Vol.2013-HPC-138 No.19
2013/2/22

分子軌道計算の GPGPU 化に向けた行列加算手法の提案

梅田宏明[†1]　塙敏博[†1]　庄司光男[†1]　朴泰祐[†1]

分子軌道計算のボトルネックである G 行列計算について CUDA による GPGPU 化を行なった. G 行列計算においては行列への加算に排他制御が必要であることが最も大きな問題となっている．我々は行列加算について排他制御を行なわない G 行列計算手法を提案し，その実装を行なった．またスクリーニングや動的負荷分散，CPU 計算とのオーバーラップなど，多くの高速化の技法を実装することにより CPU 1 コアに対し 13 倍程度の性能を実現している．元の分子軌道計算プログラムの持つ OpenMP/MPI ハイブリッド並列を利用することで，複数 GPU を使った並列計算も実現した.

1. はじめに

近年の科学技術計算向き計算機の急速な発展に対し，量子化学計算アプリケーションは十分な対応が出来ているとは言えない状況にある．多くの量子化学研究者が応用目的で利用するアプリケーションは古く，また高機能であるため大きく複雑な傾向にありこれを最新の計算機アーキテクチャ向きに並列化するには多くの労力が必要になる．この状況を打開するため，よりシンプルで拡張性に富んだ新しい量子化学プログラム実装が必要になる．そのような新しいプログラムにおいて量子化学計算の基礎的な計算手法である Hartree-Fock(HF)計算を最近の HPC 向け並列計算機上で効果的に実行させることは，大規模計算だけでなく高精度計算を考える上でも重要である.

HPC 向け並列計算機のうち汎用 CPU を利用した並列計算機については OpenMP/MPI ハイブリッド並列による対応が進みつつあり，京コンピュータでは数万コアクラスの並列計算も実行可能になってきている．一方で GPGPU のようなアクセラレータを用いた高性能計算機については倍精度演算へのハードウェアサポートが十分でなかったことも影響してその対応が遅れている．中には単精度計算を効果的に利用することで非常に高速な計算を実現している量子化学計算プログラム実装もあるものの，商用ソフトウェアであったり[1]ソースが非公開である[2]など，今後の量子化学アプリケーション開発を進めていく上で十分なノウハウの共有ができない状態であった．我々は筑波大学計算科学研究センターに導入されている密結合並列演算加速機構実証システム「HA-PACS」[3]を生かした量子化学計算アプリケーションを開発するため、ソースが公開されている国産アプリケーションである OpenFMO[4][5]をターゲットアプリとし，これの HF 計算ルーチンの GPGPU 化を試みた.

OpenFMO は九州大学の稲富らが開発している超並列計算機向けフラグメント分子軌道（fragment molecular orbital, FMO）計算プログラムである．このプログラムは C 言語で記述されており，現状の CUDA 実装では扱いやすい．また MPI/OpenMP ハイブリッド並列による超並列計算機向けの並列化[6]だけでなく，RPC 化などによる対故障性の検討にも使われるなど次世代の量子化学計算アプリのプラットフォームとしても利用が期待されている. FMO 計算では巨大な分子を複数の比較的小さなフラグメントに分割することで計算量を減少させているが，このフラグメントおよびフラグメントペア一つ一つの計算において HF 計算が利用されている．また現在の OpenFMO では HF ベースの FMO 計算のみの実装であるが, 今後は MP2 や DFT 等より高精度・高機能の計算手法の実装が必要になると予想される．国内外の研究者に個々のフラグメント計算に対するそれら手法を実装してもらうことになるため，フラグメント単独で計算している部分を抽出した開発プラットフォームの検討も進めている．本研究ではそのプロトタイプを利用した開発を行なった.

HF 計算の GPGPU 化については，計算量的に多い二電子積分の計算だけでなく計算した行列要素の G 行列への加算がボトルネックとなることが知られている．CPU 上での計算では比較的演算コア数が少なかったため演算スレッド毎に行列全体を保持することが可能であったが，GPGPU には非常に多くの演算コアがありこれらがそれぞれ行列全体を保持するのは不可能になってしまっている．このため行列要素の加算を排他的に行なう必要があり，この排他制御が性能面での大きな障害となっていることが報告されている．成瀬らは二電子積分を一旦別領域に保存し，積分演算と行列への加算を分離することでこの排他制御を回避するアルゴリズムを提案している[7]．しかしながらこのアルゴリズムでは形式的には分子サイズの４乗個の積分の読み込みが発生するため，分子サイズが大きくなった時の影響が大きすぎるのではないかと考えられる.

