修 士 論 文

スプライン基底関数系を用いた
固体系の量子モンテカルロシミュレーション
に対するGPGPUによる高速化の研究

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

上嶋 裕

2012年3月

修 士 論 文

スプライン基底関数系を用いた 固体系の量子モンテカルロシミュレーション に対するGPGPUによる高速化の研究

指導教員　前園 涼 准教授

審査委員主査　前園 涼 准教授
審査委員　松澤 照男 教授
審査委員　下田 達也 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

1010007 上嶋 裕

提出年月: 2012 年 2 月

概 要

現在， ナノ材料開発の基礎研究として， 物質特性をシミュレーションする電子状態計算の一つに量子モンテカルロ法（Quantum Monte Carlo;QMC）がある．量子モンテカルロ法は従来手法の密度汎関数法と比べ， 電子の振る舞いをより厳密に取り入れた計算ができる．そのため従来手法では正確に描くことができなかった，生体分子や磁性の問題を量子モンテカルロ法は扱えるという特徴を有する．統計手法故に，量子モンテカルロ計算コードの並列化性能は 99%以上（80,000 並列時）と非常に高く， 近年のスーパーコンピュータが超並列化されていることに伴って， その活躍が期待されている．量子モンテカルロ法は基礎方程式を解くのに恣意的な近似を用いず，方程式を実直に取り扱っているため，計算の信頼性が高い一方で，電子状態計算の主たる対象となっている固体や大規模分子の計算では， 膨大な計算時間を要することが課題となっている．

この解決策として， 並列計算を行なっている各コアで律速となっている箇所を何らかの方法で高速化できれば， 計算全体の高速化を図ることができる．これを実現する手法の一つがハイブリッド並列化である．ハイブリッド並列は，MPI（Message Passing Interface）並列が行われている各計算ノード上，または各 CPU コア上で，さらに並列計算を行う手法である．MPI 並列と共存して並列計算ができる手法に OpenMP がある．OpenMP は CPU 内のコアで処理されているプロセスを多数のスレッドに分け並列計算を行い，律速箇所を容易に並列化することができる．近年のスーパーコンピュータにおける科学計算は，MPI+OpenMP のハイブリッド並列計算が一般的になりつつある．しかし OpenMP は CPU 内のコアを用いて並列計算を行うため，顕著な高速化が見込めないという問題がよく知られている．

OpenMP を超えるハイブリッド並列手法の一つとして， グラフィックカード内の演算ユニット（Graphics Processing Unit;GPU）を利用した GPGPU （General-purpose computing on GPU）による高速化が注目されている．GPU は CPU と比べ多数の演算ユニットを有し， 浮動小数点演算能力が非常に高い．構造が単純で高性能化が容易なこともあり，近年，アクセラレータ手法として様々な分野で目覚ましい発展を遂げている．また，CPU よりも低い消費電力で高い演算性能を発揮できることから，多くのスーパーコンピュータや大規模クラスタに搭載されつつある．

本研究では，スプライン基底関数系を用いた固体系の量子モンテカルロ計算の律速箇所について GPGPU を用いた高速化を行い，計算ノード単体における計算速度面での性能向上を図ることを目的とした．実装の結果，TiO_2 固体の 1536 電子の量子モンテカルロシミュレーションにおいて，単精度で 30.67 倍の律速箇所の高速化を実現した．論文では GPU のアーキテクチャに合わせた律速箇所の換装手法や，最適化手法について述べる．実装後の結果から，単精度化による計算値への影響や，達成された演算性能について議論する．

目 次

第1章　序論

1.1　HPCに関する背景

1.1.1　スーパーコンピューティングの動向

スーパーコンピュータとは，演算処理速度がその時代の一般的なコンピュータより極めて高速なコンピュータのことである．1980年代はベクトル型のスーパーコンピュータが主流であり，各メーカーは独自のプロセッサを開発していた．しかし90年代に入ると，安価な汎用のスカラープロセッサを複数個使用し，並列処理を行うスカラー型のスーパーコンピュータが台頭した．汎用プロセッサ自体も2000年前半までは高クロック化により性能を伸ばしてきたが，消費電力の問題のため，高クロック化による性能向上よりもマルチコアの流れになった．また，スカラー型スーパーコンピュータの高性能化のために，多くの並列ネットワーク方式が開発され，今では数千もの計算ノードをネットワークで接続した超並列マシンが主流である．

現代のHPC（high performance computing）分野で用いられるスーパーコンピュータは，1ノードに8-12コアを有するCPUが複数個搭載され，そのノード間を特殊なネットワークで接続した方式が主流である．アーキテクチャとしては非常に複雑化しており，一般的な並列プログラムでは全く性能を活かせないという問題がある．そのためアーキテクチャを意識した計算アルゴリズムに持ち込むことや，プログラミング技法が非常に重要となってきている．本論文で取り扱う話題も，このような側面が多いにある．

スーパーコンピュータの大規模化が進み，今ではペタスケールコンピュータからエクサスケールの領域が議論されている．ところが，此処で問題となっているのが，システムの消費電力である．エクサスケールを達成するには，このままの消費電力の増加傾向からすると，原子力発電所2基分の電力が必要であるという試算もあり，現実的ではないという意見がある[1]．そこで，CPUよりも省電力で演算パフォーマンスの高いGPUをアクセラレータとして用いる手法が注目されている．

