(float4 xj, float4 xi, float4 apot)
 3  {
 4    /* ここは図-4と同じ */
 5  }
 6  #define NTHRE (128) // blockDim.xと同値
 7  __global__ void
 8  kernel(float4* g_xj,
 9         float* g_xi,
10         float* g_fi,
11         int ni,
12         int nj)
13  {
14    int tid      = threadIdx.x;
15    int i = blockIdx.x*NTHRE+tid;
16    float4 ai = make_float4(0.0, 0.0, 0.0, 0.0);
17    float4 xi;
18    xi.x = g_xi[i];
19    xi.y = g_xi[i+ni];
20    xi.z = g_xi[i+ni*2];
21    xi.w = g_xi[i+ni*3];
22    __shared__ float4 s_xj[NTHRE];
23    for(int j = 0; j<nj; j+=NTHRE){
24      __syncthreads();
25      s_xj[tid] = g_xj[j+tid];
26      __syncthreads();
27      for(int js = 0; js<NTHRE; js++) ai = inter(s_xj[js], xi, ai);
28    }
29    if(i<ni){
30      g_fi[i]      = ai.x;
31      g_fi[i+ni]   = ai.y;
32      g_fi[i+ni*2] = ai.z;
33    }
34  }
```

図-5　CUDAで重力加速度を求めるコード（シェアードメモリを使った例）

いることが分かります．ここでは先ほど説明したコアレスアクセスのテクニックをデバイスメモリからの読み出しとシェアードメモリへの書き込みの両方に利用していることが理解できるかと思います．この1行の記述のみで，128個のスレッドがデバイスメモリからシェアードメモリへデータを128個分移動し終えたことになります．

シェアードメモリに保存したデータはカーネル内で何度も再利用することでその効果を発揮します．図-5の重力加速度を求めるプログラムでは，一度用意したシェアードメモリのデータ s_xj をすべてのスレッドが何度も利用することになります．図-5の27行目がその様子を示している部分です．ここでは全スレッドがシェアードメモリに蓄えたデータを128回のループのあいだ再利用し続けていることが分かります．

このように何度も利用するデータは，その都度デバイスメモリから移動してくるのではなく，なるべくシェアードメモリに入れたままにしておく，もしくはそうなるように大幅にアルゴリズムの修正を行うことが高速化のポイントになります．

▶ループアンローリングを使った最適化法

最後にループアンローリングという最適化手法を紹介したいと思います．これは先ほどのシェアードメモリを使った最適化に次いで重要な最適化手法の1つです．

図-6のCUDAプログラムは図-5のプログラムにループアンローリングを適用した例です．ループアンローリングとはforループのすべての処理を展開することでforループを消してしまう方法です．図-6の27行目から33行目までがループアンローリングを適用した部分です．この例では図-5の27行目のループ処理をすべて展開しています．図-6の32行目はコメントで省略してありますが，実際のプログラムでは s_xj[5] から s_xj[126] までのループアンローリング（122行分）を行っています．

図-5の27行目のようなforループ処理ではループが回るたびに，ループカウンタ js の値の判断（js<NTHRE），およびインクリメント（js+=NTHRE）という処理が必要になりますが，図-6のようにすべてのループを展開することでこれらの処理が必要なくなります．G92世代のGPUでは，ループ処理（空ループでさえ）を入れるだけで実行サイクルが大幅に増える性質を持っているため，ループアンローリングという手法はとても効果的な最適化手法になります．

▶その他の最適化手法やその適用範囲など

今回説明した図-4, 5, 6のカーネルの性能の違いを比

```
 1  __device__ float4
 2  inter(float4 xj, float4 xi, float4 apot)
 3  {
 4    /* ここは図-4,5と同じ */
 5  }
 6  #define NTHRE (128) // blockDim.xと同値
 7  __global__ void
 8  kernel(float4* g_xj,
 9         float* g_xi,
10         float* g_fi,
11         int ni,
12         int nj)
13  {
14    int tid      = threadIdx.x;
15    int i = blockIdx.x*NTHRE+tid;
16    float4 ai = make_float4(0.0, 0.0, 0.0, 0.0);
17    float4 xi;
18    xi.x = g_xi[i];
19    xi.y = g_xi[i+ni];
20    xi.z = g_xi[i+ni*2];
21    xi.w = g_xi[i+ni*3];
22    __shared__ float4 s_xj[NTHRE];
23    for(int j = 0; j<nj; j+=NTHRE){
24      __syncthreads();
25      s_xj[tid] = g_xj[j+tid];
26      __syncthreads();
27      ai = inter(s_xj[0], xi, ai);
28      ai = inter(s_xj[1], xi, ai);
29      ai = inter(s_xj[2], xi, ai);
30      ai = inter(s_xj[3], xi, ai);
31      ai = inter(s_xj[4], xi, ai);
32      //    .... 同様 ...