我々は以前の研究において，G 行列がメモリに入らないような巨大な分子を扱う計算手法として Global Arrays Toolkit[8]を用いた分散アルゴリズムを提案している[9]．このアルゴリズムでは各プロセスが比較的小さなメモリ量を持てば良いため，当時としては巨大な分子についても計算が可能となっていた．この方法では行列要素の加算が比較的小さなベクトル要素への加算に抑えられることから，GPGPU 上での排他的な加算を回避する手法としても使えることは容易に推測できる.

†1 筑波大学
University of Tsukuba

本発表では排他的行列加算を行なわないアルゴリズムを提案し,それに基づきG行列計算のGPGPU化を行なった．本論文の構成は次のようになっている．2 節において HF 計算の理論とその疑似コードを示し，その後の 3 節では GPGPU 化についてステップごとに実装法の説明と性能評価を行なう．複数 GPU を用いた並列計算の評価をした後，4 節で先行研究について考察する．

2. Hartree-Fock 計算

ここでは HF 計算についての概略とその基本的なコードを紹介する．

2.1 Hartree-Fock-Roothaan 方程式

HF 計算は以下に示す Hartree-Fock-Roothaan 方程式について自己無撞着になるような分子軌道$\{\psi_a\}$を計算する手法である[10].

$$\mathbf{FC}=\mathbf{SC}\boldsymbol{\varepsilon} \tag{2.1}$$

$$\psi_a(\mathbf{r})=\sum_{i=1}^{N} C_{ia}\varphi_i(\mathbf{r};\mathbf{n}_i,\mathbf{R}_i) \tag{2.2}$$

$$F_{ij}=H_{ij}^{\mathrm{core}}+\sum_{k,l}^{N} D_{kl}\left\{2(ij\,|\,kl)-(il\,|\,kj)\right\} \tag{2.3}$$

$$S_{ij}=\int d\mathbf{r}\varphi_i(\mathbf{r})\varphi_j(\mathbf{r}) \tag{2.4}$$

ここで **F** は Fock 行列，**S** は重なり積分行列，**C** は分子軌道の係数行列，**ε**は軌道エネルギーベクトルであり，分子軌道は N 個の基底関数$\{\varphi_i\}$の線形結合で与えられる．$\mathbf{H}^{\mathrm{core}}$ および **D** は一電子 Hamiltonian 行列，密度行列でありそれぞれ次式で与えられる．

$$H_{ij}^{\mathrm{core}}=\int d\mathbf{r}\varphi_i(\mathbf{r})\hat{h}\varphi_j(\mathbf{r}) \tag{2.5}$$

$$D_{ij}=2\sum_{a}^{N_{elec}/2} C_{ia}C_{ja} \tag{2.6}$$

$\hat{h}$ は一電子 Hamiltonian 演算子，*Nelec* は電子数である．

本論文では(2.3)式の第 2 項を特に G 行列と呼ぶこととする．G 行列中の(*ij*|*kl*)は二電子積分と呼ばれ，基底関数の組 *i,j,k,l* について以下のように計算される．

$$(ij\,|\,kl)=\int d\mathbf{r}_1\int d\mathbf{r}_2\varphi_i(\mathbf{r}_1)\varphi_j(\mathbf{r}_1)\frac{1}{|\mathbf{r}_1-\mathbf{r}_2|}\varphi_k(\mathbf{r}_2)\varphi_l(\mathbf{r}_2) \tag{2.7}$$

G 行列を計算するのにこの積分が N^4 個必要となることから $O(N^4)$の計算量が必要となり，HF 計算において 99%以上の計算がこのために費やされている．

HF 計算では(2.1)式を自己無撞着にするために SCF(self-consistent field)計算と呼ばれる繰り返し計算を行なう．初期の分子軌道を順に更新していき密度行列が収束して変化しなくなるまで繰り返しを行なう手法である．なお $\mathbf{H}^{\mathrm{core}}$ は SCF の間に変化しない.

