修 士 論 文

# 量子モンテカルロ計算の高速化に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

寺島 義晴

2010 年 3 月

修 士 論 文

# 量子モンテカルロ計算の高速化に関する研究

指導教官　前園涼 講師

審査委員主査　前園涼 講師
審査委員　松澤照男 教授
審査委員　金子峰雄 教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

0810040 寺島 義晴

提出年月: 2010 年 2 月

## 概 要

本研究は、計算シミュレーション分野にて分子や原子を対象とした電子状態計算の一手法である量子モンテカルロ（QMC）計算の計算コードについての高速化に関する研究を扱う。量子モンテカルロ法は他手法に比べて電子の多体効果について、より実直に取扱う手法であり、電子の多体効果が重要となる生体分子系などのエネルギー計算において、有用な手法とされる。生体分子の中でも、タンパク質やDNAといった生体高分子と呼ばれる大規模分子系では、規模が大きすぎるために計算困難という問題がある。この大規模分子系を分割し、小規模の系（フラグメント）に分割することで計算可能にする手法として、フラグメント分子軌道法がある。これを量子モンテカルロ計算に適用した計算コードにより、大規模分子系は計算可能となるが、フラグメント化に呼応する余分の処理が原因となって、通常の量子モンテカルロ計算に比べて50倍程度遅くなるという問題が起きている。本論文では、上記の問題に対して、画像処理演算装置（GPU）を用いた高速化による改善を試み、実装を行なった結果と考察について述べる。実験には、FMO法を扱う上でのベンチマーク的な系であるグリシン三量体の最小フラグメントを対象系として据え、FMO-QMC計算を行ない、その計算精度と計算時間について比較を行なった。

実験の結果、精度上ではCPUとGPUの間には$\pm 1.0 \times 10^{-12}$程度の誤差しか現れず、単一プロセス上で計算を行なった場合では7.2倍、CPU内4並列の場合との比較では1.8倍GPUの計算速度が速くなることがわかった。

今回実験に用いたCPU（Intel Core i7 920）とGPU（GeForce GTX275）の倍精度実数演算に関する理論性能はそれぞれ、44.8 GFLOPS、84.24 GFLOPSとなっており、この比は上記で達成された比にほぼ一致する。この結果より、FMO-QMC計算の計算時間が倍精度演算に対する処理能力にスケールされると予測されるので、倍精度演算の性能が高いGPUを用いることで、更なる高速化を図ることが期待される。GPUの開発メーカーの一つであるNVIDIA社は、倍精度演算において624 GFLOPSの理論性能を持つとされる次世代GPUアーキテクチャ「Fermi(フェルミ)」に基づくGPUを発表しており、GPGPUによる高速化の展望は明るいと思われる。

現状、1つのノードに対して1台のGPUを割り当てているので、1つの計算ノード上ではFMO-QMC計算プロセスは現在1つしか実行出来ない。そのため、CPU上の利用していないプロセッサコアが空き状態となっている。このプロセッサコア上にて、GPU上の計算とは独立である計算をOpenMPなどの並列化プログラミングを用いて同時並行に行なえるようにすることで、1ノード辺りの計算時間は更に削減可能だと考えられる。また、現行のGPUは、単精度演算器が倍精度演算器の8倍搭載されているので単精度実数演算能力が非常に高い。よって、単精度実数による演算に精度を落としても結果の精度に影響を及ぼさない箇所については単精度実数にて演算させることで、高速化を図ることも可能と考える。

# 目 次

# 第1章　序論

## 1.1　背景

近年、CPU を初めとする演算装置は目覚しい成長を遂げ、他方では並列化技術が大きく発展したため、それまでは行なえなかったような大規模な計算が可能となり、計算機シミュレーションの分野が賑わっている。計算物理分野での例を挙げると、10 年前には計算機上で計算させること自体が困難であった生体分子のエネルギー計算が、数年前には数日の時間を掛けることで計算可能となり、現在では 1 日のうちに計算が収まってしまうといった具合である[1]。

計算物理学における計算機シミュレーションには、宇宙の構造をシミュレートするようなマクロな世界を対象とする分野、電子の動きをシミュレートして原子に働く力を算定するミクロな世界を対象とする分野など様々な分野が存在する。我々の研究グループが主務として行なっている計算は、分子や原子などのミクロな世界を舞台としたエネルギー計算である。より具体的には、物質の構造を表す一つの指標である粒子の基底状態における結合エネルギーを電子状態計算により求めていく。

電子状態計算には、経験的な近似を取り入れる立場である模型計算と経験的な近似をなるべく排除して客観性を重視する第一原理（*ab initio*）計算の二つの立場があるが、本研究グループが扱う電子状態計算は第一原理計算であり、手法としては、他手法に比べて電子間の相互作用について、より実直に取り扱う量子モンテカルロ法（QMC；Quantum Monte Carlo）[2] という手法を用いる。量子モンテカルロ計算は並列化効率が 99% 以上と並列計算との相性が非常に良く、大規模計算機を利用し並列数を稼ぐことで時間的コストを非常に大きく削減することが出来る。近年、大規模並列計算機の性能が向上し、研究室単位で 100 並列〜1,000 並列クラスの並列計算機を持つことも珍しくなくなり、10,000 並列クラスの並列計算機が普及することも近いと予想される。量子モンテカルロ計算は 1,000 並列程度までの並列度では前述の通り 99% 以上の並列化効率を発揮するが、10,000 並列以上の並列計算では並列化効率が落ちることが予見されている [1]。そのため、量子モンテカルロ計算の高速化について、並列化による高速化だけでなく、単体性能の向上による高速化についても議論する時がきている。

[1] ▷ 7.2 昨今の大規模計算機
[2] ▷ 2.2 量子モンテカルロ法

## 1.2 本研究の目的

本研究では、第一原理計算による電子状態計算手法の一つである量子モンテカルロ法に対して、計算速度面での性能向上を図ることを目的とする。特に、フラグメント分子軌道法（FMO；Fragment Molecular Orbital）[3] を利用出来るように拡張された量子モンテカルロ計算コード（FMO-QMC）[2] について、ハードウェア面からの高速化について議論する。

FMO 法は、生体高分子などの計算困難な大規模分子系を、計算可能な小規模の系（フラグメント）に分割し、それぞれのフラグメントの計算結果を再統合することで全系の計算を行なう手法である [3, 4]。計算困難な系を計算可能にする方策として非常に有用な手法であるが、FMO 法を量子モンテカルロ計算に適用した FMO-QMC 計算では、各フラグメントのエネルギー計算が同等のサイズの系を従来の量子モンテカルロ計算する場合に比べて 50 倍程度遅くなるという問題がある。この速度低下の根本は他フラグメントからの寄与を計算する処理であり、高速化を行なう余地がある。

また、高速化の手法として、GPU という画像処理演算装置を汎用計算用のプロセッサとして用いるアプローチ（GPGPU [4]）が近年注目されている。GPU は、3D グラフィクスの画像演算のために単純計算に向いた高性能なプロセッサを大量に搭載しており、安価で高性能な汎用計算用の外部ハードウェアとして利用するための研究が 2003 年頃から模索されてきた [5]。近年、デバイスによる制限が少ない GPGPU 開発環境が発表され、敷居が下がってきたため、様々な分野で活用が期待されている。国内でも良く研究されており、2009 年には長崎大学の浜田氏らが GPU を利用した PC クラスタに関する研究においてゴードン・ベル賞を受賞している [6]。

本論文では、FMO-QMC 計算コードの FMO 法に関する計算部分の高速化を GPU によって実現するための方法について議論し、実際に実装を行なった結果と考察について述べる。

## 1.3 本論文の構成

本論文では、まず第 1 章にて本研究の背景と目的・本論文の構成について述べる。続く第 2 章では、本研究の舞台となる量子モンテカルロ法による電子状態計算について述べ、今回高速化について模索する対象であるフラグメント分子軌道法を用いた量子モンテカルロ計算について概要を述べる。第 3 章では、今回行なう高速化手法である GPGPU について、原理と特徴について述べ、今回実装する上で用いた GPGPU ソフトウェア開発環境 CUDA [5] について概観と仕組みを述べる。第 4 章では、高速化を行なう上で、具体的な実装方法について述べ、高速化の評価方法や、実験対象となる系について述べる。第 5 章では、実験の結果について示し、結果や改善方法について考察し、第 6 章で結論をまとめる。また、本論文を補完する付録を第 7 章に記載する。

[3] ▷ 2.3 フラグメント分子軌道法

[4] ▷ 3.1GPU と GPGPU

[5] ▷ 3.2CUDA

# 第2章　第一原理計算と量子モンテカルロ法

本章では、計算の基礎原理となる第一原理計算の支配方程式と計算手法としての量子モンテカルロ法について概要を述べる。また、計算困難な大規模分子系を計算可能とする手法であるフラグメント分子軌道法について概要を説明し、フラグメント分子軌道法を量子モンテカルロ計算に適用した FMO-QMC 計算コードについて述べる。

## 2.1　第一原理計算の基礎原理

本研究が属する第一原理計算では、その支配方程式は以下に示す時間依存しない多体のシュレーディンガ方程式で表される。

$$\left(-\frac{1}{2}\sum_{j=1}^{N}\nabla_j^2 + V\left(\vec{r}_1,\cdots,\vec{r}_N\right)\right)\cdot\Psi\left(\vec{r}_1,\cdots,\vec{r}_N\right) = E\cdot\Psi\left(\vec{r}_1,\cdots,\vec{r}_N\right) \tag{2.1}$$

ここで、$\Psi$ は未知の固有関数であり、電子の配位セット $\{\vec{r}_j\}_j^N$ を引数とする多体の波動関数を意味する。$E$ はエネルギー固有値を意味し、(2.1) 式の $\nabla$ を含む項及び $V\left(\vec{r}_1,\cdots,\vec{r}_N\right)$ は運動エネルギー、ポテンシャルエネルギーをそれぞれ表す。物理定数が陽に表れないのは、原子単位系（a.u.）[1] 用いるためである。$\Psi\left(\vec{r}_1,\cdots,\vec{r}_N\right)$ は電子の特性による束縛条件により以下を満たさなければならない。

$$\Psi\left(\cdots,\vec{r}_j,\cdots,\vec{r}_i,\cdots\right) = (-)\cdot\Psi\left(\cdots,\vec{r}_i,\cdots,\vec{r}_j,\cdots\right) \tag{2.2}$$

この条件を満たす波動関数 $\Psi$ としては、例えば以下のような行列式が用いられる。

$$\Psi\left(\vec{r}_1,\vec{r}_2,\cdots,\vec{r}_N\right) = \frac{1}{\sqrt{N!}}\begin{vmatrix} \phi_1\left(\vec{r}_1\right) & \cdots & \phi_1\left(\vec{r}_N\right) \\ \vdots & \ddots & \vdots \\ \phi_N\left(\vec{r}_1\right) & \cdots & \phi_N\left(\vec{r}_N\right) \end{vmatrix} \tag{2.3}$$

(2.1) 式を解くことでエネルギー $E$ が求まるのだが、これは多変数の偏微分固有値問題であるため、厳密解を得ることが非常に困難である。(2.1) 式を近似的に解くための手法

[1] ▷ 7.1 原子単位系

は歴史的によく研究されており、ホーヘンベルグ・コーンらによる密度汎関数理論に基づく密度汎関数法では、$3N$ 次元空間記述である (2.1) 式を避けて、3 次元空間記述である電荷密度を基本量とした等価な一体問題に置き換えて計算する。密度汎関数法は多体問題を等価な一体問題に変換したことにより非常に高速に計算を行なうことができ、また、実験結果をしばしば良く再現するので、第一原理計算の主流として固体の計算などに用いられている。しかしながら、多体の相互作用を等価に表現する一体の実効ポテンシャルを設定する際に実質的には近似が導入される。そのような一体問題形式への置換えを用いず、(2.1) 式を $3N$ 次元空間記述のまま直接取り扱う手法として量子モンテカルロ法がある。