33      ai = inter(s_xj[127], xi, ai);
34    }
35    if(i<ni){
36      g_fi[i]      = ai.x;
37      g_fi[i+ni]   = ai.y;
38      g_fi[i+ni*2] = ai.z;
39    }
40  }
```

図-6　CUDAで重力加速度を求めるコード（ループアンローリングを適用した例）

較できるように，あまり実用的ではありませんが共有時間刻み法というシンプルな積分法を用いた条件で性能を測定しました（図-7）．ほとんど最適化をしていない図-4のカーネルを図-5のようなシェアードメモリを使うように変更するだけでも性能が大幅に向上していることが分かります．図-7では今回説明したカーネルのほかに図-6をさらに最適化したカーネルの性能も掲載しておきました．余力のある方はさらなる最適化にチャレンジしてみてはいかがでしょうか．

今回は紙面の都合上，重力加速度を求める際に重要な最適化手法を選んで紹介しました．これらの最適化手法はGPGPUが広く研究されている現在ではごく一般的な手法でもあります．現在私たちはこれらの基本的な手法のほかにも，数々の最適化手法を開発しています．たとえば，重力加速度を求める場合に高精度な計算結果が必要になる場合があります．このような状況では今回紹介したカーネルでは計算が破綻してしまいます．倍精度演算ハードウェアが使えれば良いのですがGPUの倍精度性能は4コアのCore2やOpteronと比べても大差がありません．そこで単精度演算ハードウェアを上手に使いまわして倍精度相当の演算を行うことが重要になります．このときいかに少ない単精度演算回数で倍精度演算を実現するかについての興味深い方法[10)～13)]もあります．またBarnes-Hut Treecodeという無衝突系重力多体シミュレーションで用いる高速アルゴリズムがあるのですが，今回紹介したGPUカーネルをそのまま適用しても期待した通りの性能は出せません．その性能低下の理屈や克服方法，当事者としては非常に辛い経験でしたが，性能を向上させるまでの道のりなども非常に興味深い内容です[14), 15)]．このようにGPUを用いた重力多体シミュレーションひとつに注目しても本稿だけでは説明しきれないほどの実装法のバリエーションが存在します．これら多数の興味深い内容については機会を改めて紹介できればと思っています．

GPGPUを簡単に始めてみる

私たちはKNOPPIX for CUDAと呼ばれるGPGPUのお試し用CDROMを配っています（http://www.yasuoka.mech.keio.ac.jp/cuda/ または，http://www.bi.cs.titech.ac.jp/~konishi）．これはGPUを使用するためのソフトウェア群がすでに入っているブート可能なCDROMです．つまり，GPUを搭載したPCに挿入してこのCDROMからブートするだけでGPUを使った高速な粒子法シミュレーションを体験することができます．プログラムの開発環境などもインストールされているため，簡単なプログラムを自分で開発することもできます．以下ではKNOPPIX for CUDAを使用して分子動力学シミュレーションをGPUで加速してみます．

▶動作環境と起動方法

KNOPPIX for CUDAを動作させるためには以下の条件を満たしたハードウェアが必要です．

1. x86_64 CPU，1 Gbyte程度のメモリ
2. CUDAをサポートするGPU[7)]

起動方法

1. 起動直後にKNOPPIX for CUDAのディスクを挿入
2. BIOSの設定でCD/DVDからの起動が他より優先している場合はそのまま起動するが，そうでない場合はBIOSの設定を行う
3. 途中でEnterキーを押すことが求められるのでEnterキーを押す（図-8(a)）