1.1.2　GPUの発展とGPGPU

GPUは3Dグラフィックスを描画するための専用ハードウェアである．そのためGPUは3Dグラフィックスの描画処理において頻発する，頂点座標の計算，三角形の位置計算

などのベクトル計算に特化したハードウェアとして設計されている．初期の GPU は固定された描画処理命令（シェーダ）しか扱うことができなかったが，近年ではユーザーがプログラム可能なプログラマブルシェーダ（ストリーミングプロセッサ）が導入された．このプログラマブルなストリーミングプロセッサが GPU 上に大量に搭載されるようになると，これを用いて，並列コンピューティングのように数値計算を行う動きが出てくる．これが GPU での汎用計算，GPGPU の原点である [2]．GPU メーカーも数値計算への実用性に目をつけ，2006 年に NVIDIA 社は GPU 向けの総合開発環境，CUDA を公開した．

初期の GPGPU は，複雑なプログラミングと多くの制約，倍精度が扱えないなど問題が多く，科学者から敬遠されることも少なくなかった．しかし GPU は僅か数年でそれらの問題を克服し，また構造の単純さから内部の演算コア数を劇的に増やし，現在では 500 以上ものコアを有する GPU を安価に利用できるようになった．それに伴い，GPU 単体の理論性能値は 1 TFlops 以上に達している．GPU は CPU に比べ，膨大な並列計算を安価に扱えることから，近年では多くの分野で GPGPU が用いられるようになってきてる．膨大な演算を必要とするシミュレーション分野で事例を挙げると，気象学・流体工学・金融工学・天文学・分子動力学など多岐に亘って GPGPU が用いられている．いずれも数十〜数百倍の計算高速化が達成されており [3]，活用の場は益々広まりつつある．GPGPU は，既存計算ノードの PCI-Express スロットに GPU を挿入するだけで扱うこと可能で，ノード単体の演算理論性能を向上できるという手軽さから，スーパーコンピュータやクラスタに搭載されるようになった．また，上述の消費電力の観点から intel 社も，このようなアクセラレータ技術に乗り出している．

一方で，GPGPU に対応した汎用アプリケーションが少ないことも問題視されている．これは GPGPU 特有のプログラミングと，性能が GPU アーキテクチャに依存しやい要因が大きい．前述した OpenMP の場合，既存コードの並列化可能な部分に対して，明示的に数行のコードを記述するだけで，容易に並列化を行える．また多くの既存コンパイラが OpenMP に対応しており，ユーザビリティに特化している．しかし，GPGPU の場合，別途，GPU 演算用のコードを新たに記述する必要があり，GPU アーキテクチャに合わせた最適化を行わないと十分に性能を発揮できない．そのため GPU を用いることで，却って演算が遅くなってしまうという例も多くある．また CPU-GPU 間のメモリ転送時間がかかるため，この時間ロスを補えるだけの十分な演算量と並列度を確保しなければならない．このような煩わしさから，従来の汎用アプリケーションコードを GPGPU に対応させるためには，大きな労力と技術が必要となる．しかし，近年のスーパーコンピュータへの GPU 搭載の流れから，ハードウェア性能（理論性能）とソフトウェア性能（実性能）の差が開きすぎている問題がある．したがって，汎用アプリケーションを GPU 計算に特化させることが急務となっている．

1.2　電子状態計算に関する背景

1.2.1　電子状態計算

ある物質の物理的・化学的性質の理解には，その中に存在する原子核と電子の挙動を知ることが不可欠である．通常，原子核の運動は電子に比べると非常に遅いため，原子核を固定し，電子の運動のみを扱うことで，物質の挙動として近似できる（ボルン・オッペンハイマー近似）．電子の振る舞いはシュレディンガー方程式で記述され，この基礎方程式を解き，物質の挙動を支配している，系のエネルギーと波動関数を計算する手法が電子状態計算である．

シュレディンガー方程式を解き，物質の性質を理解する試みは量子力学が誕生した 20 世紀初頭から行われてきたが，多くの電子から構成される物質についてシュレディンガー方程式を解くためには膨大な数値計算が必要なため，多くの近似を含む必要があるなどの理由から，高い精度で値を得ることは困難であった．しかし，近年のスーパーコンピュータやワークステーションの計算能力の向上と，多くの高精度な近似手法の研究により，その基礎方程式を高い精度でコンピュータで解くことが可能になった．このコンピュータの進歩の恩恵を受け，新しい特性を有する物質を生成する際に，従来は経験則を当てにして試行錯誤で創ることしかできなかったものを，コンピュータ上でシミュレーションすることで，クリーンで高効率な材料開発が行えるようになった．またこのシミュレーションの枠は基礎研究だけでなく，化学，材料，医療を筆頭とした産業界までに及び，電子状態計算はここ数年で一般的になりつつある．

大規模計算機能力を，電子状態計算にどう活用するか？といった場合，大きく，3 つの方向性がある．より大規模系に，より高精度に，より長時間のシミュレーションに，というものである．より長時間にというのは，本研究での量子モンテカルロシミュレーションの対象ではないが，ダイナミクスを含む電子状態計算で問題とされる事項である．ダイナミクスを含まないシミュレーションでは，精度を一定に保つなら，より大きなサイズを扱える．そのため最近ではデバイスのサイズにまでシミュレーションサイドを大きくするような試みがある [4]．一方，サイズを一定に保つなら，より高精度のシミュレーションが可能になる．高精度計算には，現行の電子状態計算の信頼性では描くことができない問題を扱うことができるという需要がある．粗い近似を用いた従来法の計算の場合，例えば生体系などの数百℃レベルの低エネルギーを相手にする問題では，十分な分解能を得ることができない．従来法は共有結合のエネルギースケールである数万℃オーダー程度の分解能しか有しておらず，化学反応や，更には生化学反応などを扱う場合，そこでの結合形態を見分ける精度が必要である．