## 2.2 量子モンテカルロ法 ；Quantum Monte Carlo

以下 に量子モンテカルロ法の基礎原理について記す。以後、(2.1) 式最左辺の演算子部分を

$$\hat{H} := -\frac{1}{2}\sum_j \nabla_j^2 + V(\vec{r}_1, \cdots, \vec{r}_N) \tag{2.4}$$

と書き、

$$\vec{R} = (\vec{r}_1, \cdots, \vec{r}_N) \tag{2.5}$$

として書くこととする。(2.1) 式に対して、両辺に左から $\Psi^*$ を掛けると (2.1) 式は

$$\Psi^*\left(\vec{R}\right)\hat{H}\Psi\left(\vec{R}\right) = \Psi^*\left(\vec{R}\right)E\Psi\left(\vec{R}\right) \tag{2.6}$$

となり、エネルギー固有値 $E$ は

$$E = \frac{\int d\vec{R}\cdot\Psi^*\left(\vec{R}\right)\cdot\hat{H}\Psi\left(\vec{R}\right)}{\int d\vec{R}\cdot\Psi^*\left(\vec{R}\right)\cdot\Psi\left(\vec{R}\right)} \tag{2.7}$$

$$= \frac{\int d\vec{R}\cdot\Psi^*\left(\vec{R}\right)\cdot\hat{H}\Psi\left(\vec{R}\right)}{\int d\vec{R}\cdot\left|\Psi\left(\vec{R}\right)\right|^2} \tag{2.8}$$

という多重積分を実行することで求めることが出来る。

更に、(2.8) 式の分子に $\Psi\left(\vec{R}\right)\Psi^{-1}\left(\vec{R}\right)=1$ を挿入すると、

$$\begin{aligned} E &= \frac{\int d\vec{R}\cdot\Psi^{*}\left(\vec{R}\right)\Psi\left(\vec{R}\right)\Psi^{-1}\left(\vec{R}\right)\hat{H}\Psi\left(\vec{R}\right)}{\int d\vec{R}\cdot\Psi^{*}\left(\vec{R}\right)\cdot\Psi\left(\vec{R}\right)} \\ &= \frac{\int d\vec{R}\cdot\left|\Psi\left(\vec{R}\right)\right|^{2}\cdot\Psi^{-1}\left(\vec{R}\right)\hat{H}\Psi\left(\vec{R}\right)}{\int d\vec{R}\cdot\left|\Psi\left(\vec{R}\right)\right|^{2}} \\ &= \int d\vec{R}\cdot\frac{\left|\Psi\left(\vec{R}\right)\right|^{2}}{\int d\vec{R}\cdot\left|\Psi\left(\vec{R}\right)\right|^{2}}\cdot\Psi^{-1}\left(\vec{R}\right)\hat{H}\Psi\left(\vec{R}\right) \end{aligned} \tag{2.9}$$

と変形出来る。ここで、(2.9) 式の青字の箇所を

$$P\left(\vec{R}\right)=\frac{\left|\Psi\left(\vec{R}\right)\right|^{2}}{\int d\vec{R}\cdot\left|\Psi\left(\vec{R}\right)\right|^{2}} \tag{2.10}$$

とすると $P\left(\vec{R}\right)$ は、確率密度関数が満たすべき性質、

$$\int d\vec{R}\cdot P\left(\vec{R}\right)=1 \quad , \quad 0\leq P\left(\vec{R}\right)\leq 1 \tag{2.11}$$

を有する。よって、$P\left(\vec{R}\right)$ の分布に従って発生させたサンプリング点列 $\left\{R_j\right\}_j^N$ を用いれば、(2.9) 式は

$$\begin{aligned} E &= \int d\vec{R}\cdot P\left(\vec{R}\right)\cdot\Psi^{-1}\left(\vec{R}\right)\hat{H}\Psi\left(\vec{R}\right) \\ &= \left\langle\int d\vec{R}\cdot\Psi^{-1}\left(\vec{R}\right)\hat{H}\Psi\left(\vec{R}\right)\right\rangle_{P\left(\vec{R}\right)} \\ &\approx \frac{1}{N}\sum_{j}^{N}\Psi^{-1}\left(\vec{R}_j\right)\hat{H}\Psi\left(\vec{R}_j\right) \end{aligned} \tag{2.12}$$

と統計評価を用いて近似評価できる (大数の法則)。このとき、統計誤差は $\sqrt{N}$ に反比例し、$N\to\infty$ で (2.9) 式の多重積分に一致する (中心極限定理)。

このようなサンプリングはモンテカルロ法を用いて実行できる [7, 8]。従って、一度 $\Psi$ が決定すればこのようにして固有値 $E$ を統計平均として求めることが可能である。

しかし、そもそも元の固有値問題の未知量 $\Psi$ は与えられていない。そこで、$\Psi$ に近い

と考えられる波動関数 $\Psi_{\mathrm{Trial}}$ を試行推定とし、

$$E_{\mathrm{Trial}} = \left\langle \int d\vec{R} \cdot \Psi_{\mathrm{Trial}}^{-1}\left(\vec{R}\right) \hat{H} \Psi_{\mathrm{Trial}}\left(\vec{R}\right) \right\rangle_{P\left(\vec{R}\right)} \tag{2.13}$$

$$\approx \frac{1}{N} \sum_{j}^{N} \Psi_{\mathrm{Trial}}^{-1}\left(\vec{R}_j\right) \hat{H} \Psi_{\mathrm{Trial}}\left(\vec{R}_j\right) \tag{2.14}$$

として $E_{\mathrm{Trial}}$ を評価する。試行推定を系統的に改善するための方策としては、以下に述べる変分原理に基づく数値最適化による手法が挙げられる（変分モンテカルロ法）。

### 2.2.1 変分原理

(2.1) 式の固有関数 $\{\Psi_j\}_j^N$ は完全規格直交系をなし、任意の試行波動関数 $\Psi_{\mathrm{Trial}}$ は以下のように固有関数 $\{\Psi_j\}_j^N$ で展開できる [9]。

$$\begin{aligned} \Psi_{\mathrm{Trial}} &= C_0\Psi_0 + C_1\Psi_1 + \cdots \\ &= \sum_j C_j \Psi_j \end{aligned} \tag{2.15}$$

ここで、$\Psi_0$ は基底状態に対応する固有関数、$\Psi_1, \Psi_2, \cdots$ は励起状態に対応する固有関数である。(2.7) 式に (2.15) 式を代入すると、

$$E_{\mathrm{Trial}} = \frac{\int d\vec{R} \cdot \sum_j C_j^* \Psi_j^* \cdot \hat{H} \cdot \sum_k C_k \Psi_k}{\int d\vec{R} \cdot \sum_j C_j^* \Psi_j^* \cdot \sum_k C_k \Psi_k} \tag{2.16}$$

となる。(2.1) 式より $\hat{H}\Psi_i = E_i\Psi_i$ となるので、

$$\begin{aligned} E_{\mathrm{Trial}} &= \frac{\int d\vec{R} \cdot \sum_j C_j^* \Psi_j^* \cdot \sum_k C_k \hat{H} \Psi_k}{\int d\vec{R} \cdot \sum_j C_j^* \Psi_j^* \cdot \sum_k C_k \Psi_k} \\ &= \frac{\int d\vec{R} \cdot \sum_j C_j^* \Psi_j^* \cdot \sum_k C_k E_k \Psi_k}{\int d\vec{R} \cdot \sum_j C_j^* \Psi_j^* \cdot \sum_k C_k \Psi_k} \end{aligned} \tag{2.17}$$

と代入出来る。また、$\{\Psi_j\}_j^N$ は完全規格直交系であるため、$\Psi_j^*\Psi_k$ は $j = k$ の時のみ非 0 となる。よって、

$$E_{\mathrm{Trial}} = \frac{\sum_j E_j \cdot C_j^* C_j \cdot \int d\vec{R} \cdot \Psi_j^* \Psi_j}{\sum_j C_j^* C_j \cdot \int d\vec{R} \cdot \Psi_j^* \Psi_j} \tag{2.18}$$

$E_j$ はエネルギー準位を表し、$E_0 < E_1 < \cdots$ であるため、以下の式が成り立つ。

$$\begin{aligned} E_{\mathrm{Trial}} &= \frac{\sum_j E_j \cdot C_j^* C_j \cdot \int d\vec{R} \cdot \Psi_j^* \Psi_j}{\sum_j C_j^* C_j \cdot \int d\vec{R} \cdot \Psi_j^* \Psi_j} \\ &\geq \frac{\sum_j E_0 \cdot C_j^* C_j \cdot \int d\vec{R} \cdot \Psi_j^* \Psi_j}{\sum_j C_j^* C_j \cdot \int d\vec{R} \cdot \Psi_j^* \Psi_j} \end{aligned} \tag{2.19}$$

(2.19) 式を整理すると、

$$\begin{aligned}(2.19)\text{ 式} &= E_0 \cdot \frac{\sum_j C_j^* C_j \cdot \int d\vec{R} \cdot \Psi_j^* \Psi_j}{\sum_j C_j^* C_j \cdot \int d\vec{R} \cdot \Psi_j^* \Psi_j} \\ &= E_0 \end{aligned} \tag{2.20}$$

となる。つまり、

$$\begin{aligned} E_{\text{Trial}} &= \frac{\int d\vec{R} \cdot \Psi^* \left(\vec{R}\right) \cdot \hat{H} \Psi \left(\vec{R}\right)}{\int d\vec{R} \cdot \left|\Psi \left(\vec{R}\right)\right|^2} \\ &\geq E_0 \end{aligned} \tag{2.21}$$

という関係が成り立ち、この時、試行波動関数 $\Psi_{\text{Trial}}$ で評価した固有値 $E_{\text{Trial}}$ は基底状態の $E_0$ 以上かつ試行推定が厳密解に一致する時、$E$ と等しくなる（上界性）。簡単に言えば、より良い試行推定はより低い $E_{\text{Trial}}$ を与える。これを量子力学における変分原理と呼ぶ。

変分モンテカルロ法は、変分原理に基づいてエネルギー固有値 $E_{\text{Trial}}$ が小さくなるように試行波動関数 $\Psi_{\text{Trial}}$ を調整を繰り返し、固有関数 $\Psi$ を求める手法である。試行波動関数 $\Psi_{\text{Trial}}$ の調整方法としてはジャストロー因子と呼ばれる関数の付加や、バックフロー関数の付加などの手法がある [10] が、本論文ではそれらの詳細については割愛する。

### 2.2.2 メトロポリス法

(2.12) 式では、$P\left(\vec{R}\right)$ に従うサンプリング点列にて評価するとあるが、あるサンプリング点列が所望の確率分布に向かうためには、「確率過程が所与の定常状態分布 $P\left(\vec{R}\right)$ に向かうことを保証する」という詳細釣合いの条件を満たさなければならない。この条件を満たすようなサンプリング点列の更新法の一つにメトロポリス法がある。分布が $\vec{R}_\alpha$ から $\vec{R}_\beta$ へ遷移する確率を $T\left(\vec{R}_\alpha \to \vec{R}_\beta\right)$ と書くと、メトロポリス法は、$F\left(\vec{R}_\alpha \to \vec{R}_\beta\right) = F\left(\vec{R}_\beta \to \vec{R}_\alpha\right)$ となる確率過程を用いて次のように表される。

$$T\left(\vec{R}_\alpha \to \vec{R}_\beta\right) = \begin{cases} F\left(\vec{R}_\alpha \to \vec{R}_\beta\right) \cdot \left\{P\left(\vec{R}_\beta\right)/P\left(\vec{R}_\alpha\right)\right\} & for \quad P\left(\vec{R}_\beta\right) > P\left(\vec{R}_\alpha\right) \\ F\left(\vec{R}_\alpha \to \vec{R}_\beta\right) & for \quad P\left(\vec{R}_\beta\right) < P\left(\vec{R}_\alpha\right) \end{cases} \tag{2.22}$$