4. KNOPPIXが立ち上がったら画面下部のタスクバーより起動するデモソフトを選択

▶デモプログラムの実行とコンパイル

KNOPPIX for CUDA には，重力多体問題，分子動力学シミュレーション，渦輪のシミュレーション，津波のシミュレーションなどが含まれています．ここでは NaCl の分子動力学シミュレーションプログラム claret について解説します．当プログラムは福井大学の古石氏によって開発されたものです．

Claret を実行するには，起動後のデスクトップ画面（図 -8(b)）の下にあるバーから図 -8(c) に示すアイコンをクリックします．表示される Gflops 値は 1 つのペア相互作用計算を 78 演算として計算しています．ポテンシャルは Tosh-Fumi ポテンシャルを使用しています．キー操作は古石氏の Web ページ [8] に書いています．温度を上下させてリアルタイムに融解・凝固を体験できます．

プログラムをコンパイルするには，以下のようにします．まず，デスクトップ画面の一番下にあるバーから，図 -8(d) に示すアイコンをクリックしてターミナルプログラムを起動します．次に以下に示すコマンドを入力します．

```
> su - root
> cd /usr/local/claret
> make clean
> make
```

再度 claret のアイコンをクリックしてデモプログラムを起動します．

次に CPU のみで同様の計算を行ってみましょう．ソースファイルは /usr/local/claret/mr3_host.c です．24 行目では OpenMP を使ってマルチコア CPU を使うようにしています．以下のようにコンパイルしてから再度アイコンをクリックすると，CPU で計算して表示します．Intel Core2 Duo 2.5GHz（Dual core）では，1.2 Gflops 程度出ていました．

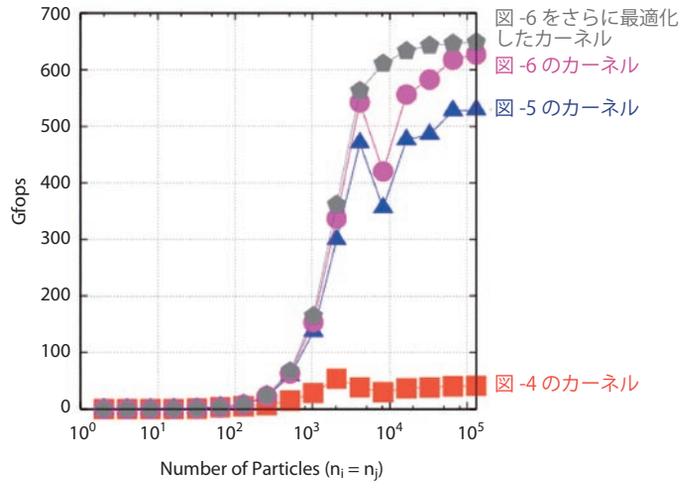

図 -7　重力体問題での計算速度．横軸は計算に用いた粒子数を，縦軸は計算速度（Giga floating operation per second : Gflops）を表します．図 -4 ～ 6，および図 -6 をさらに最適化した 4 種類のカーネルについての測定結果をプロットしています．2 粒子間に働く重力計算の演算数を一般的な 38 演算で換算してあります．GPU は ASUS EN8800GTS/512M/TOP を 1 台だけ使用しました．

```
> make -f Makefile_host clean
> make -f Makefile_host
```

さらに GPU を使って計算してみましょう．/usr/local/claret/mr3.cu を使います．コンパイルは以下のように行います．最初に gcc のバージョンを変えてコンパイルするためにリンクを張りなおします．これは gcc-4.2 では CUDA のコンパイルに失敗するからです．再度アイコンをクリックして実行します．

```
> ln -sf /usr/bin/gcc-4.1 /usr/bin/gcc
> make -f Makefile_gputest clean
> make -f Makefile_gputest
```

この GPU コードでは恐らく数 Gflops 程度しか出ません．しかしこれまで述べたようなチューニングを施すと 10 倍以上速くなります．

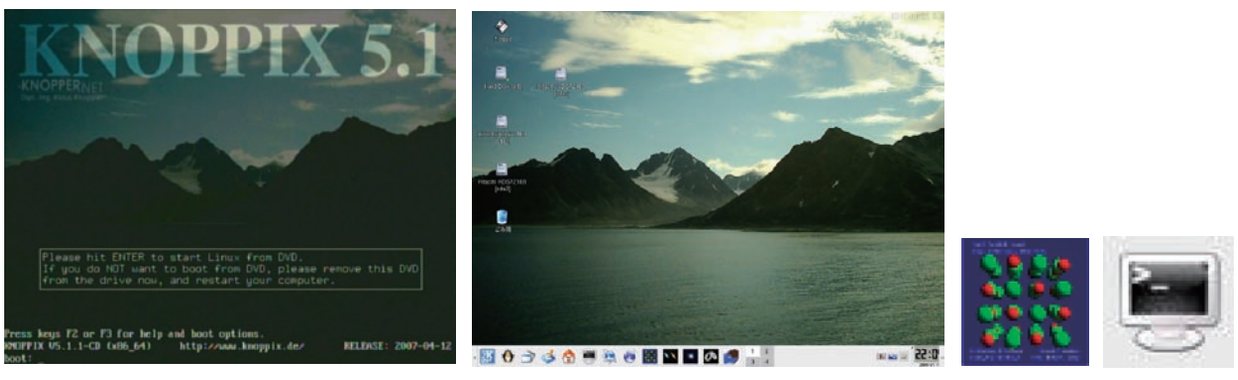

図 -8　左から (a) ブート画面，(b) デスクトップ画面，(c)claret デモプログラムのアイコン，(d) ターミナルプログラムのアイコン

関数名	機能
MR3init	アクセラレータを初期化する