1.2.2 量子モンテカルロ法

本グループが扱う量子モンテカルロ法（Quantum Monte Carlo;QMC）は電子状態計算手法の一つである．従来の電子状態計算手法に比べ，非常に高い精度でシミュレーションが可能で，従来法では困難であった，数百℃オーダーの分解能が必要とされる生体分子や磁性の問題を扱うことができる．したがって，現代のバイオテクノロジーやスピントロニクスといった分野での応用が期待されている．一方で，量子モンテカルロ法はシュレディンガー方程式を従来手法より実直に取り扱うため，計算コストが大きい問題がある．

しかしながら，量子モンテカルロ法はエネルギー値の算出に統計的手法を用いるため，スーパーコンピュータで並列計算する際に，計算時間に比べ，通信時間が殆ど発生しない．また発生するノード間通信は，全対全（All to all）通信ではなく，一対全（Broadcast/Gather）通信である．そのため，並列コンピューティングにおいて最大のボトルネックとなるノード間のネットワーク帯域に影響されにくく，並列数を増やせば増やすほど，リニアに演算速度を伸ばせる利点がある．ゆえに，膨大なコア数を有する現代のスーパーコンピュータにおいて，最も性能を発揮でき，システムサイズが大きい大規模分子や固体周期系のような物質に対しても，電子状態計算を高精度に行える手法として注目されている．

量子モンテカルロ法は計算コストが高いため，数年前までは，比較的小規模の原子・分子に適用されるのみであった．ただ，系のサイズを N で特徴付けると（例えば粒子数），計算量は N^3 にスケールするという特質がある．精密な電子状態計算法として分子科学的な手法があるが，これらの計算量は N^7 にスケールするため [5]，この手法を固体のような大規模な系に用いることは困難である．しかし，量子モンテカルロ法は非常に信頼性が高いにもかかわらず，超並列計算機を利用できる環境であれば，固体系が扱えるという特徴を有する．固体系には，磁性や表面の問題など，非常に高精度が要求される問題が大いに存在する．量子モンテカルロ計算はこのような新境地を開くことができる手法であり，計算速度を向上することでのインパクトは大きい．

1.3 研究対象と目的

本研究では電子状態計算手法の一つである量子モンテカルロ法の汎用計算コードの高速化を目的とする．取り分け，取り扱う電子数が多く計算コストが大きい，固体周期系に対する計算の高速化に焦点を絞る．

背景で述べたように，量子モンテカルロ計算の並列性能は非常に高く，汎用計算コードの並列化効率は 80,000 並列時で 99%の効率に達する．したがって，更なる計算速度の向上を目指すには，計算ノード単体での性能向上を施すことの方がたやすい．我々の先行研究では，電子状態計算の一種であるフラグメント分子軌道法（FMO）を量子モンテカルロ法に拡張した「FMO-QMC 法」に対し，GPGPU を適用した前例がある．この研究で

は，FMO-QMC 計算の律速箇所を GPGPU に部分換装することで，23.6 倍の高速化を達成している [6]．しかし FMO-QMC 法は分子系の計算にしか適用することができず，我々が課題の一つとしている，固体系の問題では扱えない．

そこで，固体系の電子状態計算が可能な，量子モンテカルロ法の汎用計算コードへの GPGPU の適用を行った．先行研究と同様，各ノードで実行されている計算の律速箇所に対して，GPGPU を用いた高速化を行い，計算ノード単体での性能向上を目的とする．本論文では，量子モンテカルロ計算コードの律速部分の高速化を GPGPU によって実現するための方法について議論し，実装を行った結果と考察について述べる．

第2章　計算原理

本章では電子状態計算の基礎となるシュレディンガー方程式の取り扱いと，方程式を解くために用いられる手法について概要を述べる．

2.1　問題の構造

2.1.1　シュレディンガー方程式

電子状態計算とは，以下の多変数偏微分固有値方程式を基礎方程式として，これを解く計算全般を指す：

$$\left\{ -\frac{1}{2}\sum_{j=1}^{N}\nabla_j^2 + V\left(\vec{r}_1,\vec{r}_2,\cdots,\vec{r}_N\right) \right\} \Psi\left(\vec{r}_1,\vec{r}_2,\cdots,\vec{r}_N\right) = E\Psi\left(\vec{r}_1,\vec{r}_2,\cdots,\vec{r}_N\right) \tag{2.1}$$

ただし N は系に含まれる電子数で, $(\vec{r}_1,\vec{r}_2,\cdots,\vec{r}_N)$ は N 個の電子の位置を表す．∇ を含む項は系の運動エネルギーを，$V\left(\vec{r}_1,\vec{r}_2,\cdots,\vec{r}_N\right)$ の項はポテンシャルエネルギーを表す．Ψ は未知の固有関数で，系の電子配位セットを引数とした多体波動関数である．(2.1) 式の $\{\cdots\}$ の演算子部分は $\hat{H}$ (Hamiltonian) と呼ばれ，以後，演算子 $\hat{H}$ を用いて表す．[1](2.1) 式は時間の変化に関わらず成立するので，定常状態を表しており時間を含まないシュレディンガー方程式と呼ばれ，この固有値問題を解いて得られる固有値 E が系の全エネルギーを与える．

ボルンオッペンハイマー近似では，原子核位置はパラメーターとして与えられ，(2.1) 式の演算子 $\hat{H}$ は固定される．問題の構造は，したがって，所与の演算子 $\hat{H}$ に対して，固有値 E と固有関数 Ψ を解くという固有値問題となる．問題に与えられる入力は原子核位置，すなわち，分子構造や結晶構造である．このような形式は，原子核位置以外に経験的なパラメーターや実験値を要求しないので，第一原理計算（*ab initio* calculation）とも呼ばれる．多変数関数の偏微分固有値問題を厳密かつ解析的に取り扱うことは不可能であり，多くの近似解法や数値的手法が提案・開発されている．

[1]基礎方程式中の物理定数を 1 とおいた原子単位系 (*a.u.*) を用いている [7].