2.2 スクリーニング

二電子積分の数が非常に多いことが HF 計算のボトルネックになっているため，不要な二電子積分計算を削減するためのスクリーニング手法が考えられてきた．その一つが重なり積分によるスクリーニングである[11]．通常の基底関数はそれぞれの原子を中心としたガウス関数から構成されているため，原子間距離が離れているような基底関数同士ではその重なりは非常に小さくなる．その積から計算される二電子積分の値も同様に小さくなることを利用して積分計算を削減する手法である．これを利用すると *i,j,k,l* の 4 重ループで構成される G 行列計算は，重なりのある基底関数のペア *ij,kl* についての 2 重ループとして計算できる．

ijkl が揃った段階で行なうのが Schwarz の不等式(2.8)を用いたスクリーニング手法である[12].

$$(ij\,|\,kl)\le\sqrt{(ij\,|\,ij)}\sqrt{(kl\,|\,kl)} \tag{2.8}$$

事前に(*ij*|*ij*)および(*kl*|*kl*)を計算しておけば，その積から二電子積分(*ij*|*kl*)の取りうる最大値が決定でき，この値が閾値よりも小さい場合の計算を省略するスクリーニング手法である．特に(2.3)式による G 行列計算を考えると，スクリーニングは(2.8)そのものではなく密度行列要素との積に対して実行すれば良いことがわかる。この手法は SCF 計算において収束に近づいた時には非常に有効になってくる．G 行列の計算を前 SCF サイクルとの差分の形に変形すると，二電子積分と密度行列要素との積の部分は二電子積分と密度行列の差に置き換えることができる．SCF 計算が収束に近づいた時は密度行列の差分が小さいため，これの積によるスクリーニングが非常に有効に働くのである．

2.3 OpenFMO での実装

OpenFMO は九大の稲富らが開発した超並列 FMO 計算向きの FMO 実装である．本研究では OpenFMO から SCF 計算部分を切り出したコードを利用して開発を行なった．二電子積分の計算には小原積分[13]を採用し，積分タイプごとに別々の関数として G 行列の計算を行なっている．これは SIMD 動作をする GPGPU にとっては都合が良い実装である．なおここまでの説明では *i,j,k,l* を基底関数のインデックスとして説明したが，実際の HF 計算では基底関数は複数の縮約基底で展開されるため，これに対応する縮約シェル(contracted-shell)に関するインデックスとして読み替える必要がある．

OpenFMO の SCF 計算部分はラウンドロビンで静的に ij を分散させる静的負荷分散による OpenMP/MPI ハイブリッド並列化がなされている．またコードは C 言語で記述されており，CUDA による GPGPU 化にも適している．

2.4 (ps,ss)タイプ疑似コード

GPGPU 化のためのサンプルとしてここでは最も主要な積分タイプである(ps,ss)タイプを取り上げることとする．

(ps,ss)タイプの G 行列計算の疑似コードを図 1 に示した.

```
for (ijcs=0; ijcs<N_ijcs; ijcs++) {
  for (klcs=0; klcs< N_klcs; klcs++) {
    if (check_schwarz(ijcs, klcs, Dmax)) {
      x = calc_2e_psss(ijcs, klcs);
      for (i=0,iao=iao0; i<3; i++,iao++) {
        G[iao][jao] += 4*x[i]*D[kao][lao];
        G[kao][lao] += 4*x[i]*D[iao][jao];
        G[iao][kao] -=   x[i]*D[jao][lao];
        G[iao][lao] -=   x[i]*D[jao][kao];
        G[jao][kao] -=   x[i]*D[iao][lao];
        G[jao][lao] -=   x[i]*D[iao][kao];
      }
  }
}
```

図 1　(ps,ss)タイプ G 行列計算の疑似コード

2.2 節で示したように, G 行列の計算は重なり積分で値を持つ縮約シェルの組 ijcs,klcs に対する二重ループで与えられる．ここで縮約シェルペア ijcs に対応する基底関数のインデックス iao および jao は ijcs をインデックスとする配列で与えられるとする．klcs についても同様である．check_schwarz()関数は Schwarz の不等式を計算する関数であり，密度行列の差分の最大値 Dmax を利用してスクリーニングを行なう．

calc_2e_psss()は(ps,ss)タイプの二電子積分計算をする関数であり，シェルタイプに相当する数の二電子積分（ここでは 3 個）を返してくる．G 行列への加算は二電子積分の対称性により 6 つの要素への加算の形で書くことができる．

3. GPGPU 化

ここでは本研究で利用する GPGPU マシン HA-PACS について説明した後, GPGPU 化の詳細について順に説明する．

3.1 HA-PACS

密結合並列演算加速機構実証システム「HA-PACS」は筑波大学計算科学研究センターに導入された GPGPU クラスタシステムである．HA-PACS プロジェクトでは，演算加速装置を中心とする超並列計算機におけるアプリケーションおよび多数の演算加速装置を効率的に結合する並列システムの研究開発を狙いとしており，分子軌道計算を利用した生体分子のシミュレーションも対象アプリケーションの一つとなっている．