(2.22) 式は、$F\left(\vec{R}_\alpha \to \vec{R}_\beta\right)$ で試行更新を行ない、その結果、平衡に向かうならば採択、そうでない場合でも $\left\{P\left(\vec{R}_\beta\right)/P\left(\vec{R}_\alpha\right)\right\}$ の確率で採択することを意味する。具体的な $F\left(\vec{R}_\alpha \to \vec{R}_\beta\right)$ の与え方としては、乱数 $\xi \in (0,1)$ を用いて、

$$\vec{R}_\beta = \vec{R}_\alpha + \left(\xi - \frac{1}{2}\right) \cdot \Delta \tag{2.23}$$

にて試行更新し、$\left\{P\left(\vec{R}_\beta\right)/P\left(\vec{R}_\alpha\right)\right\} < \xi$ で結果を棄却する方策が一般的に取られる。

## 2.3 フラグメント分子軌道法

タンパク質やDNAなどの生体高分子と呼ばれる分子は系のサイズが大きく、扱う空間の大きさと変数の数により、量子モンテカルロ法による計算手法では現行の計算機でも計算を行なうことが困難な系である。多体相互作用が重要と考えられている過程も多く、かつ、これらは高速で簡便な密度汎関数法が難渋する問題として知られているため、量子モンテカルロ法の適用が期待される[11]。生体分子系は、1分子が空間の全ての領域において密に存在しているわけでなく、ある種の塊ごとに分布している。そのため、結合を上手く切断し、系を計算可能な小規模系に分割しようとするアプローチが行なわれてきた。その手法の一つとしてフラグメント分子軌道法（FMO：Fragment Molecular Orbital method）がある[3, 4]。

フラグメント分子軌道法では、大規模分子系を複数の小規模系（フラグメント）に分割し、それぞれのフラグメントで計算を行なった結果を最後に統合することで元の系のエネルギーを求めるというアプローチを行なう。フラグメントごとの計算を行なう際、当該フラグメント以外のフラグメントについては、静電場として近似し、外場として扱う。$i$番のフラグメントのエネルギーを$E_i$、二つのフラグメント$i$と$j$のペア（フラグメントペア）のエネルギーを$E_{ij}$とすると、$L$個のフラグメントに分割された系における全系のエネルギー$E_{\mathrm{All}}$は

$$E_{\mathrm{All}} \approx \sum_{i=1}^{L-1}\sum_{j=i+1}^{L} E_{ij} + (L-2)\sum_{i=1}^{L} E_i \tag{2.24}$$

と近似される[3, 4]。各フラグメント及びフラグメントペアのエネルギーは、(2.1)式により求めることが可能であり、他フラグメントからの寄与は、(2.1)式のポテンシャルエネルギー$V\left(\vec{R}\right)$の項に含まれる。より具体的には、ある$j$番のフラグメント上での電子のセット$\vec{R}^{(j)} = (\vec{r}_1, \cdots \vec{r}_N)$に対するハミルトニアン$\hat{H}^{(j)}$については、

$$\hat{H}^{(j)} = -\frac{1}{2}\sum_{l=1}^{N}\nabla_l^2 + \sum_{i}^{N}\sum_{i\neq j}^{N}\frac{1}{\left|\vec{r}_i - \vec{r}_j\right|} - \sum_{i}^{N}\sum_{\alpha}^{K}\frac{Z_\alpha}{\left|\vec{r}_i - \vec{r}_\alpha\right|} + \sum_{l=1}^{N} U_{\mathrm{ES}}(\vec{r}_l) \tag{2.25}$$

このように表される。ここで、右辺の第1項はフラグメント内の運動エネルギー、第2項はフラグメント内の電子間に働くクーロンポテンシャルであり、$\vec{r}_i$、$\vec{r}_j$は各電子の位置を表す。第3項は電子と原子核の間に働くクーロンポテンシャルで$\vec{r}_\alpha$は原子核の位置を表す。そして第4項が他フラグメントからのクーロンポテンシャルを表している。

$U_{\mathrm{ES}}(\vec{r})$は、

$$U_{\mathrm{ES}}(\vec{r}) = \sum_{m=1}^{M}\frac{\rho(\vec{r}_m)}{\left|\vec{r} - \vec{r}_m\right|} - \sum_{\beta=1}^{K}\frac{Z_\beta}{\left|\vec{r} - \vec{r}_\beta\right|} \tag{2.26}$$

と表される。ここで、$\rho(\vec{r})$は他フラグメント上の電荷密度を表し、$\vec{r}_m$は他フラグメントを$M$個のセルに分割した際の各々のセルの中心座標を表す。(2.26)式の右辺第1項は従っ

て他フラグメント上の電子からの寄与、第2項が他フラグメント上の原子核からの寄与を表す。

よって、(2.25) 式で表される $\hat{H}$ を用いた (2.14) 式で各フラグメントのエネルギーを統計評価し、(2.24) 式で全系のエネルギーを求めることがFMO-QMC計算の基本方針となる。ただし、全系のエネルギー $E_{\mathrm{All}}$ の分散 $\sigma^2_{\mathrm{All}}$ は量子モンテカルロ法によって統計評価された各フラグメントのエネルギー $E_i$ の分散 $\sigma^2_i$ およびフラグメントペアのエネルギー $E_{ij}$ の分散 $\sigma^2_{ij}$ の和である。よって、$L$ 個のフラグメントに分割された系のエネルギー $E_{\mathrm{All}}$ の分散 $\sigma^2_{\mathrm{All}}$ は、

$$\sigma^2_{\mathrm{All}} = \sum_{i=1}^{L-1} \sum_{j=i+1}^{L} \sigma^2_{ij} + (L-2)^2 \cdot \sum_{i=1}^{L} \sigma^2_i \tag{2.27}$$

となる。

量子モンテカルロ計算では $\vec{R}$ を更新するごとに $\hat{H}$ を再評価するが、FMO-QMCでは (2.25) 式に示したように、外場の影響 $U_{ES}$ を考慮する計算が含まれるため、通常の量子モンテカルロ計算に比べて計算速度が落ちる。

本研究では、この (2.26) 式の計算を高速に行なうための手法について勘案する。手法として、最初に思い浮かぶのは、(2.26) 式の計算を MPI や OpenMP を用いて並列化することであるが、量子モンテカルロ計算自体は MPI 並列による並列化によってリニアに高速化が可能であるので、よほど潤沢に並列化数が稼げる計算機でなければ、この手法は得策ではない。次に思い浮かぶ手法としては、外部アクセラレータを導入し、その上で (2.26) 式の計算を行なう手法が考えられる。本論文では、この外部ハードウェアに画像処理演算装置（GPU）を用いた高速化の手法について実装を行ない、実際に計算を行なった結果について報告する。

## 2.4 量子モンテカルロ計算コード

本研究では、量子モンテカルロ計算コードとして「CASINO」[12] という計算コードを利用する。量子モンテカルロ計算コードには他にも、「QMCPACK」[13] などがあるが、それらに比べ「CASINO」は計算対象や手法に関して汎用性の高い計算コードであり、周期系/孤立系/電子ガス系、平面波基底/ガウシアン基底、擬ポテンシャル計算/全電子計算といったものを単一の実行バイナリを持って取り扱うことが可能である。「CASINO」は現時点では FMO 法に対応していないため [12]、前園によって FMO 法を取り扱えるように拡張された「CASINO」計算コード（FMO-CASINO）を利用する [2]。

# 第3章　GPGPU

本章では、本研究で高速化手法として用いるGPGPUという技術についての概略と手法の特徴について述べる。

## 3.1　GPUとGPGPU

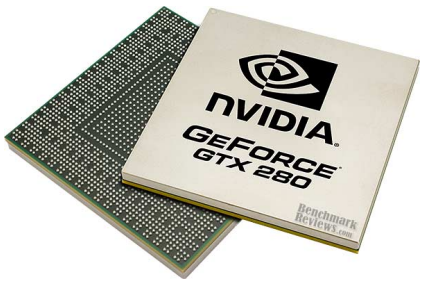


図 3.1: GTX 280 チップ

図 3.2: GTX280 グラフィックカード

GPGPU（General Perpose computing on GPU）とはGPUによる汎用計算を行う手法を指す。GPU（Graphics Processing Unit）とはその名の通り画像処理を専門に行なう演算装置であり、図3.1に示すような集積回路の形をしている。これを搭載した図3.2のボードがPC（パーソナルコンピュータ）などに搭載されている。PCやワークステーションの画像演算部、特に3Dグラフィックスで同時多発的に発生する頂点の座標変換やピクセルの陰影処理に必要なベクトル計算を行なっている3Dの計算処理は、単純な計算を大量に行なうことが多いため、その種の計算に特化したプロセッサをGPUは大量に載せている。例えば、NVIDIA社のGeForce GTX 275では1.404 GHzで動作するストリーミングプロセッサが240個搭載されている。

大量の単純計算に強いため、単精度浮動小数演算の計算能力については、コンシューマ向けのCPUを大きく上回る。近年のコンシューマ向け演算装置で例を挙げると、Intel社のIntel Core i7 975 Extremeは理論性能として55.36 GFLOPS [14] [1] であるのに対し、

[1] ▷ 7.3FLOPS：Floating point number Operations Per Second

NVIDIA 社の GeForce GTX 275 という GPU では理論性能が単精度で 1.02 TFLOPS、倍精度でも 84.24 GFLOPS と大きく上回っている。上記の GPU は高価な専用ハードウェアではなく、PC パーツ専門店に行けば、誰でも 2 万円程度でこの GPU が載ったグラフィックボードを入手することが可能である。また、GPU の性能はまだ発展途上の域にあり、2010 年上期には単精度で 2.46 TFLOPS、倍精度では 624 GFLOPS もの性能を持つ GPU が NVIDIA 社より発売される予定である [16]。

このようにある種の計算に対して特異な性能を発揮する GPU を、画像演算だけでなく汎用計算にも利用しようという考えが当然起きた。それが GPGPU の始まりである。そもそも、GPU 上で汎用計算が行なえるようになったのは比較的最近であり、制限はあるもののある程度自由なプログラムを行なうことが可能になった DirectX9 世代の GPU からである。また、2004 年 8 月には世界で初めての GPU に関する GPGPU 研究報告学会「GP2」が行なわれている [15]。しかし、当時はまだ GPGPU 向けの開発環境が整っておらず、GPU にテクスチャに対する 3D グラフィック演算に見せかけて汎用計算を行なうといった手法が採られていた。その後、sh [2] や BrookGPU [3] など、テクスチャなどの仕様に依存しない GPGPU 向けの開発環境が提案されたが、これら初期の頃の開発環境で制作されたソフトウェアは動作させる GPU に強く依存していたため、ソフトウェアを共有することが難しく普及に至らなかった。そのような状況の中、2006 年 11 月に業界初の標準的な開発環境として、NVIDIA 社が GPGPU 開発環境「CUDA [4] 」（Compute Unified Device Architecture）を正式発表した[5]。