2.1.2 変分法による定式化

多体のシュレディンガー方程式の固有値 E を求める一つの方策に，以下に述べる変分法がある．(2.1) 式を変形して求める．電子位置のセット $\{\vec{r}_1, \vec{r}_2, \cdots, \vec{r}_N\}$ を $\vec{R}$ と略記し，次のように，両辺に左から $\Psi\left(\vec{R}\right)$ の複素共役を掛けると

$$\Psi^*\left(\vec{R}\right)\hat{H}\Psi\left(\vec{R}\right) = \Psi^*\left(\vec{R}\right)E\Psi\left(\vec{R}\right) \tag{2.2}$$

となり，E について変形すると

$$\begin{aligned} E &= \frac{\int d\vec{R}\ \Psi^*\left(\vec{R}\right)\hat{H}\Psi\left(\vec{R}\right)}{\int d\vec{R}\ \Psi^*\left(\vec{R}\right)\Psi\left(\vec{R}\right)} \\ &= \frac{\int d\vec{R}\ \Psi^*\left(\vec{R}\right)\hat{H}\Psi\left(\vec{R}\right)}{\int d\vec{R}\ \left|\Psi\left(\vec{R}\right)\right|^2} \end{aligned} \tag{2.3}$$

を得る．元来未知の多体波動関数 Ψ に対し，試行関数を与えれば，上式の多重積分を評価することでエネルギー値を評価することができる．波動関数の自乗絶対値 $\left|\Psi\left(\vec{R}\right)\right|^2$ は粒子の存在確率密度分布を示すので，N 粒子系の場合，次のように規格化される：

$$\int d\vec{R}\ \left|\Psi\left(\vec{R}\right)\right|^2 = N \tag{2.4}$$

(2.3) 式にて評価される E は真の固有値に対して変分性を有する．これは以下のように示される：真の波動関数は未知なので，任意の試行関数 $\Psi_{\rm trial}$ を仮定し，ハミルトニアン $\hat{H}$ の規格直交化された固有関数 Φ_i の組 $(i = 0, 1, 2, \cdots)$ で

$$\Psi = \sum_i a_i \cdot \Phi_i \tag{2.5}$$

と展開する．(2.1) 式より

$$\hat{H}\Phi_i = E_i\Phi_i \tag{2.6}$$

なので，(2.3) 式にこれらを代入すると

$$\begin{aligned} \frac{\int d\vec{R}\ \Psi^*_{\rm trial}\hat{H}\Psi_{\rm trial}}{\int d\vec{R}\ \Psi^*_{\rm trial}\Psi_{\rm trial}} &= \frac{\int d\vec{R}\left(\sum_j a_j^*\Phi_j^*\right)\left(\sum_i a_i E_i\Phi_i\right)}{\int d\vec{R}\left(\sum_j a_j^*\Phi_j^*\right)\left(\sum_i a_i\Phi_i\right)} \\ &= \frac{\sum_{ij} a_j^* a_i E_i \int d\vec{R}\ \Phi_j^*\Phi_i}{\sum_{ij} a_j^* a_i \int d\vec{R}\ \Phi_j^*\Phi_i} \end{aligned} \tag{2.7}$$

となる．$\Psi_{\rm trial}$ は規格直交系で展開しているので $\int d\vec{R}\,\Phi_j^*\Phi_i = \delta_{ij}$ であり，上式は

$$\frac{\int d\vec{R}\,\Psi_{\rm trial}^*\hat{H}\Psi_{\rm trial}}{\int d\vec{R}\,\Psi_{\rm trial}^*\Psi_{\rm trial}} = \frac{\sum_i E_i\,|a_i|^2}{\sum_i |a_i|^2} \tag{2.8}$$

となる．ここで，$E_0 < E_1 < E_2 \cdots$ なので

$$\sum_i E_i\,|a_i|^2 \geq \sum_i E_0\,|a_i|^2 \tag{2.9}$$

が成り立つ．したがって

$$\begin{aligned} E_{\rm trial} &= \frac{\int d\vec{R}\,\Psi_{\rm trial}^*\hat{H}\Psi_{\rm trial}}{\int d\vec{R}\,\Psi_{\rm trial}^*\Psi_{\rm trial}} \geq E_0 \cdot \frac{\sum_i |a_i|^2}{\sum_i |a_i|^2} \\ &\geq E_0 \end{aligned} \tag{2.10}$$

という関係が成り立つ．この時，試行関数 $\Psi_{\rm trial}$ が厳密な波動関数 Ψ と一致するとき，試行関数 $\Psi_{\rm trial}$ を用いて評価した固有値 $E_{\rm trial}$ は，E の厳密解と一致する．真の試行関数を得ることは不可能であるが，$E_{\rm trial}$ を最小化するように，試行関数を調整してやれば，厳密解に近い基底状態エネルギーの近似値を得ることができる．

実際は，$E_{\rm trial}$ を最小化するために，いくつかの変分パラメータ $\{\alpha_n\}$ を含む試行関数 $\Psi_{\rm trial}\left(\vec{r}_1, \vec{r}_2, \cdots, \vec{r}_N; \{\alpha_n\}\right)$ を準備し，$E_{\rm trial}$ を最小化するようにパラメータを調整する手法が取られる．

2.2 近似手法

多体問題の難しさは，多体自由度を「混ぜる」相互作用の存在にある．相互作用がなければ変数分離ができ，一体問題に持ち込めるので扱いが楽になる．したがって電子状態計算の現行法では，相互作用の部分に近似を施して扱っている．近年のナノテクノロジーの高度化により，電子状態計算の実用的適用も地平を拡げてきた．それにともなって，現行法での電子間相互作用の近似的取り扱いに起因して理論計算予見が十分な信頼性を達成しない事例が多く知られてるようになってきた．そのため現行法を超えた，本論文で扱うところの量子モンテカルロ法に需要が高まってきている．本論文で扱う量子モンテカルロ法は恣意的な近似を用いず，相互作用を真面目に取り扱うため，現行法では書けない電子相関を取り入れた電子状態計算が可能である．この節では量子モンテカルロ法とは異なる現行法について簡単に触れておく．