今回利用した HA-PACS ベースクラスタは，4 台の GPU を搭載した計算ノードが 268 ノード InfiniBand により接続された HPC システムである. 計算ノードは 8 コアの Intel E5 CPU(Sandy Bridge-EP, 2.6GHz)を 2 台に 128GB のメモリを搭載している．またこのノードには 4 台の NVIDIA M2090 GPU が装着されており，これらの GPU を効果的に利用することが HA-PACS システムの性能を引き出すための重要なポイントとなっている．

以下のベンチマーク計算では特に断わりのない限り以下の条件で HA-PACS を用いて計算を行なった．コンパイラは Intel icc 13.0 および NVIDIA CUDA5.0 を利用し，BLAS および LAPACK ライブラリとして Intel MKL 10.3 を利用した. 通信が必要な場合には MPI ライブラリとして Intel MPI 4.0 を利用し，InfiniBand を用いた通信を行なっている．

3.2 Naïve 実装

図 1 で示されたコードを GPGPU 上で動作させるため，ijcs に関するループをスレッドブロックに，klcs に関するループを各スレッドに分配する並列化手法を採用した．この処理分割手法の場合には G 行列が複数のスレッド間で共有されているため，G 行列への加算を排他的に行なう必要がある. しかしながら現在の CUDA では倍精度浮動小数点数に対する atomicAdd()演算がサポートされていないため, atomicCAS()を利用した排他制御を利用したアトミック演算で定義することになる[14]. 特に HA-PACS で採用されている Fermi 世代の GPU では最新の Kepler 世代の GPU に比べこの処理が遅いことがわかっており[15]，G 行列への加算がボトルネックになることが予想される．また GPGPU での実行では各スレッドが SIMD 動作を行なうため，Schwarz 不等式を利用したスクリーニング（check_schwarz()）についても特別な考慮が必要となる．

naïve 実装として G 行列への加算を倍精度浮動小数演算版 atomicAdd()に置きかえたものを実装して性能を測定した．疑似コードは再掲しないが，G 行列へのアトミック加算処理を最少とするために一時変数を使ってできるだけ外側のループに移動している．ループのタスク分割は ijcs および klcs インデックスをラウンドロビンでそれぞれスレッドブロック，スレッドに分配した．また G 行列はスレッドブロック毎に全体を持つようにし，全積分タイプについて計算後に加算してホストに転送している．密度行列 Dmax および D については G 行列計算カーネル中で書き換えないため，GPU ごとに一つずつを各 SCF サイクルの最初に転送している．G[iao][jao]のための一時配列変数を共有メモリに取った以外は G 行列を含めグローバルメモリ又はローカルメモリを利用している．

サンプル計算はグリシン 5 量体（38 原子）についての HF/6-31G(d)計算で, これを HA-PACS の 1 台の GPU を利用して計測した(表 1). この計算では収束までに 14 回の SCF サイクルを費やしているが, 表 1 の時間は (ps,ss)タイプの G 行列計算について全てのサイクルで費やした時間の和を示している．CPU は HA-PACS の CPU 1 コアを利用した場合の経過時間であり，GPU については 64 スレッドブロッ

ク，128 スレッド/スレッドブロックの構成を利用した場合の経過時間になっている．前述したように表 1 に示された時間には行列の転送時間は含まれておらず，計算だけの時間となっている．

CPU 1 コアに対し naïve な実装では 5.5 倍の性能向上となっている．アトミック加算の影響を調べるために排他制御なしでの時間測定をするのが望ましいが，その場合は正しい計算結果を与えず SCF サイクルの過程を再現できないためここでは行なっていない．

表 1　(ps,ss)タイプ G 行列計算の性能評価

プロセッサ	高速化手法	Time/s	Speedups
CPU(1core)		54.2	x1.0
GPU(64, 128)	naïve	9.8	x5.5
	アトミック加算削除	5.9	x9.2
	Gi,Gj 加算削減	5.4	x10.1
	スクリーニング分離	4.6	x11.9
	動的負荷分散	4.1	x13.1