## 3.2 CUDA：Compute Unified Device Architecture

「CUDA」は NVIDIA 社より提供される、NVIDIA 社製 GeForce 8 以降の GPU 上でシームレスかつスケーラブルな動作・実行を保証する GPGPU 開発環境の名称である。CUDA は「NVIDIA C Compiler」を中心とした C 開発環境セット「CUDA TOOLKIT」と、いくつかのサンプルコードやライブラリを含む開発者向け SDK「CUDA SDK」、NVIDIA GeForce シリーズなどの NVIDIA 製 GPU を汎用プロセッサとして扱うためのドライバ「CUDA ドライバ」の 3 点からなる。NVIDIA C Compiler はその名の通り、C 言語をベースとしたコンパイラであり、GPU の利用のために変数の型宣言や関数が拡張されている。CUDA は NVIDIA 社製 GeForce 8 以降の GPU 上でのみ実行可能という制限があるが、「コンパイラによって生成されたデバイスに依存しない中間コードを CUDA ドライバが動作させる GPU に適したネイティブコードに変換して実行する」という仕組みで動作するため、対応 GPU であれば GPU の世代や種類によらず単一のバイナリで GPU を利用した計算が可能と、従前の方式に比べ高い汎用性を持っている。

---

[2] ▷ sh：http://libsh.org/

[3] ▷ BrookGPU：http://graphics.stanford.edu/projects/brookgpu/

[4] ▷ CUDA：http://www.nvidia.com/object/cuda_home.html

[5] ▷ 7.4 CUDA 以外の GPGPU 開発環境

汎用計算用の外部アクセラレータとしてはGRAPEやFPGAなどがあるが、科学技術計算向けの専用機は設計コストが掛かるため、開発スパンが長く、高価である。グラフィックボードはPCにとって主要なパーツの一つであるため、常に開発が続けられ、比較的短い開発スパンで新しい製品が生まれている。また、大衆向けの製品であるために安価で大量に入手することが可能である。そのため、PCクラスタとの相性が非常に良い。対応GPUさえあれば誰にでも利用が可能であるため、ユーザによるコミュニティが大きく広がっている。

ただし、どのような計算コードでもGPUによる高速化が期待されるというわけではない。メインメモリとGPU上のデバイスメモリをつなぐバスが比較的低速であるため、CPUとGPUの間で頻繁にやり取りするようなコードはGPUによる高速化には適さない。なるべく受け渡すデータは少なく、デバイス上で行なう処理が多く、大規模な並列化が可能な計算がGPUによる高速化に適している。

### 3.2.1 CUDAの仕組みとスレッド管理

CUDAの計算コードはCPUを動作させる「ホストコード」とGPUを動作させる「デバイスコード」からなる。ホストコードには通常のC言語コードに加えて「GPU上で実行する関数をどの程度並列化するのか」などの初期設定やPC側のメモリ（ホストメモリ）からGPU上のメモリ（デバイスメモリ）へのデータ転送命令などが記述される。デバイスコードでは、GPU上で動作する関数（カーネル関数）の中身が記述される。ホストコードで指定された設定に従いカーネル関数がスレッド化されてGPU上で並列実行される。

現在販売されているCUDA対応GPUの内部には、図3.3に示すSM（Streaming Multiprocessor）という処理ユニットが複数、例えばGTX 275では30個、搭載されており、SMの内部には、SP（Streaming Processor）と呼ばれる最小単位の演算処理ユニットが8つずつ搭載されている。SPは単精度の実数演算までに対応し、倍精度実数演算についてはSMに1つずつ搭載されている倍精度演算器（DPU：Double Precision Unit）で行なう。このため、現行のGPUでは倍精度実数に関する演算能力が単精度に比べて8倍低くなる。GPUへの命令は、SMにより解読されてSPで実行されるが、SMは4サイクルに1度しか命令を解読出来ない。しかし、4サイクルの間同じ命令を発行し続けることが出来るので、解読した命令を4サイクルの間、8つのSPで実行する。そのため、32個のスレッドが「ウォープ（warp）」という単位で管理され、SIMD[6]型で実行している。同じ命令を32スレッドが実行するが、分岐命令の実行時にウォープ内に分岐が含まれる場合は、異なる分岐方向のスレッドは待機させて全ての分岐を実行する。そのため、ウォープ内の分岐が繰り返されると、ねずみ算式に実行する命令が増える。これはウォープダイバージェント呼ばれるが、GPUの性能低下を招くので、可能な限りウォープ内の分岐は減らす必要がある。

[6] ▷ SIMD：Single Instruction Multiple Data

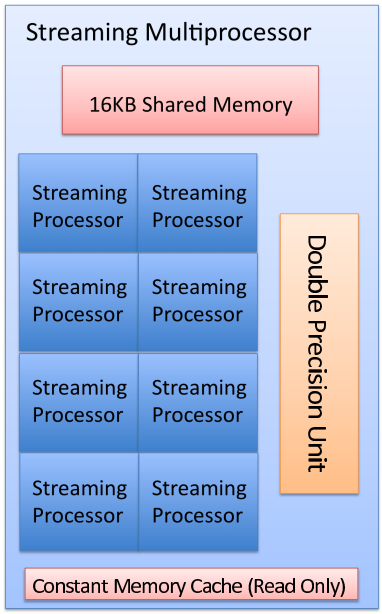


図 3.3: Streaming Multiprocessor の概念図

GPU では無数に発行出来るスレッドを管理するためにグリッドとブロックによる階層構造を用いる。図 3.4 のように、グリッドはブロックを二次元的に管理し、ブロックはスレッドを三次元的に管理する。また、全てのスレッドは同じカーネル関数を実行するが、SM のスペックによる制限などの理由で上手くグリッドとブロックに割り振る必要がある。このことについて詳しい話は §4.4 にて述べる。

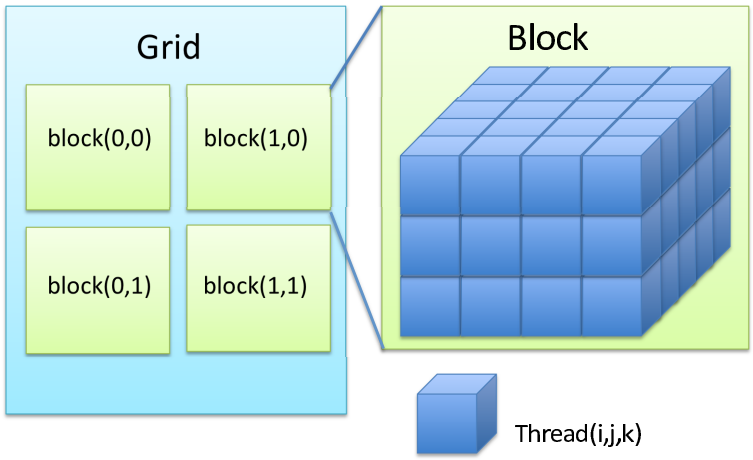


図 3.4: グリッドとブロックの概念図

### 3.2.2 CUDA で利用可能なメモリ

GPU 上のメモリは、グラフィックカード上に載っているメモリ（オフチップメモリ）と GPU のチップ上に載っているメモリ（オンチップメモリ）の二種類分けられ、更に用途やアクセス方法によって 6 種類に分けられる。オフチップメモリはホストプログラムからのアクセスが可能であり、大容量だが、低速である。オンチップメモリはホストプログラムからはアクセスできず小容量だが、非常に高速にアクセスが可能である。以下の表 3.1 に CUDA で扱える GPU メモリについて記す。

| メモリの種類 | メモリの場所 | cache | R/W | 使える範囲 | 保持する範囲 |
|---|---|---|---|---|---|
| レジスタ | オンチップ | − | R/W | 当該スレッド内 | スレッド実行中 |
| ローカルメモリ | オフチップ | 無し | R/W | 当該スレッド内 | スレッド実行中 |
| シェアードメモリ | オンチップ | − | R/W | ブロック内の全てのスレッド | 当該ブロック |
| グローバルメモリ | オフチップ | 無し | R/W | 全てのホストとスレッド | ホストが確保している間 |
| コンスタントメモリ | オフチップ | 有り | R | 全てのホストとスレッド | ホストが確保している間 |
| テクスチャメモリ | オフチップ | 有り | R | 全てのホストとスレッド | ホストが確保している間 |

表 3.1: CUDA で扱える GPU のメモリの種類

レジスタは GPU チップ上に実装されている高速に読み書き可能なメモリであり、主にカーネル関数上で宣言した変数がここに格納される。SM 毎に 16,384 個あり、レジスタが足りなくなった場合はローカルメモリにレジスタのデータを退避させて新しいデータを格納する。ローカルメモリはチップの外にあるため、レジスタに比べ 100 倍程度低速である。よって、なるべく余計な変数や配列を定義しないようにし、レジスタを節約することが必要である。シェアードメモリはチップ上の SM ごとに 16KB ずつ実装されているメモリであり、ブロック内の全てのスレッド間で共有することが出来る高速なメモリである。グローバルメモリはチップ外に実装されたメモリであり、ローカルメモリ同様、オンチップメモリと比べて 100 倍程度低速なメモリである。グラフィックカードのビデオメモリ上に実装されているので容量は製品によるが 512MB〜1GB と非常に大容量である。ホストコード上の大規模なデータはここに格納され、必要な分をシェアードメモリに上にロードして利用するのが基本指針となる。コンスタントメモリはチップ外に 64KB 実装されているが、SM ごとに設けられたコンスタントキャッシュにより全てのスレッドから高速に参照することが可能である。ただし、カーネル関数から書き込みは出来ない読み込み専用のメモリであるので、定数などに利用される。テクスチャメモリは画像処理に適した特殊なメモリであり、主にテクスチャユニットという 3D グラフィクスで使うテクスチャの参照を高速化するために使われる装置などで利用される。

# 第4章　実装方法

本章では、具体的にFMO-QMCの計算コードをCUDA計算コードに実装する方法や性能の比較方法について述べる。

## 4.1　実装に関する方針

まず、CUDA実装の対象となる「FMO-CASINO」[2]はFortranにより書かれた計算コードであり、無償で提供されるCUDA開発環境には現在、C言語のコンパイラしか付属していないため、CUDAに計算させたい部分をFortranコードを用いて拡張することは難しい。しかし、「CASINO」および「FMO-CASINO」は大規模な計算コードでかつ、50以上のコードが高度にモジュール化されており、その全容を把握して全てのコードをC、またはC++に移植するには多大な時間と労力を必要とする。更に、「CASINO」は現在も開発が積極的に行なわれている計算コードである[12]ので、全てを移植するメリットは小さい。有償のFortran言語およびC言語向けのコンパイラとしてPGI Accelerator Compilers[18]があり、これを利用する手もあるが、本研究では、コンパイラによって生成されるオブジェクトをリンクさせることで言語間の連携を可能にする複合プログラミングという技術を利用したコード拡張について扱う。複合プログラミングによって、CUDA上で実行したい部分のみC言語で記述し、Fortranコード上からC言語で書かれたCUDA関数を呼び出すという方策を採ることができる。

## 4.2　複合プログラミング

コンパイラはコードをコンパイルすることでオブジェクトを生成し、オブジェクト化したコードをリンクさせることで、実行バイナリを生成する。このオブジェクトがFortranコンパイラ、Cコンパイラの間で共有可能なものなので、これを利用することでFortranコードからCコードの関数を呼び出すことも、その逆も可能である。ただし、コード上の規約や制限により、いくつかの規則に沿ってコードを書く必要がある。主な注意点としては、

**甲)** Cコード側の関数名の最後に「_」を付加する必要がある

**乙)** 引数には参照渡し（ポインタ）のみが利用可能

**丙)** 多次元行列の要素のアドレスが異なる

の 3 点が挙げられる。（甲）はオブジェクトの中での関数名の与え方がC 言語と Fortran 言語で異なることによる。（乙）の制限については Fortran 言語の仕様に起因する。（丙）は、Fortran 言語では列優先順に配列要素が格納され、C 言語では行優先順に配列要素が格納されるという仕様の違いによって注意すべき問題である。