2.2.1 1電子軌道描像の導入

(2.1) 式の基礎方程式のポテンシャルエネルギー V の項は

$$V\left(\vec{r}_1,\vec{r}_2,\cdots,\vec{r}_N\right)=\sum_{i=1}^{M}\sum_{j=1}^{N}\frac{-Z_i}{\left|\vec{R}_i-\vec{r}_j\right|}+\sum_{i=1}^{N}\sum_{j>1}^{N}\frac{1}{\left|\vec{r}_i-\vec{r}_j\right|} \tag{2.11}$$

と分けられる．上式の第一項は電子と原子核，第二項は電子と電子間に生じるクーロン相互作用による静電ポテンシャルである．とりわけ第二項の電子間相互作用が多電子の自由度を混ぜ，問題を難しくしている．電子間相互作用が無ければ変数分離ができるので，多体問題を一体問題に持ち込む様々な近似法が開発されている．その一つのハートリー・フォック近似について簡単に述べる．

基礎方程式を解くにあたり多体波動関数 $\Psi\left(\vec{r}_1,\vec{r}_2,\cdots,\vec{r}_N\right)$ の形が未知であるため，ハートリー近似では軌道関数と呼ばれる1電子波動関数 $\{\phi_j\left(\vec{r}_j\right)\}$ を用いて，Ψ を

$$\Psi\left(\vec{r}_1,\vec{r}_2,\cdots,\vec{r}_N\right)=\phi_1(\vec{r}_1)\phi_2(\vec{r}_2)\cdots\phi_N(\vec{r}_N) \tag{2.12}$$

と表すことができると仮定し，これを試行関数 Ψ_{trial} とする．これは元来，変数分離できない Ψ を ϕ で変数分離できると仮定するという重大な近似である．Ψ は (2.4) 式の波動関数の規格化条件を満たす必要があるので，ϕ に対して

$$\int\left|\phi_j\left(\vec{r}_j\right)\right|^2 d\vec{r}=1 \tag{2.13}$$

という規格化条件を課せば Ψ に関する規格化条件を満たされる．§2.1.2 で述べた変分原理の問題は，(2.13) 式の拘束条件のもと

$$\int\Psi_{\text{trial}}^{*}\hat{H}\Psi_{\text{trial}}\,d\vec{r}_1 d\vec{r}_2\cdots d\vec{r}_N$$

が最小になるよう $\{\phi_j\left(\vec{r}_j\right)\}$ を調整するという問題になる．ラグランジュの未定乗数 ε_j を導入すると

$$F=\int\Psi^{*}\hat{H}\Psi d\vec{r}_1 d\vec{r}_2\cdots d\vec{r}_N-\sum_{j=1}^{N}\varepsilon_j\left[\int\left|\phi_j\left(\vec{r}_j\right)\right|^2 d\vec{r}-1\right] \tag{2.14}$$

が極値をとりうる $\{\phi_j\left(\vec{r}_j\right)\}$ を決定すればよいことになる．この変分問題を解いて得られるオイラー・ラグランジェの方程式は

$$\left\{\hat{h}_j\left(\vec{r}_j\right)+\frac{1}{4\pi\varepsilon_0}\sum_{i(\neq j)}^{N}\int\frac{\left|\phi_i\left(\vec{r}_i\right)\right|^2}{\left|\vec{r}_i-\vec{r}_j\right|}d^3\vec{r}\right\}\phi_j\left(\vec{r}_j\right)=\varepsilon_j\phi_j\left(\vec{r}_j\right)\qquad(j=1,2,\cdots,N) \tag{2.15}$$

となり，これはハートリー方程式と呼ばれる．(2.15) 式の {} 内の第二項は j 番目の電子と残りの電子がつくる平均的なクーロン場（平均場）との相互作用を示す．したがって，ハートリー近似は電子間相互作用が平均においてのみ考慮されており，このような近似を平均場近似と呼ぶ．

また，系の多体波動関数は電子の特性（反対称性）

$$\Psi\left(\cdots,\vec{r}_j,\cdots,\vec{r}_i,\cdots\right)=(-)\,\Psi\left(\cdots,\vec{r}_i,\cdots,\vec{r}_j,\cdots\right) \tag{2.16}$$

を満たさなければならない．この条件を満たす，最もシンプルな方法が (2.17) 式ように一体軌道関数 ϕ の積和に変数分離した形を仮定することである．

$$\Psi\left(\vec{r}_1,\cdots,\vec{r}_N\right)=\frac{1}{\sqrt{N!}}\begin{vmatrix}\phi_1\left(\vec{r}_1\right) & \cdots & \phi_1\left(\vec{r}_N\right)\\ \vdots & \ddots & \vdots\\ \phi_N\left(\vec{r}_1\right) & \cdots & \phi_N\left(\vec{r}_N\right)\end{vmatrix} \tag{2.17}$$

この行列式をスレーター行列式と呼び，これを試行関数として用いてエネルギー値を得る手法がハートリー・フォック（Hartree-Fock;HF）法である．

ハートリー・フォック法は粗い近似を含むが，厳密な基底エネルギーの大部分を表現できることが知られている [13]．しかし，残りの僅かなエネルギー誤差のため，ハートリー・フォック法では数 kcal/mol 程度エネルギー（=数百℃）の化学反応を記述できない．ハートリー・フォック法で表現できないエネルギー（$E_{\text{correlation}}$）

$$E_{\text{correlation}}=E_{\text{exact}}-E_{\text{Hartree Fock}}$$

を電子相関と呼び，この電子相関を取り入れるために，配置間相互作用法（Configuration Interaction;CI）や，密度汎関数理論（Density Functional Theory;DFT）などを用いた計算手法が開発されており，平均場近似を超える試みが盛んに行われている．