3.3 アトミック加算の削除

排他制御は一つの積分につき 6 つの行列要素 G_{ij}, G_{kl}, G_{ik}, G_{il}, G_{jk}, G_{jl} への加算について必要になっている．我々はこの行列要素が 3 種類，即ち G_{i*}, G_{j*}および G_{kl} に分類されることに着目し，アトミック加算を行なわない G 行列の更新法を提案する．これは以前著者が提案した大規模 Fock 行列（G 行列）生成アルゴリズムを GPGPU アーキテクチャに応用したものである．このアルゴリズムでは個々のプロセスは計算に必要な部分配列 G_{i*}, G_{j*}, G_{kl} のみを利用し，適宜 Global Array 上に分散して保持している G 行列に加算を行なっている．Global Array への加算のためにアトミック加算が必要であるが，加算は klcs ループの外でありさらに Global Array の非同期通信を利用することで排他制御等のコストを隠蔽することが可能であった．以下で今回提案するアルゴリズムを説明する．

このアルゴリズムのポイントは，ある ijcs に対する G 行列計算を考えた時に G_{kl} 以外は G_{i*}, G_{j*}という限られた範囲しか行列を更新しないことにある．このため G_{i*}および G_{j*} さえ各スレッドで保持していれば，klcs ループ内部での G 行列更新はローカルな G_{i*},G_{j*}への更新に置き換えることができ，排他制御を考えなくても良くなる。残る G_{kl} 要素であるが，G_{kl} は klcs で与えられる kao,lao のペアに対応する G 行列要素であり，異なる klcs でその位置が重複することはない．すなわち klcs が異なる計算を行なっている限りにおいては，排他制御が必要なくなる．

本研究で行なった実装では，Gi[Nao]および Gj[Nao]配列をスレッド数分だけグローバルメモリに確保し各スレッドの作業用配列とした．また G 行列へのアクセス範囲を制限しキャッシュの利用を促進するため，G_{kl} 要素に対応する G 行列の配列 Gkl[Nklcs]をスレッドブロックに一つ確保した．全てのスレッドが異なる klcs を計算することを保証するため，klcs ループ終了後にスレッド間の同期を必要とする．また同期をとることによって Gi[], Gj[]への各スレッドへの更新が終了したことも保証している．

Global Array を用いた以前の実装と異なり，GPU を用いる本研究の実装では klcs ループ終了後にスレッドブロック内の全てのスレッドの Gi[]および Gj[]を集約し G 行列に加算する．Gi[],Gj[]の集約については配列の各要素を順に加算して G 行列を更新していけば良いだけのため排他制御の必要はない．一方 Gkl[]については ijcs ループ終了後に G 行列に加算するだけである．もちろん Gkl[]の要素は G 行列の別々の要素に対応しており，この更新についてもまた排他制御をする必要がない．

この実装を行なったコードについても 3.2 と同様のベンチマーク計算を行なった(表 1)．atomicAdd()を用いた実装で 9.8 秒かかっていた計算が 5.9 秒と半分近くにまで短縮されているのがわかる．これにより CPU 1 コアに対し 9.2 倍の速度を示したことになる．

3.4 Gi[],Gj[]加算の最適化

3.3 での実装において Gi[],Gj[]は基底関数の全体について集約および G 行列への加算処理を行なっていた．しかしながら G 行列計算における積分タイプが限定されている状況では全基底関数を扱う必要はない．例えば(ps,ss)タイプでは kao および lao は s 型の基底関数の範囲しか取らないため，その範囲の要素についてのみ集約・加算処理を行なえば十分である．

この実装を加えたコードに対し再度ベンチマーク計算を実行したところ，5.4 秒と 0.5 秒程度の性能向上が見られた．本ベンチマーク計算では基底関数の総数が 349 個，s 型の基底関数の数が 96 個であるので，集約する配列要素数はおよそ 1/3 程度に減少した。この結果から Gi[],Gj[]行列の集約には 0.5 秒弱程度のコストが掛っていることがわかる．

3.5 Schwarz スクリーニングの分離

3.2 でもコメントしたように，SIMD 動作を行なう GPGPU では分岐のコストが大きくなるため Schwarz スクリーニング処理についても特別な考慮が必要と思われる．SCF サイクルが収束に近づいた時のように効果的に積分がスクリーニングされる場合，疑似コード（図 1）の実装では極一部のスレッドのみが計算を行ないほとんどのスレッドがその終了を待つことになりかねない．そこで Schwarz スクリーニングの過程を klcs ループから分離し，スクリーニングされて残った klcs についてのみ計算のためのループを実行するよう実装した．