具体的な Fortran コードからの C 言語関数呼び出し処理を行なっているコードを抜粋し図 4.1 に記す。また、CUDA 計算コードを Fortran コードから呼び出す「CUDA カーネルに対する FORTRAN インターフェイス」というサンプルコードが NVIDIA 社の CUDA 公式サイトに公開されており [19]、CUDA と Fortran による複合プログラミングの参考としてわかりやすい。

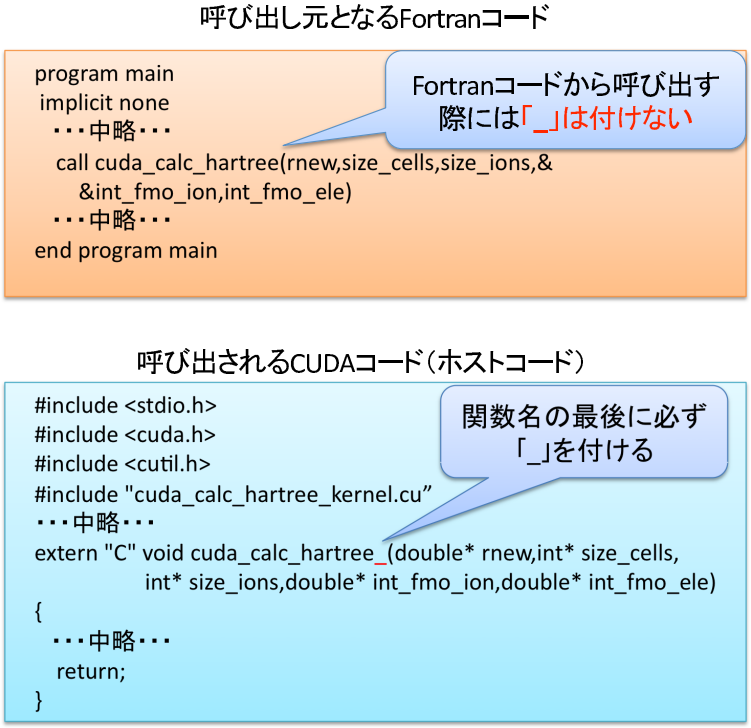


図 4.1: Fortran コードから C 言語で書かれた CUDA 関数の呼び出し（抜粋）

## 4.3 具体的な実装箇所と方策

§4.1〜§4.2 にて、CUDA による GPU 計算を「FMO-CASINO」へ実装する具体的な方策として、「必要な部分を関数化し、部分的に実装する」という方針は決まったが、具体的に「FMO-CASINO」のどこの計算を CUDA 関数で行なうかは述べていない。本節では、「FMO-CASINO」の中で、どのような計算を CUDA 関数上に実装するのか述べる。

§2.3 にて、「FMO-CASINO」の計算律速が（2.25）式の外場からの寄与を表す $U_{\mathrm{ES}}$ を計算する過程で起きていると述べた。つまり、各電子に対して行なわれる（2.26）式の評価を GPU を用いて CPU に比べて高速に計算を行なうことが出来れば、「FMO-CASINO」の高速化が可能となる。よってまずは、この計算を行なっているコードを検証し、CUDA 関数として実装した場合に高速化が見込めるか検討する。

「CASINO」および「FMO-CASINO」はモジュール化が進んだコードであり、処理ごとに関数が設けられ、開発者にとって読み下しやすいコードとなっているため、目的のコードを見つけることは容易である。「FMO-CASINO」では、電子の配位セット $\{r_j\}_{j=1}^{N}$ の更新を行なった後、各電子に対して順に（2.25）式を再評価している。ある更新した電子の配位 $\vec{r}_{\mathrm{new}}$ に対する（2.26）式の評価は $\vec{r}_{\mathrm{new}}$ を引数とする *calc_hartree* という関数として実装されている。以下の図 4.2 に *calc_hartree* 関数のフローチャートを示す。

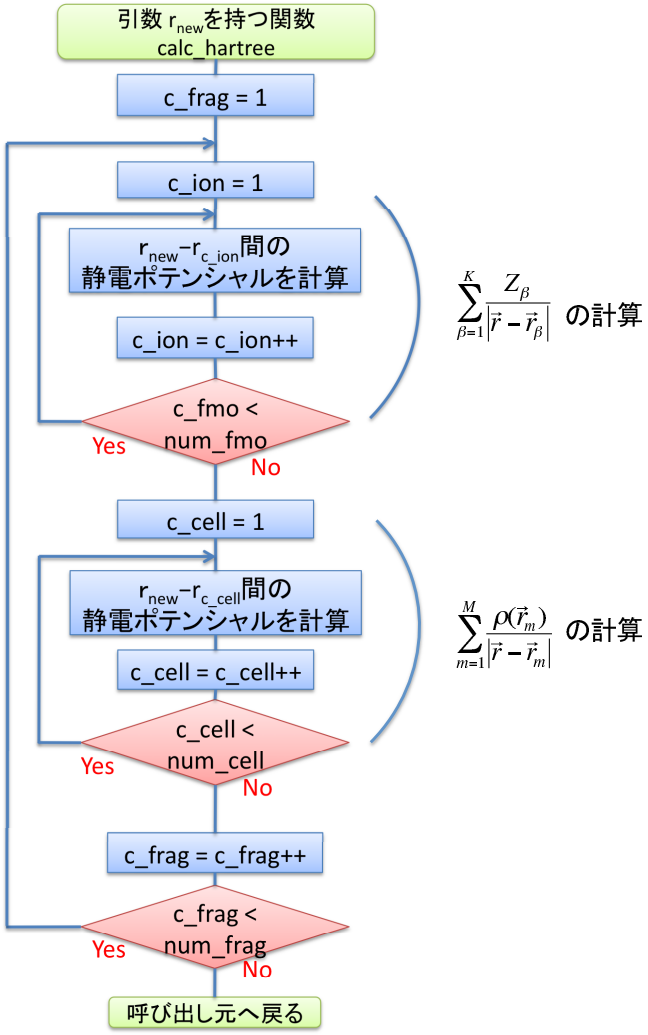


図 4.2: *calc_hartree* 関数のフローチャート

図 4.2 のフローチャートにおいて、*c_frag*、*c_ion*、*c_cell* はカウンタであり、それぞれ $\vec{r}_{new}$ が属していない他フラグメント、他フラグメント内の原子核、他フラグメントの電荷密度を表すセルを意味する。*num_frag*、*num_ion*、*num_cell* はそれらの最大値を表す。*num_ion* は他フラグメント内の原子核を意味しているので、多くとも 3 桁のオーダーで収まるが、*num_cell* は他フラグメントを無数のセルに分割した合計が入るので、非常に大きい。今回計算対象として扱う系のフラグメントでは、原子核が 19 個であるのに対して総セル数は 912,673 となっている。ただし、電荷密度が非常に小さく結果に影響をほとんど与えないセルは「FMO-CASINO」の事前計算により除外されているため、実際に参照するセルは 268,782 となる。セルの配位と電荷密度のデータは、GPU に転送するデータの中では大きなものとなるが、これらは更新された $\vec{r}_{new}$ に関わらず一定で、値を参照されるだけのデータであるので、「FMO-CASINO」の初期設定時に GPU 上のグローバルメモリへデータを転送しておけば、同じデータを再利用できる。関数呼び出しごとに必要な CPU-GPU 間のメモリ転送は更新した $\vec{r}_{new}$ と計算結果のみなので、通信コストは計算コストに対して十分小さくなることが予見出来る。

原子核に対する計算とセルに対する計算は互いに独立して行なうことが可能であり、また、1 つ 1 つのセルに対する計算も独立に実行可能なので、並列化による高速化が有効だと予見出来る。

これらの見地より、(2.26) 式の評価関数を CUDA 関数にて実装することは有効であると判断出来る。また、原子核ポテンシャルの計算にかかる計算コストはセルのそれに比べて十分小さく、GPU 上でセルに対する計算を行なっている間に空き状態となる CPU を使って計算を行なっても問題ないと思われる。よって、原子核に対する計算は CPU 上で GPU と並行して行なう。*calc_hartree* 関数の GPU 対応版、*cuda_calc_hartree* 関数と名付けるこの CUDA 関数は以下の図 4.3 に記すフローチャートのイメージで実装する。

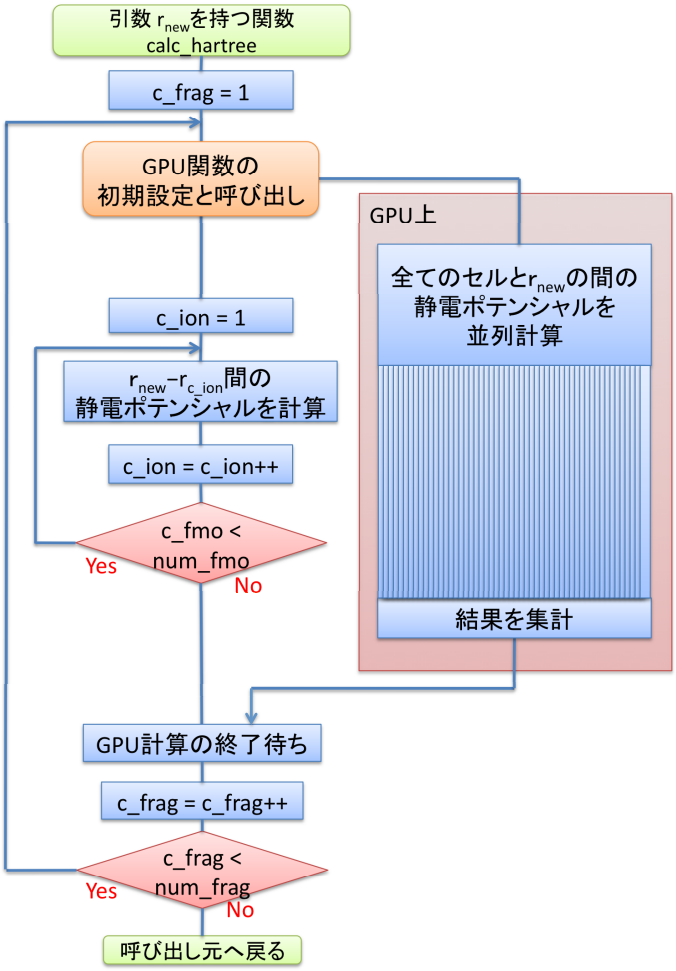


図 4.3: *cuda_calc_hartree* 関数のフローチャート

## 4.4 ブロック数およびスレッド数の調整

実装する *cuda_calc_hartree* 関数のフローチャートは §4.3 にて述べたので、残るは GPU 上で動作するカーネル関数に関する問題となる。計算対象のセルが全て独立に計算出来るので、スレッドを数十万生成して計算させる、ということも理論上は可能であるがグローバルメモリからのロードなどのデータ転送時間に対して計算時間が短くなりすぎ、効率が悪くなると予想される。また、スレッド数が増えすぎると、SM 当たりに利用可能なシェアードメモリやレジスタの制限を超えてしまい、実行効率が下がることが予想される。カーネル関数に設定するブロック数、スレッド数には、値に関して具体的な指針があるので、それに従い設定を行なう。まず、ブロック数であるが、1 つのブロックは SM をまたぐ事は出来ず、1 つの SM には「リソースに余裕があれば」複数のブロックを割り当てることが可能（同時に実行される）、という制限がある。1 つのブロックに可能な限