2.2.2 密度汎関数法

現在，最も広く使われている電子状態計算の理論が DFT 法である．DFT は「電子の電荷密度 $n\left(r\right)$ が正しく与えられれば，厳密な系の基底エネルギーが決まる」という Hohenberg-Kohn の定理を基礎におく．この定理をもとに，多体問題の厳密な基底エネルギーが見かけ上の 1 電子シュレディンガー方程式（Kohn-Sham 方程式）から得られる基底エネルギーに一致することが示されている．Kohn-Sham 方程式には交換・相関エネルギーの項があり，電子密度の汎関数として与えられる．問題は電子密度と基底エネルギーを結びつける汎関数が未知であり，これを如何に表現するかというもので，様々な汎関数が研究されている．

DFT で多くの物質において非常に高い精度の計算を可能としている方法が局所密度近似（Local Density Approximation;LDA）である．局所密度近似は本来，Kohn-Sham 方程

式における電子密度の汎関数を扱う部分を，空間の各点での局所密度だけで決まる交換・相関エネルギー密度で近似するという手法をとる．局所密度近似は多くの成功を収めている一方で，不都合も明らかになっている．例えば，半導体特性の重要指標であるバンドギャップを過小評価し，場合によっては絶縁体を金属と予測する例が多々ある．また，動的なゆらぎを伴う分散力やファンデルワールス力を扱えない問題がよく知られている[11][12].

2.3 量子モンテカルロ法

2.3.1 モンテカルロ積分による評価

ハートリー・フォック法や密度汎関数法では，元来多体座標 $(\vec{r}_1, \vec{r}_2, \cdots, \vec{r}_N)$ 上で記述される問題が，一体軌道関数に還元して扱われた．その過程で近似が導入され，特に電子間相互作用についての精度や信頼性が犠牲になった．これに対し量子モンテカルロ法は多体波動関数を多体座標の上で直接扱う．

量子モンテカルロ法は (2.3) 式の多重積分をモンテカルロ積分で数値評価する．(2.3) 式は

$$\begin{aligned} E &= \frac{\int d\vec{R}\, \Psi^* \left(\vec{R}\right) \hat{H} \Psi \left(\vec{R}\right)}{\int d\vec{R} \left|\Psi \left(\vec{R}\right)\right|^2} \\ &= \int d\vec{R}\, \frac{\left|\Psi \left(\vec{R}\right)\right|^2}{\int d\vec{R} \left|\Psi \left(\vec{R}\right)\right|^2} \Psi^{-1} \left(\vec{R}\right) \hat{H} \Psi \left(\vec{R}\right) \end{aligned} \tag{2.18}$$

と変形できる．ここで

$$P\left(\vec{R}\right) = \frac{\left|\Psi \left(\vec{R}\right)\right|^2}{\int d\vec{R} \left|\Psi \left(\vec{R}\right)\right|^2} \tag{2.19}$$

は $P\left(\vec{R}\right)$ は確率密度関数の性質，

$$\int P\left(\vec{R}\right) d\vec{R} = 1, \qquad 0 \leq P\left(\vec{R}\right) \leq 1$$

を満たす．したがって，(2.18) 式は確率密度関数 $P\left(\vec{R}\right)$ による $\Psi^{-1} H \Psi$ の期待値とみなすことが出来る．$P\left(\vec{R}\right)$ の分布に従うランダムな点 $\{R_j\}_j^M$ を生成（重点的サンプリング）

することができれば，(2.18) 式は

$$\begin{aligned} E &= \int d\vec{R}\, P\left(\vec{R}\right) \Psi^{-1}\left(\vec{R}\right) \hat{H}\Psi\left(\vec{R}\right) \\ &= \left\langle \Psi^{-1}\left(\vec{R}\right) \hat{H}\Psi\left(\vec{R}\right) \right\rangle_{P(\vec{R})} \\ &\simeq \frac{1}{M}\sum_{j}^{M} \Psi^{-1}\left(\vec{R}_j\right) \hat{H}\Psi\left(\vec{R}_j\right) \end{aligned} \tag{2.20}$$

となり，固有値 E はサンプリング値の統計平均として評価できる．大数の法則より $M \to \infty$ で (2.20) 式の E は (2.18) 式のそれと一致し, 中心極限定理より統計誤差は $\sqrt{M}$ に反比例して縮小する [14].

§2.1.2 に述べた多体変分法では，変分パラメーターを含んだ試行関数 $\Psi_{\rm trial}\left(\vec{R};\alpha\right)$ を与えて，件の (2.10) 式を最小化するように試行関数を調整していくという手順であった．$\Psi_{\rm trial}$ を与えると，確率密度関数 P が確定し，上記の方策で E を統計推定量として評価できる．この統計推定量を最小化するように再度 $\Psi_{\rm trial}$ を調整して厳密解に迫っていく手法を変分モンテカルロ法（Variational Monte Carlo;VMC）という．

2.3.2 メトロポリス法

上述の変分モンテカルロ法では，所与の確率密度関数 P に対して，その分布で発生するランダムな点 $\{R_j\}_j^M$ を生成することが鍵となった．これを実現する方法として，点列 $\left\{\vec{R}^{(t)}\right\}$ をマルコフ連鎖で確率的にドライブして所望の $P\left(\vec{R}\right)$ の分布に従うよう遷移させる手法がある．この際の遷移確率 T は次の詳細釣り合いの条件

$$P\left(\vec{R}\right) T(\vec{R} \to \vec{R}') = P\left(\vec{R}'\right) T(\vec{R}' \to \vec{R}) \tag{2.21}$$