MR3free	アクセラレータの終了処理を行う
MR3SetTable	2 粒子間の相互作用関数を変更する
MR3calccoulomb	粒子に働くクーロン力を計算する．Ewald 法の実空間の力も計算できる
MR3calcvdw	粒子に働く分子間力を計算する
MR3calcewald	Ewald 法の波数空間の力を計算する

表 -3　基本となる共通ライブラリ（MR3 ライブラリ）

```
average error log      =    5.80040365324832
average error log      =    6.51959649824525
average error log      =    6.49177280058796
average error log      =    6.38714237982568
average error log      =    5.91469907493689
average error log      =    6.48583370312286
average error log      =    6.45003564982928
```

図 -10　make check の実行結果の最初の部分

```
****************************************
 RESULTS of ENERGY at Lennard-Jones part
 ****************************************

    n     energy_host    energy_MDone e_host/e_MDone atom type
    1 -0.1146360E-02 -0.1146361E-02    0.9999998588  3
    2 -0.1881699E-03 -0.1881699E-03    0.9999998929  1
    3 -0.1133257E-02 -0.1133258E-02    0.9999996338  4
```

図 -9　sample_md3 の実行結果の最初の部分

▶共通ライブラリとサンプルプログラム

MDGRAPE 用にはこれまで開発されてきたライブラリ（MR3 ライブラリと呼ぶ）があります．GPU や PS3 でも同じ引数のラッパーを介することで MR3 ライブラリと共通の機能が使用可能になっています．このため，比較的単純な計算を行うだけであれば，この共通ライブラリを使用することで GPU や PS3 のプログラミングの手間を大幅に省くことができます．MDGRAPE-3 用のものは公開されていますが，GPU や PS3 用のものも今後公開予定です．

表 -3 に MR3 ライブラリ関数の一部をまとめています．MR3init はプログラムの最初で呼び出し，GPU や PS3 などのアクセラレータを初期化します．MR3free はプログラムの最後に呼び出し，アクセラレータの終了処理を行います．MR3calccoulomb は粒子間のクーロン力を，MR3calcvdw は分子間力を計算します．詳しくは /opt/MDGRAPE3/doc/user_manual.pdf を参照してください．

MR3 ライブラリのサンプルプログラム sample_md3.f（/usr/local/sample にある）は，慶應義塾大学の泰岡氏によって書かれたプログラムで，クーロン力，分子間力，Ewald 法の実空間の力，Ewald 法の波数空間の力を計算して CPU と精度を比較します．CPU でのコードも含まれているので MR3 ライブラリの計算式をコードを見ながら確認できます．sample_md3.f のコンパイルと実行は以下のように行います．前述の計算を 5 ステップ行います．図 -9 は実行結果の一部です．MDone または grape と書かれた個所が GPU での計算結果です．