分離されたスクリーニングプロセスについては klcs ループをスレッドに分割し，生き残った klcs を記録する配列 surv_klcs[]に保存することとした．この surv_klcs[]への格納

に関して共有メモリ上にカウンタを用意し，整数タイプのatomicAdd()演算を行なってインデクシングを行なっている．klcs に関するの本体ループは得られた surv_klcs[]の要素を各スレッドに分配する形で実装した．
Schwarz スクリーニングの分離処理を加えた実装に対するベンチマーク計算をこれまでと同じ条件で行なった(表 1)．5.4 秒かかっていた(ps,ss)タイプの G 行列計算時間が 4.6 秒まで短縮した．今回のテストケースは比較的小さい分子を扱っているため大きな性能向上にはなっていないが，1,000 基底を超えるような分子ではスクリーニングの効果が大きいため，より大きな速度向上が期待される．

3.6 動的負荷分散

二電子積分のスクリーニングは負荷分散にも影響を与えている．ある ijcs に対応する計算量はスクリーニングにより生き残る二電子積分の数により決まるが，これを理論的に推定するのは困難である．特に密度行列との積を利用するスクリーニングでは SCF サイクルごとにスクリーニングの効果も変化するため，毎回実測する以外にないと言える．ここまでの実装では ijcs についてはラウンドロビンでスレッドブロックに分配する静的負荷分散手法を採用しているが，この手法では並列度が上昇すると負荷の不均衡による性能低下が顕著になることがわかっている．

そこでスレッドブロックで共有するカウンタを用いた動的負荷分散手法の実装を行なった．カウンタはグローバルメモリに配置され，各スレッドブロックのマスタスレッドのみが atomicAdd()により排他的にカウンタの取得と加算を実行する．排他制御を行なうコストと負荷分散改善のバランスを考えて，ここでは計算するべき ijcs の値を一度に 2 つずつ取得することとした．また最初に計算する ijcs についてはカウンタへのアクセスの競合が明確であるため，事前に与えている．

動的負荷分散による性能向上を表 1 に示した．この時もスレッドブロック数，ブロック当たりのスレッド数はこれまでと同じく 64 スレッドブロック，128 スレッド/ブロックのままである．3,603 個の ijcs を 64 個のスレッドブロックにラウンドロビンで静的に分配した場合に 4.6 秒であったのが，カウンタによる動的負荷分散を用いることで 4.1 秒まで高速化されている．

3.7 HF 計算全体としての性能評価

ここまで(ps,ss)タイプの G 行列計算について取り上げてきたが，実際には(ss,ss)から(dd,dd)，さらに大きな基底関数を使った場合には f 関数や g 関数が含まれる積分も存在する．我々は(ps,ss)以外の積分タイプについても実装を進めており，現時点では(ps,ss)の他に(ss,ss)と(ps,ps)タイプについての G 行列作成ルーチンの GPGPU 化が終了している．これら積分タイプについての G 行列更新は上記(ps,ss)タイプで提案したアルゴリズムとほぼ同じものを利用している．

これまでと同じテスト計算について，積分タイプごとの実行時間および SCF 計算全体の実行時間を表 2 に示した．ここで GPGPU 化された 3 タイプ以外の積分タイプについては others にまとめて示している．また GPU の初期化に関する時間は含まれていない．

積分タイプが大きくなるにつれて積分計算ルーチン内のループ長が揃わなくなるため，G 行列更新部分のアルゴリズムが同じであっても対 CPU 比での性能は低下してしまう．その結果(ss,ss)や(ps,ss)では CPU 1 コアに対し 13.1 倍の性能であった性能が(ps,ps)では 6.8 倍にまで低下してしまっている．これを防ぐために基底関数のさらに詳細な並べ替えについての検討も計画している．

表 2　HF 計算の性能評価

	CPU(1core)	GPU(64block, 128thread)		
	Time/s	Time/s	Speedups	Time(wO)/s
Total	360.1	246.7		236.7
(ss,ss)	24.7	1.9	x13.1	0.0
(ps,ss)	54.2	4.1	x13.1	0.0
(ps,ps)	40.0	5.9	x6.8	0.0
others	240.1	233.7		235.6

今回行なった GPGPU 化では動的負荷分散のケースも含めて GPU のみで計算を実行するような実装であり，CPU は何もしていない状態になっていった．そこで CPU の同時実行についても検討を行なった．個々の積分タイプについての計算について GPU と CPU で同時に実行する利用する方法ではホストと GPU 間で負荷バランスを取らねばならず問題が複雑になってしまう．そこで積分タイプごとに CPU または GPU に振り分ける手法を採用した．特に d 関数を含むような二電子積分の計算では一時配列が非常に大きくなるなど GPU 化しても性能があまり出ないことが考えられ，これを CPU に振り分けることは合理的と思われる．