を満たすように設定すれば良いことが知られている．この条件を満たす T の設定法の一つにメトロポリス法があり [15]，遷移確率は次の式で決定される．

$$T(\vec{R} \to \vec{R}') = \min\left(1, \frac{P\left(\vec{R}'\right)}{P\left(\vec{R}\right)}\right) \tag{2.22}$$

これを用いて，量子モンテカルロ計算では次のアルゴリズムで電子位置のセット $\vec{R}$ の更新が行われる：

step 0 初期状態 $\vec{R}^{(0)}$ を生成

step 1 ある点 $\vec{R}$ からランダムに変化させた新しい点 $\vec{R}'$ を作る

step 2 確率密度関数の比 $\xi = \frac{P(\vec{R'})}{P(\vec{R})}$ を計算

step 3 一様乱数 $\eta \in [0,1]$ を生成し，$\eta \leq \xi$ なら $\vec{R'}$ を採用，$\eta > \xi$ なら棄却

step 4 step1 に戻り，繰り返す

2.3.3 量子モンテカルロ法電子状態計算

§2.3.1 の末尾で述べたように，変分モンテカルロ法では変分パラメーター群 α を含んだ多体の試行関数を仮定し，変分原理に基づいて α の最適化を行う．試行関数の初期値としては，ハートリー・フォック法や密度汎関数法などの電子状態計算で得られた一体軌道関数を用いて，(2.17) 式のスレーター行列式を構成する．試行関数はこれにジャストロー因子を付与したスレーター・ジャストロー型試行関数

$$\Psi_{\text{trial}}\left(\vec{R};\alpha\right) = \exp\left[J\left(\vec{R};\alpha\right)\right] \cdot \begin{vmatrix} \phi_1(\vec{r}_1) & \cdots & \phi_1(\vec{r}_N) \\ \vdots & \ddots & \vdots \\ \phi_N(\vec{r}_1) & \cdots & \phi_N(\vec{r}_N) \end{vmatrix} \tag{2.23}$$

を用いる．ジャストロー関数がない場合，この変分モンテカルロ法はハートリー・フォック法と等価になる．ジャストロー関数は電子相関を取り入れるための新たな自由度であり，多体波動関数の振幅変調を表現する．

エネルギー固有値は上述のモンテカルロ積分法に基づいて

$$E_{VMC} = \frac{1}{M}\sum_{j=1}^{M}\left(\Psi_{\text{trial}}^{-1}\left(\vec{R}_j\right)\hat{H}\Psi_{\text{trial}}\left(\vec{R}_j\right)\right) \tag{2.24}$$

と統計推定値で評価する（α の依存性を省略した）．統計値なので，E_{VMC} = 平均値 $\pm$ エネルギー統計誤差 となり，サンプル数 M を大きくすることで，誤差を絞ることができる．この誤差は化学分野で必要とされている分解能 $\Delta\varepsilon$ ∼0.001hartree （$1hartree = 4.3598 \times 10^{-18}[J]$)[16] 程度以下にに収まるまで絞られる．

変分モンテカルロ法における計算結果の信頼性は，結局は仮定した試行関数 Ψ_{trial} の質で決まってしまう．本論文で直接は取り扱わないが，これを超える手法として拡散モンテカルロ法（Diffusion Monte Carlo;DMC）という手法が用いられている．そこでは，初期推定として設定した試行波動関数に基づく確率分布関数が巧みな射影演算によって，より厳密解に近いものに近づいていく．これにより，さらに精度よくエネルギー値を計算することが出来る [17]．本研究では変分モンテカルロ法における計算高速化をテーマとしており，変分モンテカルロ法の計算手法について，更に概略を述べる．

2.3.4 MPIによる並列化性能

変分モンテカルロ計算では，求められるエネルギー分解能を得るためにステップ数 M を数千万以上に設定する必要があるが，MPIを用いた実用的な分散処理が行われる．MPIを用いた現行実装では，(2.24) 式における個別のサンプル値

$$\Psi_{\mathrm{trial}}^{-1}\left(\vec{R}_j\right)\hat{H}\Psi_{\mathrm{trial}}\left(\vec{R}_j\right) \tag{2.25}$$

が各演算コアで分散処理され，これらをマスターノード上に集約し，最終的に統計平均が評価される．量子モンテカルロ計算のパッケージの並列化効率は，ストロングスケーリングでは1,000並列で99%以上，さらに20,000並列でも90%の効率に達する．またウィークスケーリングならば80,000並列時に99%の効率である [18]．統計計算であるため各並列コアの間で依存性が全く発生せず，演算時間に比べ通信時間が十分小さいことから，高い並列化効率が達成されている．

2.3.5 変分モンテカルロ計算の律速部分

MPIによる高速化は現行実装において十分高い効率で実現している．更なる高速化の余地は，各演算コア単体上で評価される (2.25) 式のサンプル値評価である．その律速部分は §2.3.2 で述べたメトロポリス・アルゴリズムの箇所と特定される．そこでは，

step 1 ある i 番目の電子の位置を更新する $(\vec{r}_i \to \vec{r}_i')$

step 2 更新前と更新後の試行関数の比

$$\xi = \frac{\Psi_{\mathrm{trial}}\left(\vec{r}_1, \vec{r}_2, \cdots, \vec{r}_i', \cdots, \vec{r}_N\right)}{\Psi_{\mathrm{trial}}\left(\vec{r}_1, \vec{r}_2, \cdots, \vec{r}_i, \cdots, \vec{r}_N\right)} \tag{2.26}$$