りスレッド数を設定した場合は 1 つの SM に対して 1 つのブロックが割り当てられることとなり、SM の総数以上にブロックが設定された場合は、バレル式に実行するブロックを次々に切り替えて複数のブロックを並列に実行する。よって、ブロック数は SM の総数の倍数で与えることが指針となる。スレッド数については、§3.2.1 にも述べたように 32 スレッドがウォープ単位で実行されるため、32 の倍数を基本とし、更にメモリ入出力の待ち時間の影響を考慮する（メモリレイテンシの隠蔽）と 64 の倍数が良しとされる。また、1 つの SM で利用できるシェアードメモリとレジスタの総数にはそれぞれ 16KB と 16,384 個という制限があり、1 つのブロックが利用するそれらの総数が SM の制限を超えると、実行効率が極端に下がる。よって、実装するカーネル関数に合わせてスレッド数を決定する必要がある。これらがスレッド数に関する指針である。カーネル関数が利用するレジスタ数やシェアードメモリのサイズは、「CUDA SDK」に付属する CUDA プロファイラを利用することで知ることが出来る。また、レジスタ数やメモリの使用状況、ブロック辺りのスレッドから、GPU 資源の占有率を割り出すツール [20] があり、これらも参考し、ブロック数やスレッド数を調整する。

本論文では、ブロック数については利用する GPU(GTX 275) の SM の総数 30 に設定し、1 ブロック辺りのスレッド数は 128 に設定した。よって、30 × 128、3840 のスレッドを GPU 上で並列に実行する。

## 4.5 利用する計算資源

本論文の開発・実験には、以下に記すスペックおよび環境の計算機を 4 台繋いだ PC クラスタを利用する。

| | |
|---|---|
| CPU | Intel Core i7 920 2.66 GHz(Max 2.80 GHz) |
| GPU | GeForce GTX 275 |
| マザーボード | ASUS RAMPAGE II GENE |
| メモリ | DDR3-1333 2GB × 6 |
| OS | Linux Fedora 10 |
| CUDA | CUDA version 2.3 |

表 4.1: 計算機のスペック

ただし、CPU の Hyper-Threading 機能 [21] は BIOS 上からオフに設定し、4 コアの CPU として使用する。また、上記の GPU、GeForce GTX 275 の詳細なスペックは以下の表 4.2 の通りである。ここで、「Compute Capability」の項目はハードウェアレベルでの CUDA サポートバージョンを表し、「1.3」以上は倍精度実数演算のサポートを意味する。

| | |
|---|---|
| Compute Capability | 1.3 |
| グローバルメモリサイズ | 895 MB |
| SM の総数 | 30 |
| SP の総数 | 240 |
| SP のクロック | 1.404 GHz |
| コンスタントメモリサイズ | 64 KB |
| シェアードメモリサイズ | 16 KB per block |
| ウォープサイズ | 32 |
| スレッドの最大数 | 512 per block |
| メモリバンド幅 | 127 GB per sec. |

表 4.2: GPU のスペック

## 4.6 計算の対象と評価方法

実際に「FMO-CASINO」に *cuda_calc_hartree* 関数を実装し、CPU のみで計算する場合と、GPU も利用して計算を行ない、その比較を行うが、本研究では、計算の比較対象を行なうための対象系として、生体分子系の中では小規模で、ベンチマークに適した系であるグリシン三量体を設定する。また、本論文の性能評価では、グリシン三量体を FMO 法で分割した際の最小フラグメントに対して、CPU、GPU のそれぞれでエネルギー計算を行ない、その計算時間と計算精度についての結果を比較検討する。また、1 ノード上で計算を行なった場合と MPI 並列により 4 ノードで計算を行なった場合、CPU1 台のコア全てを使った計算と GPU1 台の計算の比較も行なう。

# 第5章　計算結果と考察

本章では、実際に *cuda_calc_hartree* 関数を「FMO-CASINO」に実装し[1]、変分モンテカルロ計算を行なった結果について述べ、それに対する考察と改善案について述べる。

## 5.1　計算結果

§4.6 にて述べた実験を行なった結果、以下の結果が得られた。まず、MPI 並列を行なわなず単体プロセスにて「FMO-CASINO」を実行した結果、計算時間については、従来通り CPU のみで「FMO-CASINO」を行なった計算（以後、CPU 計算とする）では 342.1*sec.*、*cuda_calc_hartree* 関数を利用し、CPU と GPU で行なった計算（以後、CPU+GPU 計算）では 48.7*sec.* という結果となり、GPU を利用することで約 7 倍高速化され、単体プロセス当たりの計算速度が飛躍的に向上した（図 5.1）。計算精度についても、CPU 計算と CPU + GPU 計算の間に差異は $\pm 1.0 \times 10^{-12}$ 程度しか表れず、統計誤差に吸収され計算結果には影響を与えないと思われる。また、MPI 並列を用いて 4 プロセスに分割し、CPU に搭載された 4 つのプロセッサコア全てを使って CPU 計算を行なった場合、その計算時間は 87.0*sec.* と単プロセスと比べて高速に計算できるが、この場合と比較しても CPU+GPU 計算にて計算の方が 1.78 倍高速である（図 5.2）。MPI 並列を利用して、2 ノード 2GPU にて 2 プロセスで計算した場合は、CPU 計算は 171.18*sec.*、CPU+GPU 計算では 22.83*sec.* となり、同様に 4 ノード 4GPU にて 4 プロセスで計算した場合は、CPU 計算は 171.18*sec.*、CPU+GPU 計算では 22.83*sec.* と、計算ノードの数に関わらず、約 7 倍の性能比という結果となった（図 5.3、図 5.4）。図 5.5 に 1、2、4 ノードの計算結果をまとめた。図 5.5 から、CPU 計算、CPU+GPU 計算の両方で計算時間が計算ノードの数に応じてリニアに減少していることが分かる。一定の性能比が維持されてることからも分かるが、*cuda_calc_hartree* 関数を実装したことによって、従来の「FMO-CASINO」の MPI 並列による高速化に対して悪影響を与えずに、単体プロセス辺りの計算速度の向上に成功していることが分かる。