```
> su - root
> cd /usr/local/sample
> make
> ./sample_md3
```

計算の精度を簡単にチェックするには，以下のコマンドを入力します．

```
> make check
```

CPU と GPU との相対計算精度が何桁あるか表示します．6 桁程度あっているはずです．図 -10 はこのとき表示される数値です．

おわりに

GPU や PS3 は低価格で魅力的なハードウェアですがプログラミングは普通の PC に比べると遥かに大変です．今回は特に GPU に注目して最適化手法を解説しました．では，そこまでして GPU で計算を高速化する必要があるのだろうか，数年待てば CPU もまたコア数を増やすので無駄な努力ではないか，という疑問を抱かれるのは自然なことだと思います．

私たちは GPU の実用化への取り組みと同時に，近年マルチコア化・ベクトル化へ進化しつつある CPU に関しても同様な取り組みを行っています（図 -11）．実は CPU でも程度の差こそありますが，並列度が増すにつれて，GPU と同じように並列化・最適化が難しくなります．将来は CPU も GPU と同じようにワンチップに強力なベクトル演算プロセッサを備えたコアがたくさん搭載されるでしょう．私たちは近い将来 GPU のような多数のプロセッサコアを備えた計算機しか入手できなくなるのではないかと考えています．このような時代のことを私たちはメニーコアコンピューティングの時代と呼

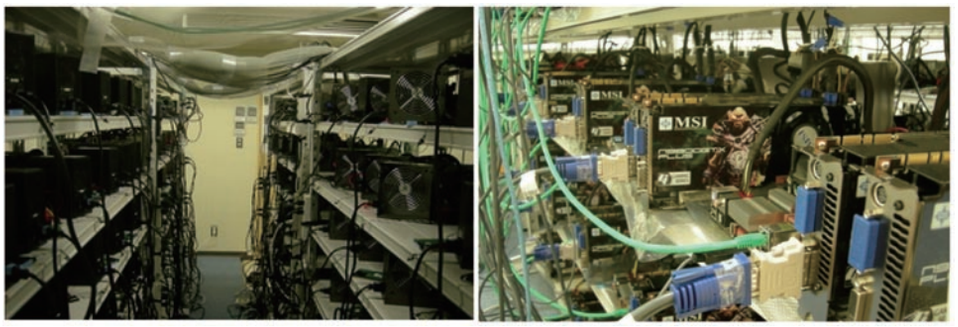

図-11　長崎大学 GPU クラスタ（長崎大学工学部）．256 台の GPU（GeForce 9800GTX+）をネットワークで接続したシステム．システム全体で単精度ピーク性能は 190Tflop/s に達する．一般的な PC クラスタと違い，各計算ノードに極端にマルチコア化・ベクトル化されたプロセッサを用いていることが特徴である．長崎大学では慶應義塾大学，東京工業大学，理化学研究所などと協力しながら，このような特殊な PC クラスタを実際のアプリケーションで効率良く利用するための方法を研究している．

んでいます．

私たちは，GPU を用いた高速化の経験から，メニーコアコンピューティングの時代には CPU でさえも GPU と同様に時間をかけて極端な並列化・最適化を行えなければ本来ハードウェアが持っている性能を引き出すことは非常に難しいだろうと感じています．実用レベルでの使用に応えられる自動並列化コンパイラの登場が待ち望まれますが，現在は人間が並列化作業を行うことを前提とした OpenMP や MPI などの作業効率の向上を目的にしたツール（CUDA もその 1 つ）がようやく整いつつある段階でしかありません．そのためプログラムのどこをどう並列化すればよいかについては，あくまで人間が考えなければならない仕事です．

現在の私たちが取り組んでいる GPU プログラミングの最適化・並列化の研究成果は，現在利用できる計算機環境で最高性能を得るという目的だけでなく，時代に先行して極端にベクトル化・マルチコア化されるであろう CPU を有効に活用するための準備を行うという目的があります．GPU を用いたさまざまな分野での並列化・最適化の研究成果は，将来のメニーコアコンピューティングの時代にこそ必要とされるのではないかと期待しています．

参考文献

1) Sugimoto, D., Chikada, Y., Makino, J., Ito, T., Ebisuzaki, T. and Umemura, M. : A Special Purpose Computer for Gravitational Many-body Problems, Nature, 345, pp.33-35 (1990).