ここまでの性能評価では，積分タイプごとの実行時間を測定するためにカーネル関数呼び出しの前後で明示的にカーネル関数の終了を待つように実装していた．そこでこの終了待ちをコメントアウトし，GPU と CPU の計算をオーバーラップさせる実験を行なった（表 2 の Time(wO)）．カーネル関数の終了を待たないため，GPU で実行する積分タイプの計算は CPU からの計測では見掛け上時間がかかっていない．一方 CPU で計算する部分についてもほとんど性能に影響はなく，計算全体に費やす時間のみが減少しており，CPU と GPU の計算オーバーラップが機能していることが確認できた．

3.8 複数 GPU の利用

HA-PACS システムではノードあたり 4 台の GPU が利用可能であり，大きな分子軌道計算ではこれらを効果的に利用することが望まれる．また OpenFMO プログラムは

OpenMP/MPI ハイブリッド並列に対応しており，この並列化と組み合わせることも重要である．本研究では MPI 1 ランクにつき 1 台の GPU を割り当てる形で複数 GPU に対応した. GPU の制御は OpenMP のマスタスレッドのみが行なっている．CPU 間および GPU 間は OpenFMO の元コードと同じように静的負荷分散を採用し，GPU 内のスレッドブロック間については動的負荷分散を行なっている．

HA-PACS 上でこの実装の性能評価を行なった. HA-PACS では各ノードに 8 コア CPU が 2 台および GPU が 4 台実装されているため，ノードごとに 4 スレッド並列で実行される MPI プロセスを 4 つ起動する構成になっている．性能評価に用いた分子は DNA の CG 対二つからなるモデル分子で 126 原子からなっている．この分子の HF/6-31G(d)計算(1,282 基底)を 4 ノードまで利用して実行した． 表 3 にこのテスト計算についてマスタプロセスで計測した実行時間を示した．なおこの計算において SCF サイクルは 15 回まで行なったが結果は収束していない．また各積分タイプの実行時間測定においてはプロセス間の同期を取っていないので，プロセス間の負荷のアンバランスが見えている可能性がある．

CPU 1 プロセス（4 コア）を用いた場合に比べ 4 ノード 16GPU を利用した場合には 20 倍から 40 倍の性能向上を示している．表 2 において CPU 1 コアに対し 13.1 倍および 6.8 倍の速度向上が得られていたことを考えると，この性能は並列化効率 75%程度に相当すると概算できる．SCF 計算全体を見ると G 行列計算よりも低い性能向上にしかなっていないが，これは SCF サイクルごとに行なっている対角化の時間が見えてしまっているからである．

前節と同様に CPU と GPU の計算のオーバーラップをさせた結果（Time(wO)）を見ると，このケースでも計算のオーバーラップによって GPU の計算時間は完全に隠蔽されており計算全体が高速に実行できている．

表 3　複数 GPU を用いた性能評価

#procs	CPU	GPU(64 block, 128 thread)			
	1	16		16	
	Time /s	Time /s		Time(wO) /s	
Total	3,138.5	210.6	(x14.9)	179.9	(x17.4)
(ss,ss)	169.4	4.0	(x42.8)	0.0	
(ps,ss)	416.8	10.9	(x38.3)	0.0	
(ps,ps)	354.0	16.5	(x21.4)	0.0	
others	2,121.9	137.0	(x15.5)	136.7	(x15.5)

4. 先行研究

G 行列作成の GPGPU 化おける行列加算の問題とその解決策を扱ったものとしては，成瀬らの報告があげられる．彼らは二電子積分計算部分と行列への加算部分を分離することで排他制御を行なわない行列加算を実現した．具体的には計算した二電子積分を一旦グローバルメモリに保存する．その後全スレッドでその積分を評価し，自分に割り当てられた G 行列の範囲についての加算を実行する．G 行列への加算は共有メモリを用い，加算終了後にグローバルメモリにコピーする方法を取っている．我々の提案手法の利用メモリ利用が分子サイズとスレッド数に比例することに比べ，成瀬らの方法は利用するメモリ量をコントロールしやすいため大きな分子にも適用が容易だと思われる．