[1] ▷ 7.5 *cuda_calc_hartree* 関数のプログラムコード

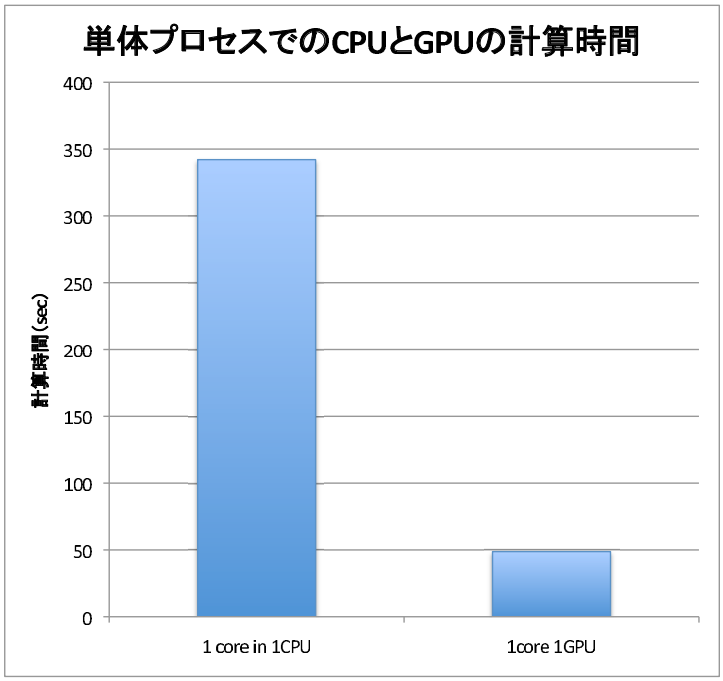


図 5.1: 1 プロセスでの CPU 計算と GPU 計算の計算時間比較

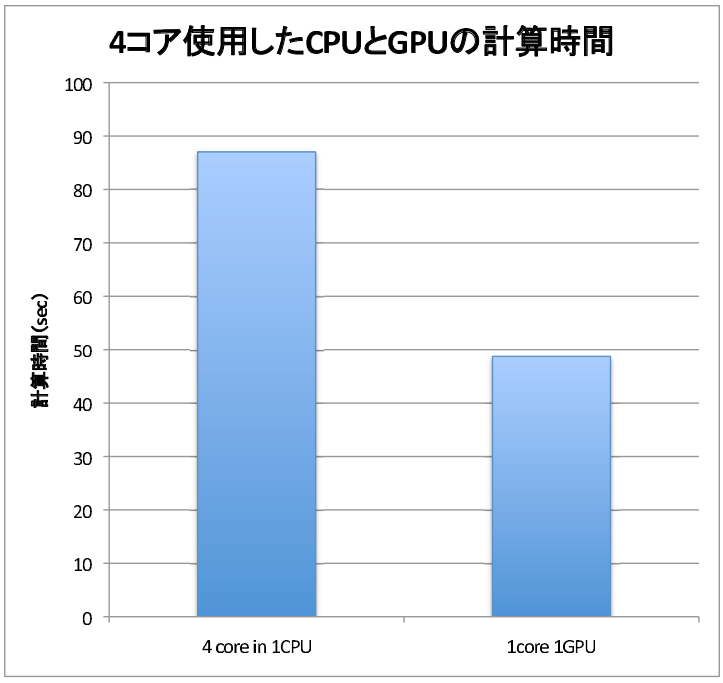


図 5.2: 4 プロセス CPU 計算と 1 プロセス GPU 計算の計算時間比較

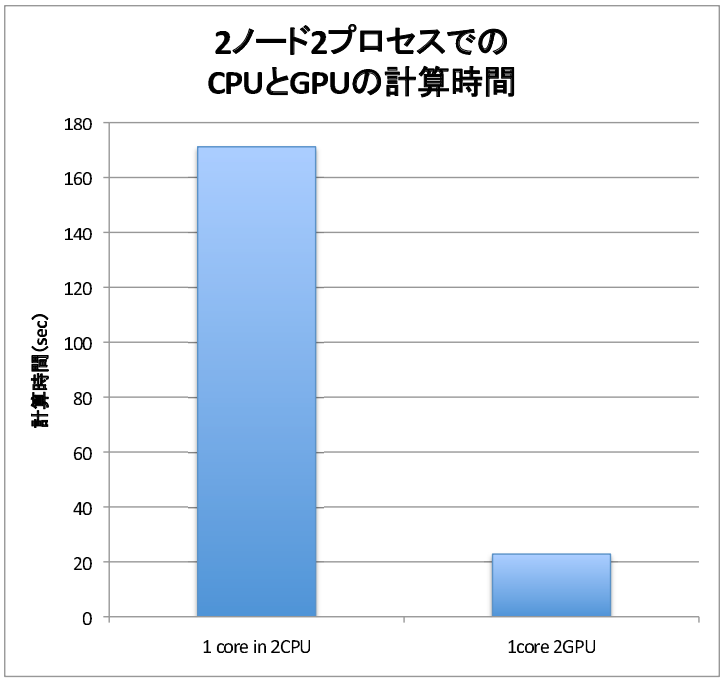


図 5.3: 2 プロセス CPU 計算と GPU 計算の計算時間比較

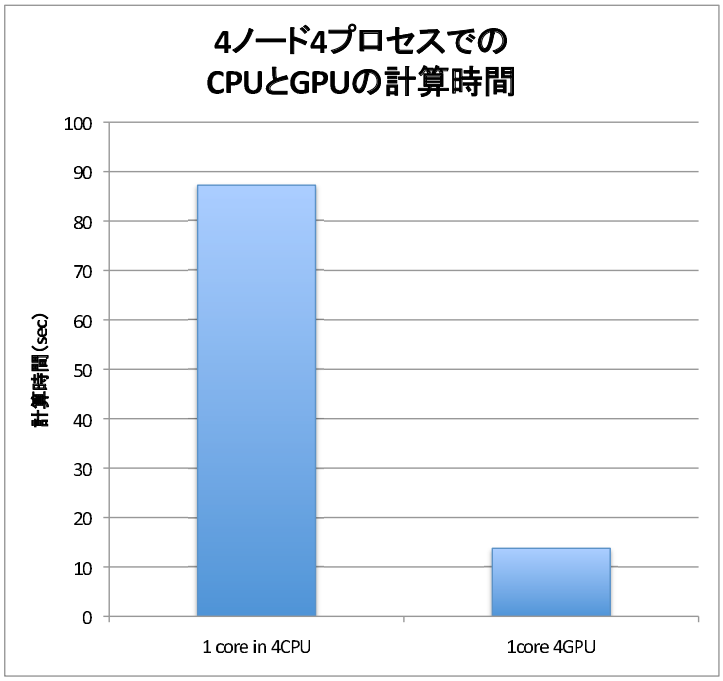


図 5.4: 4 プロセス CPU 計算と GPU 計算の計算時間比較

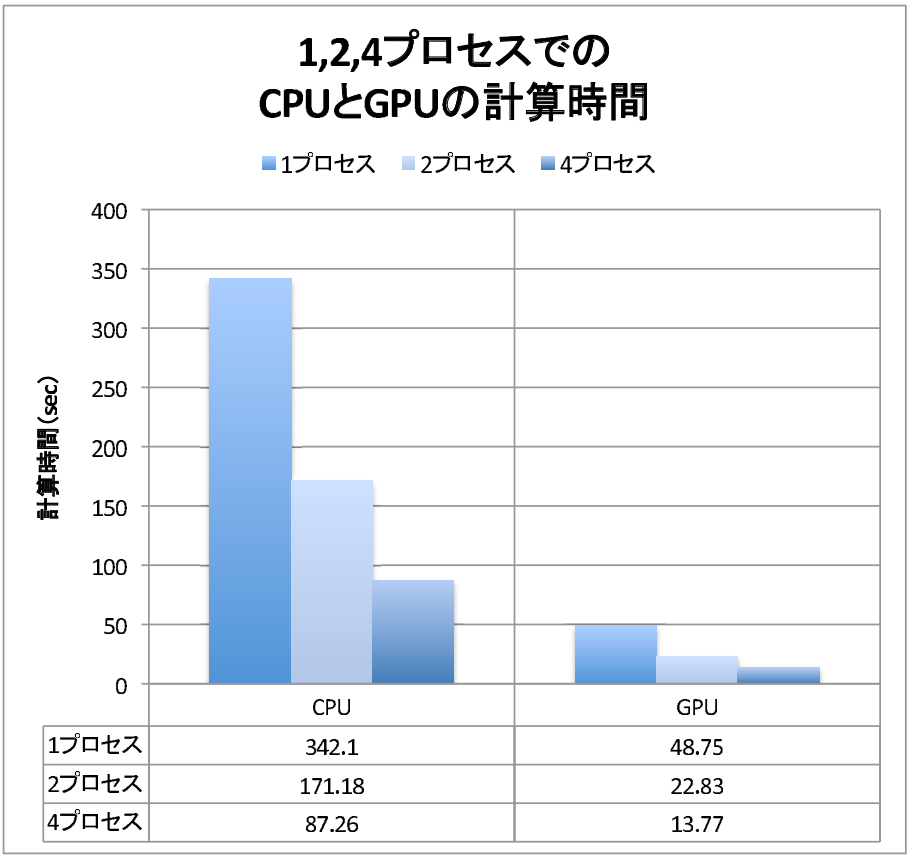


図 5.5: プロセス数に対する計算時間の比較

## 5.2 結果の考察・改善点

§5.1 に示した通り、「FMO-CASINO」の計算に GPGPU を取り入れることで、プロセス辺りの計算時間を 7 分の 1 に短縮することに成功した。また、CPU のコア 4 つ全てを利用した場合と比較しても、GPU を利用した計算の方が 1.78 倍高速である。今回使用した CPU：Intel Core i7 920 の理論性能は 44.8 GFLOPS であり、GPU：GeForce GTX 275 の倍精度演算に関する理論性能は 84.24 FLOPS となっており、上記の比にほぼ一致する。このことから、*cuda_calc_hartree* 関数が GPU の性能を最大に引き出して計算を行なっていると考える。現行の NVIDIA 製 GPU では倍精度演算の性能が単精度の性能に比べて 8 分の 1 と低いが、次世代の GPU ではその性能比が 1:4 程度まで改善され、624GFLOPS 程度まで引き上げられる [16] とのことなので、次世代の GPU に換装することだけでも大きく性能の改善が見込める。また、今回の実装では元の「FMO-CASINO」計算コードの記述に従い、倍精度実数にて計算を行なったが、*cuda_calc_hartree* 関数について、単精度実数に精度を落として計算した場合でも結果の精度に影響しない部分を単精度実数で計算することで、現状の GPU でも大幅な性能改善が見込める。ただし、単精度実数にて計算を行

なう場合、精度面で結果に影響が起きないように注意して議論しておく必要がある。他にも、現状 1 台の PC に対して 1 台の GPU を載せて計算しているが、1 台の PC に搭載可能な GPU は多くとも 3 台までである。そのため、GPGPU を利用して「FMO-CASINO」を計算実行する際、いくつかの CPU プロセッサコアが空き状態となる。よって、空いているプロセッサコアの上で「FMO-CASINO」内の独立した別の計算を行なうことで、単体プロセス辺りの計算時間を更に短くすることが可能と思われる。

# 第6章　結論

本論文では、フラグメント分子軌道法と量子モンテカルロ法を組み合わせた計算手法の高速化に、画像演算処理装置（GPU）を用いた汎用計算手法を適用し、単体プロセスの性能向上を目指した。フラグメント分子軌道法は、大規模系をより小さな系へ分割し、各々の計算結果を再統合することで元の系のエネルギーを計算する手法であり、生体高分子のような、そのままでは第一原理計算の適用が難しい系を適用可能にする手法である。量子モンテカルロ法は電子の多体相互作用の評価に関して高い信頼性を持つ手法であるが、フラグメント分子軌道法と組み合わせると、他フラグメントからの寄与を計算するために通常の量子モンテカルロ計算に比べて 50 倍もの計算時間がかかるようになる。この速度低下を解決する手法の 1 つとして、上記の計算を GPU 上で行なう手法を提案し、実際に実装し性能評価を行なった。

その結果、計算精度を維持しつつ、従来の計算速度に比べて、単体プロセス辺り約 7 倍の高速化に成功した。CPU のみでは計算ノード 1 台につき、4 プロセスまで同時に計算出来るが、本手法による高速化では、GPU を利用する制限上、計算ノード 1 台につき、多くとも 3 プロセスまでしか動作出来ないが、単体プロセスでも、4 プロセスの従来の計算に対して本手法による計算の方が 1.78 倍程度高速である。これらの性能比は演算装置の倍精度実数演算能力に依存しており、次世代 GPU の出現によって、本手法による計算の更なる速度向上が期待出来る。また、GPU を利用するプロセスは 1 つに限られるが、他の CPU プロセッサコアは利用可能なので、空いているプロセッサコアに独立した計算を行なわせることで、高速化が期待される。現行の NVIDIA 製 GPU は単精度実数演算器が倍精度実数演算器の 8 倍も実装されているため、単精度演算能力が非常に高い。仮に上記の計算を単精度実数にて行なうことが可能であれば、計算速度が演算能力の比で向上しているので、更に 8 倍の高速化が可能と考えられる。当然、精度を落とすことで最終的な計算精度に影響が起きる可能性があるため、慎重に扱う必要があるが、一部を単精度実数にて計算できるように改良するだけでも計算速度を幾許か引き上げられる可能性が考えられる。

# 第7章　付録

## 7.1　原子単位系

原子単位系は量子物理学などで数式を簡潔に表現するために用いられる単位系であり、電荷 $e$、電子の質量 $m_e$、プランク定数 $\hbar$、ボーア半径 $a_0$ を 1 とする単位系である。本論文では、エネルギーの基本単位として、水素原子から電子 1 つを取り除く時に必要なエネルギーを $1hartree$ とするハートリー原子単位系を利用している。ハートリー原子単位系では、上記の単位を一つの単位 a.u.(atomic unit) で表す。それぞれの具体的な値は以下のように換算される。

$$
\begin{aligned}
1a.u. &= e = 1.6022 \times 10^{-19}(C) \\
1a.u. &= m_e = 9.1095 \times 10^{-31}(kg) \\
1a.u. &= \hbar = \frac{h}{2\pi} = 1.0546 \times 10^{-34}(J \cdot s) \\
1a.u. &= a_0 = \frac{4\pi \cdot \varepsilon_0 \cdot \hbar^2}{m_e \cdot e^2} = 5.2918 \times 10^{-11}(m) \\
1hartree &= \frac{e^2}{4\pi \cdot \varepsilon_0 \cdot a_0} = 4.3598 \times 10^{-18}(J)
\end{aligned}
$$

## 7.2　昨今の大規模計算機

世界の最も高速なコンピュータシステムを LINPACK [1] のベンチマークによりランク付けする評価プロジェクトとして、TOP500[22] というプロジェクトがある。5 年前、TOP500 の首位は、米ローレンス・リバモア国立研究所が所有する 65,536 コアの並列計算機「BlueGene/L」であり、そのピーク演算性能は 183.50 TFLOPS であった。最新版の 2009 年 11 月のリストによると、現在の首位は米オークリッジ国立研究所が所有する Cray XT5 によるシステム（224,162 コア）であり、そのピーク演算性能は 2331 TFLOPS と、TOP 性能は年を追って上昇し続けている。近年の TOP500 の大規模並列計算機は並列化度数を高めることによる計算機の性能を図っており、一般の研究者が利用する計算機も同じ傾向を持っている。本学に 2009 年まで導入されていた Cray XT3（AMD Opteron150 2.4 GHz × 4 × 90：360 コア）から、XT5（Quad-Core AMD Opteron 2.4 GHz (Shanghai) × 2 × 256：2048 コア）へ変更され、一つのジョブが可能な並列度が格段に上昇している。

[1] ▷ LINPACK：http://www.netlib.org/linpack/

並列化技術とチップ設計技術の向上により、コア辺りのコストが安くなっているので、近い将来、研究室単位で 1,000 コア、10,000 コアクラスの並列計算機が設置される日も近いと思われる。

## 7.3 FLOPS

FLOPS（Floating point number Operations Per Second）とは、演算装置が持つ、「1 秒間に可能な実数演算の回数」を意味し、以下の式により与えられる：

$$(1\text{ クロック毎の実数演算処理数}) \times (\text{クロック数}) \times (\text{プロセッサコア数}) = (\text{FLOPS})$$

で求められる。現行の Intel CPU では 1 クロック毎に積和算命令を 2 回行なう。積算と和算の 2 つの命令を同時に 2 回行なうので実数演算 4 回分と換算される。今回使用した CPU：Intel Core i7 920 では、最大 2.80 GHz で動作する 4 つのコアが内蔵されているので、

$$4 \times 2.80 \times 4 = 44.8(\text{GFLOPS})$$

と算定される。GPU：GeForce GTX 275 については、1 クロックに 1 回の積和算を倍精度実数演算で実行可能な DPU が SM 毎に 1 つ、計 30 個実装されており、1.404 GHz で動作するので、倍精度実数演算に対する理論性能は、

$$2 \times 1.404 \times 30 = 84.24(\text{GFLOPS})$$

と算定される。単精度実数演算については、SM 毎に 8 個づつ計 240 個実装されている SP が、1 クロックごとに 1 回の積和算と積算の 2 命令を行なうので、

$$\begin{aligned} 3 \times 1.404 \times 240 &= 1010.88(\text{GFLOPS}) \\ &= 1.01(\text{TFLOPS}) \end{aligned}$$