2) Narumi, T., Ohno, Y., Okimoto, N., Koishi, T., Suenaga, A., Futatsugi, N., Yanai, R., Himeno, R., Fujikawa, S., Ikei, M. and Taiji, M. : A 55 TFLOPS Simulation of Amyloidforming Peptides from Yeast Prion Sup35 with the Specialpurpose Computer System MDGRAPE-3, In Proceedings of SC2006, Tampa, CD-ROM (2006).
3) Amisaki, T., Toyoda, S., Miyagawa, H. and Kitamura, K. : Development of Hardware Accelerator for Molecular Dynamics Simulations : A Computation Board that Calculates Nonbonded Interactions in Cooperation with Fast Multipole Method, ***J. Comput. Chem.***, 24, 5, pp.582-592 (2003).
4) http://grape.mtk.nao.ac.jp/~nitadori/phantom/
5) Narumi, T., Sakamaki, R., Kameoka, S. and Yasuoka, K. : Overheads in Accelerating Molecular Dynamics Simulations with GPUs, In Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'08), Dunedin, New Zealand, pp.143-150 (2008).
6) Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G. and Schulten, K., *J. Comput. Chem.*, 28, 16, pp.2618-2640 (2007).
7) http://www.nvidia.com/object/cuda_learn_products.html
8) http://atlas.riken.go.jp/~koishi/claret.html
9) Hamada, T. and Iitaka, T. : The Chamomile Scheme : An Optimizedalgorithm for N-body Simulations on Programmable Graphics Processing Units, ArXiv Astrophysics E-prints, Astro-ph/073100 (2007).
10) Nitadori, K. : High-accuracy N-body Simulations on GPU, Poster in AstroGPU 2007, Princeton (2007).
11) Takahashi, T. : Hardware Acceleration for Boundary Element Methods, Harvard/Riken GPGPU Symposium 2008, Cambridge (2008).
12) Narumi, T., Hamada, T., Nitadori, K., Sakamaki, R., Kameoka, S. and Yasuoka, K. : High-Performance Quasi Double-Precision Method using Single-Precision Hardware for Molecular Dynamics Simulations with GPUs, In Proceedings of HPC Asia 2009, Kaohsiung, Taiwan, in press.
13) Nitadori, K. : Doctor Thesis, Tokyo University (2009)
14) Hamada, T.: Internals of the CUNBODY-1 Library : Particle/force Decomposition and Reduction, Invited Talk in AstroGPU 2007, Princeton(2007).
15) Hamada, T., Nitadori, K., Masada, T. and Taiji, M. : 50.5 Mflops/dollar and 8.5 Tflops Cosmological N-body Simulation on a GPU Cluster, In Proceedings of the ACM/IEEE SC08 Conference for High Performance Computing, Networking, Strage and Analysis, Austin (2008).

（平成 20 年 12 月 30 日受付）

成見　哲（正会員）
narumi@a7.keio.jp

1998 年東大総合文化研究科博士課程修了．博士（学術）．同年より理化学研究所研究員，2007 年より慶應義塾大学特別研究講師．専門は高性能計算．2000 年および 2006 年に Gordon Bell 賞を受賞．

濱田　剛（正会員）
hamada@cis.nagasaki-u.ac.jp

2004 年東大総合文化研究科博士課程退学．博士（学術）．2006 年より理化学研究所基礎科学特別研究員，2008 年より長崎大学テニュアトラック助教．専門は FPGA や GPU の自動並列化コンパイラに関する研究．IPA 2005 年度上期天才プログラマー／スーパークリエータ認定．

小西史一（正会員）
konishi@is.titech.ac.jp

1998 年東京都立科学技術大学大学院博士課程退学．博士（工学）．1998 年同大助手，2000 年より理化学研究所研究員．2008 年より東京工業大学グローバル COE「計算世界観の深化と展開」特任准教授．専門は，大規模バイオインフォマティクス．