しかしながら共有メモリのサイズには限界があるため，G 行列全体を更新するためには多数回保存した積分を読み込むことになる．成瀬らの方法では G_{kl} 要素が行列全体を動いてしまうため，行列の全領域の更新が必要となるからである．我々が今回提案したアルゴリズムを使えば G_{kl} 要素の加算に排他制御は必要ないことに着目すると，成瀬らの手法に我々の提案手法を組み合わせることで更新する G 行列の範囲を制限することができ，彼らの手法の弱点を克服できると期待できる．

5. おわりに

本発表では分子軌道計算の基礎的な計算手法である HF 計算について, その主たる構成要素の G 行列計算を GPGPU 化しその性能の評価を行なった．G 行列計算では行列への加算に排他制御が必要なことが大きな障害となっていたが, それを避けるアルゴリズムを提案し高速化を実現した．またスクリーニングや動的負荷分散, さらには複数 GPU の利用や CPU との同時計算など多くの高速化のための手法を実装している．性能評価では(ps,ss)タイプの積分について CPU 1 コアの 13 倍程度の性能を実現している．複数 GPU を用いた並列計算では HA-PACS 4 ノードを用いて CPU 4 コアに対し 20 倍から 40 倍程度の性能向上を実現した．また成瀬らの手法に我々の提案手法を組み合わせることで, 彼らの手法をさらに高速化できる可能性も示した．

今後の課題として，他の積分タイプの GPGPU 化や成瀬らの手法への応用があげられる．一方で G 行列計算のアルゴリズムは今回取り上げたものだけではなく，分子のサイズなどによって様々な手法が開発されており，そのような手法の GPGPU 化も視野に入れた開発が必要である．

HF 計算は分子軌道計算の基本ではあるものの ，実用的にはより高精度な計算手法やプロパティが必要とされる．このため多くの計算手法を組み合わせることが重要であり, OpenFMO も単に大規模 FMO 計算のためのアプリケーションとしてだけではなく，高速な分子軌道計算のためのプラットフォームとして発展させていく必要がある．本研究はそれに向けた開発協力の一環である．

謝辞 本研究で使用した HF 計算コードは九州大学の稲富らによる OpenFMO プログラムの一部を抜粋して提供していただいている．また本研究の一部は文部科学省特別経費「エクサスケール計算技術開拓による先端学際計算科学教育研究拠点の充実」事業, および JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」, 研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」による．

参考文献

1) PetaChem, http://www.petachem.com/
2) Yasuda, K.: Two-Electron Integral Evaluation on the Graphics Processor Unit, Journal of Computational Chemistry, 29, pp.334-342 (2008).
3) HA-PACS, http://www.ccs.tsukuba.ac.jp/CCS/research/project/ha-pacs
4) OpenFMO, http://www.openfmo.org/OpenFMO/index.html
5) Maki, J. et al.: One-sided Communication Implementation in FMO Method, Proc. HPCAsia07, pp.137-142 (2007).
6) 稲富雄一 他：並列 FMO プログラム OpenFMO の性能最適化, 情報処理学会研究報告, Vol. 2012-HPC-133, No. 35, pp. 1-8, 2012
7) 成瀬 彰: 分子軌道法プログラムの GPGPU 化, サイエンティフィック．システム研究会 アクセラレータ技術 WG 成果報告書 [2012-03-26 発行], 添付資料 16 (2012), http://www.ssken.gr.jp/MAINSITE/download/wg_report/accelerator/index.html
8) Global Arrays Toolkit, http://www.emsl.pnl.gov/docs/global/
9) Umeda, H. et al.: Parallel Fock matrix construction with distributed shared memory model for the FMO-MO method, Journal of Computational Chemistry, 31, pp.2381-2388 (2010).
10) Szabo, A. and Ostlund, N. S.: 新しい量子化学, 東京大学出版会 (1987).
11) Almlöf, J. et al.: Principles for a Direct SCF Approach to LCAO-MO Ab-Initio Calculations, Journal of Computational Chemistry, 3, pp.385-399 (1982).
12) Häser, M. and Ahlrichs, R.: Improvements on the Direct SCF Method, Journal of Computational Chemistry, pp.104-111 (1989).
13) Obara, S. and Saika, A.: Efficient recursive computation of molecular integrals over Cartesian Gaussian functions, Journal of Chemical Physics, 84, pp.3963-3974(1986).
14) NVIDIA: CUDA C Programming Guide Version 4.2, Appendix B.11, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
15) NVIDIA: Tuning CUDA Applications for Kepler, http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html