と算定され、倍精度実数演算に対する理論性能に比べて非常に高い。

## 7.4 CUDA 以外の GPGPU 開発環境

現在、コンシューマ向けの GPU 生産メーカーとして NVIDIA 社と双璧をなしている会社として、AMD 社（旧：ATI 社、2006 年に AMD 社により買収）がある。AMD 社も後発ながら 2008 年に GPGPU 向けの統合環境として FireStream SDK を公開している。GPGPU 言語の一つである Brook を源流としており、CUDA 同様 C 言語をベースとしているので、統合環境として CUDA と並び立つことが期待されていたが、後発であったことや、初期のリリースでは汎用計算用の高価なデバイス（AMD FireStream シリーズ）専用の開発環境であったことなどの要因により、広く普及するには至らなかった。その後、ATI Stream

と名前を改め、AMD 製のコンシューマ向け GPU 上でも動作する環境が整ったが、現在は AMD 製デバイス専用の開発環境の構築ではなく、プロセッサメーカーによる束縛のない開発環境への対応に注目しているようである。これについては NVIDIA 社も同様の考えであるようで、両者共、DirectX Compute Shader や OpenCL という統合開発環境への対応を明言している。このため、近い将来、GPU のプロセッサメーカーに依存しない GPGPU 開発環境が出現すると思われる。

## 7.5 *cuda_calc_hartree* 関数のプログラムコード

今回作成した *cuda_calc_hartree* 関数のホストコードおよびデバイスコードについて以下に記す。

**ホストコード**

```
1  #include <stdio.h>
2  #include <cuda.h>
3  #include <cutil.h>
4  #include "cuda_calc_hartree_kernel.cu"
5
6  double* fmo_ion_pos;
7  double* fmo_ion_charge;
8  double* d_fmo_cell_centre;
9  double* d_fmo_cell_charge;
10
11 extern "C" void cuda_fmo_setup_
12          (double* h_fmo_ion_pos,double* h_fmo_ion_charge,
13           double* h_fmo_cell_centre,double* h_fmo_cell_charge){
14   int fmo_cellCentreSize = sizeof(double)*268782*3*1;
15   int fmo_cellChargeSize = sizeof(double)*268782*1;
16
17   CUDA_SAFE_CALL(
18   cudaMalloc( (void**) &d_fmo_cell_centre,fmo_cellCentreSize));
19   CUDA_SAFE_CALL(
20   cudaMemcpy( d_fmo_cell_centre,h_fmo_cell_centre,
21                 fmo_cellCentreSize , cudaMemcpyHostToDevice) );
22
23   CUDA_SAFE_CALL(
24   cudaMalloc( (void**) &d_fmo_cell_charge,fmo_cellChargeSize));
25   CUDA_SAFE_CALL(
26   cudaMemcpy( d_fmo_cell_charge,h_fmo_cell_charge,
```

```
27                fmo_cellChargeSize , cudaMemcpyHostToDevice) );
28
29    fmo_ion_pos = h_fmo_ion_pos;
30    fmo_ion_charge = h_fmo_ion_charge;
31
32    return;
33  }
34  extern "C" void cuda_fmo_finalize_(){
35    CUDA_SAFE_CALL(cudaFree(d_fmo_cell_centre));
36    CUDA_SAFE_CALL(cudaFree(d_fmo_cell_charge));
37    return;
38  }
39  extern "C" void cuda_calc_hartree_
40                  (double* rnew,int* size_cells,int* size_ions,
41                    double* int_fmo_ion,double* int_fmo_ele)
42  {
43    const int ThAryElem = 128;
44    const int GriAryElem = 30;
45
46    int_fmo_ion[0] = 0.0;
47    int_fmo_ele[0] = 0.0;
48
49    double int_fmo_ele_buf[30];
50    for(int i=0;i<30;i++){int_fmo_ele_buf[i]=0.0;}
51    double* d_int_fmo_ele_buf;
52    CUDA_SAFE_CALL(
53    cudaMalloc( (void**) &d_int_fmo_ele_buf,sizeof(double)*30 ));
54    CUDA_SAFE_CALL(
55    cudaMemcpy( d_int_fmo_ele_buf,int_fmo_ele_buf,
56                  sizeof(double)*30 , cudaMemcpyHostToDevice) );
57
58    CUDA_SAFE_CALL(
59          cudaMemcpyToSymbol(c_rnew, rnew, sizeof(double)*4));
60
61    dim3     grid( GriAryElem, 1, 1);
62    dim3  threads( ThAryElem, 1, 1);
63    cuda_sum02_Kernel<<< grid, threads>>>
64      ( d_fmo_cell_centre, d_fmo_cell_charge, d_int_fmo_ele_buf);
```

```
65
66    double dist_fmo[4];
67    for (int ifmo=0 ;ifmo < size_ions[0];ifmo++){
68      for (int i=0; i<3; i++){
69        dist_fmo[i] = rnew[i] - fmo_ion_pos[3*ifmo+i];
70      }
71      dist_fmo[3] = sqrt( dist_fmo[0]*dist_fmo[0] +
72                          dist_fmo[1]*dist_fmo[1] +
73                          dist_fmo[2]*dist_fmo[2] );
74      int_fmo_ion[0] -= fmo_ion_charge[ifmo]/dist_fmo[3];
75    }
76
77    CUDA_SAFE_CALL( cudaMemcpy(int_fmo_ele_buf,d_int_fmo_ele_buf,
78                    sizeof(double)*30, cudaMemcpyDeviceToHost) );
79    for(int i=0;i<30;i++){ int_fmo_ele[0] += int_fmo_ele_buf[i];}
80
81    CUDA_SAFE_CALL(cudaFree(d_int_fmo_ele_buf));
82    return;
83  }
```

**デバイスコード**

```
1   __constant__ double c_rnew[4];
2
3   __global__ void cuda_sum02_Kernel(
4                   double* fmo_cell_centre,double* fmo_cell_charge,
5                   double* g_int_fmo_ele_buf)
6   {
7     __shared__ double dist_fmo[128*3];
8     __shared__ double s_int_fmo_ele_buf[128];
9
10    double dist_fmo4;
11
12    const unsigned int tnum =
13                  blockIdx.x * blockDim.x + threadIdx.x;
14    const unsigned int bid = blockIdx.x;
15    const unsigned int tid = threadIdx.x;
16
17    s_int_fmo_ele_buf[tid] = 0;
```

```
18    for (int j = (tnum*70); j<(tnum*70)+70 && j < 268782 ; j++){
19      for (int i=0; i<3 ;i++){
20        dist_fmo[tid+i*128] =
21                  c_rnew[i] - fmo_cell_centre[ 268782*i + j];
22      }
23      dist_fmo4 = sqrt((dist_fmo[tid]*dist_fmo[tid]) +
24                       (dist_fmo[tid+128]*dist_fmo[tid+128]) +
25                       (dist_fmo[tid+2*128]*dist_fmo[tid+2*128]));
26      s_int_fmo_ele_buf[tid] += fmo_cell_charge[j]/dist_fmo4;
27    }__syncthreads();
28
29    if( tid < 64){
30          s_int_fmo_ele_buf[tid] += s_int_fmo_ele_buf[tid+64] ;}
31          __syncthreads();
32    if( tid < 32){
33          s_int_fmo_ele_buf[tid] += s_int_fmo_ele_buf[tid+32] ;}
34    if( tid < 16){
35          s_int_fmo_ele_buf[tid] += s_int_fmo_ele_buf[tid+16] ;}
36    if( tid <  8){
37          s_int_fmo_ele_buf[tid] += s_int_fmo_ele_buf[tid+ 8] ;}
38    if( tid <  4){
39          s_int_fmo_ele_buf[tid] += s_int_fmo_ele_buf[tid+ 4] ;}
40    if( tid <  2){
41          s_int_fmo_ele_buf[tid] += s_int_fmo_ele_buf[tid+ 2] ;}
42    if( tid == 0){
43      s_int_fmo_ele_buf[0]  += s_int_fmo_ele_buf[1];
44      g_int_fmo_ele_buf[bid] = s_int_fmo_ele_buf[0];
45    }__syncthreads();
46    return;
47  }
```

# 参考文献


[1] 前園涼・他，§4.5「量子モンテカルロ法による電子状態計算の実用化」,「計算力学シミュレーションハンドブック・超ペタスケールコンピューティングの描像」，日本計算工学会編，丸善，2009.

[2] R. Maezono，H. Watanabe，S. Tanaka，M.D. Towler and R.J. Needs，“Fragmentation Method Combined with Quantum Monte Carlo Calculations”， J. Phys. Soc. Jpn. **76**, 064301:1-5 (2007).

[3] K. Kitaura，T. Sawai，T. Asada，T. Nakano and M. Uebayasi，“Pair Interaction Molecular Orbital Method: An Approximate Computational Method for Molecular Interactions”, Chem. Phys. Lett. **312**，319-324 (1999).

[4] K. Kitaura，E. Ikeo，T. Asada，T. Nakano and M. Uebayasi，“Fragment Molecular Orbital Method: An Approximate Computational Method for Large Molecules”， Chem. Phys. Lett. **313**，701-706 (1999).

[5] “ACM Workshop on General Purpose Computing on Graphics Processors”, http://www.cs.unc.edu/Events/Conferences/GP2/， Aug. 2004

[6] 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”Conference on High Performance Networking and Computing, Gordon Bell finalists (2009)， ISBN:978-1-60558-744-8

[7] ハーベイ・ゴールド，ジャン・トボチニク 著，石川正勝，宮島佐介 訳，計算物理学入門，ピアソン・エデュケーション，2000

[8] J. M. ティッセン 著，松田和典，道廣嘉隆・他 訳，計算物理学シュプリンガー・ジャパン，2003

[9] 朝永振一郎，“第 6 章　物質の波動論”，量子力学 II，第二版，みすず書房，1997, p.1-114.

[10] W. M. C. Foulkes, L. Mitas, R. J. Needs and G. Rajagopal. Rev. Mod. Phys. **73** (2001) 33.

[11] 永瀬 茂，平尾 公彦， 現代化学への入門 17 分子理論の展開
岩波出版，2002

[12] R. J. Needs，M. D. Towler， N. D. Drummond and P. Lopez Rios，J. Phys. Condensed Matter 22，023201 (2010).

[13] QMCPACK Wiki， http://cms.mcc.uiuc.edu/qmcpack/index.php/

[14] インテル®マイクロプロセッサー製品の輸出規制基準
http://www.intel.co.jp/jp/support/processors/sb/CS-023143.htm

[15] GP2， http://www.cs.unc.edu/Events/Conferences/GP2/

[16] NVIDIA Fermi， http://www.nvidia.com/fermi

[17] 青木 尊之，額田 彰， はじめての CUDA プログラミング， I/O BOOKS，2009

[18] PGI Accelerator Compilers， http://www.pgroup.com/resources/accel.htm

[19] CUDA プログラミングツール及び Fortran サンプルコード,
http://www.nvidia.co.jp/object/cuda_programming_tools_jp.html

[20] CUDA Occupancy Calculator,
http://www.nvidia.co.jp/object/cuda_programming_tools_jp.html

[21] Intel Hyper-Threading Technology,
http://www.intel.com/jp/technology/platform-technology/hyper-threading/index.htm

[22] TOP500,
http://www.top500.org/