を求める

step 3 試行関数比 ξ が一様乱数 $\eta \in [0,1]$ 以上ならば位置の更新を採択し，そうでない場合は棄却する

という一連の評価が行われている．これを全ての電子 $i = 1 \sim N$ に関して繰り返し，一つの更新配位 $\vec{R}$ が確定すると，(2.25) 式のサンプル値が評価され統計平均に足し上げられる．電子位置の更新（$\vec{r} \to \vec{r}'$）の度に (2.23) 式における軌道関数 $\{\phi_j(\vec{r}_i')\}_j^N$ の再評価が行なう．よって，この再評価のために「blip3d」と呼ばれる計算ルーチンの呼び出し回数が非常に多く，ここが律速となっている．blip3dは変分モンテカルロ計算全体の30%程度を占めていることが分かっており，この部分の高速化が最大の課題となっている．

2.3.6 律速の支配要因

電子状態計算では，§2.2.1 で述べたスレーター行列式を構成する軌道関数 $\{\phi_j(\vec{r}_i')\}$ は更に，

$$\phi_j(\vec{r}_i) = \sum_{l=1}^{M} C_l^{(j)} \cdot \chi_l(\vec{r}_i) \tag{2.27}$$

と，解析的性質のよい基底関数系 $\{\chi_l(\vec{r}_i)\}$ で展開されて取り扱われる．基底関数系には，計算対象となる系を表現しやすい基底を用いるのが一般的で，固体周期系（単位セルが空間的に繰り返される系）の場合には平面波基底，分子系の場合はガウシアン基底が最もポピュラーに用いられる．量子モンテカルロ法の場合，計算量は基底関数系の選定に大きく影響を受ける：量子モンテカルロ計算の律速部分 (2.26) 式の評価は，ある電子位置の更新 $\vec{r}_i \to \vec{r}_i'$ に対して Ψ_{trial} を評価しなおすという手順に帰着する．それは更には Ψ_{trial} を構成する軌道関数 $\left\{\phi_j(\vec{r}_i')\right\}$ の評価に帰着する．したがって最終的には，基底関数系 $\left\{\chi_l\left(\vec{r}_i'\right)\right\}$ の更新評価に帰着する．したがって，更新を受ける基底関数の数が多ければ多いほど，計算量は嵩み計算が遅くなる．平面波基底の場合，ある電子の位置更新という局所的なイベントでも系全体に亘った平面波の重ね合わせとして記述される．したがって，如何なる局所的変化に対しても，全ての基底関数（平面波）が更新されなければならない．この数は通常，数万から数十万で膨大な計算量となる．一方，ガウシアン基底のように局所的な基底関数系の場合，電子位置の更新による局所的な変動は，その周りの高々数百程度の基底系の更新で記述しきれるため，更新すべき計算量が抑えられる．このように基底系の局所性が佳いと前述したの律速部分はより高速に評価できる．そこで，局所性のもっともよいスプライン関数系を基底関数にして軌道関数を表現するという方策に至る．これを実装したのが，B-スプライン基底（bases-spline;blip）[19] で，固体周期系の場合，平面波基底ではなく blip 基底を用いると，軌道関数の再評価は 200 倍程度高速になる [20].

この理由から，量子モンテカルロ計算での固体周期系の計算は blip 基底を用いるのが一般的であり，本研究では blip 基底を扱った変分モンテカルロ計算における軌道関数 $\{\phi_j(\vec{r}_i')\}$ の再評価部分の高速化に着目した．

2.3.7 量子モンテカルロ法のコード

本研究では換装化の対象として汎用量子モンテカルロ計算コード「CASINO」[21] を対象とする．CASINO はケンブリッジ大のグループが開発・保守を行なっている計算コードで，孤立原子系，分子系，固体周期系，モデル系など多彩な対象を，さまざまな基底関数系で扱うことが可能である [22]．CASINO は Fortran90 で記述されたソースコードが公開されているため，独自に機能を付け加えることや改良が可能である．CASINO の他に知られている同種の汎用的大規模コードとしては「CHAMP」,「QMCPACK」,「QWalk」などが挙げられる [23]-[25].

本研究では，次の理由から固体周期系の blip 関数基底による計算に対象を限定する：量子モンテカルロ法による電子状態計算では，固体周期系対象の大規模計算に最も期待が寄せられている．これは電子相関を正確に取り入れた計算手法として，量子モンテカルロ法と対等している分子軌道法（Molecular Orbital method;MO）が，固体周期系には適用できないためである．また固体周期系というのは，実用的な電子デバイスの舞台など，応用の裾野も大きい．故に，固体周期系対象の大規模な量子モンテカルロ計算に期待が大きいのだが，それに伴う計算コストの増大が課題となっている．固体周期系には §2.3.6 で述べたとおり blip 基底計算が一般に用いられる．したがって，blip 基底関数系を用いた固体周期系の量子モンテカルロ計算に対して高速化への需要とインパクトが最も大きく，本研究ではこれに対象を限定した．

第3章　GPGPU

GPGPU の成り立ちや発展は背景に記述した通りなので，この章では GPGPU を用いる上で重要な GPU のアーキテクチャと演算の仕組み，及びプログラミング方法について説明する．

3.1　アーキテクチャ

現在，GPGPU を用いることができる演算コアを搭載したグラフィックボードには NVIDIA 社と AMD 社の製品がある．前者の製品では，近年の GeForce シリーズや Quadro，HPC 向けの Tesla，後者では Radeon というラインナップが挙げられる．製造メーカーやモデルの違いによって，内蔵コアの名称やアーキテクチャが異なる部分があるが，GPGPU を扱う上で理解しておくべき大まかな構造はどちらも同じである．本論文では，NVIDIA 社の GeForce GTX 480 におけるアーキテクチャについて述べる．

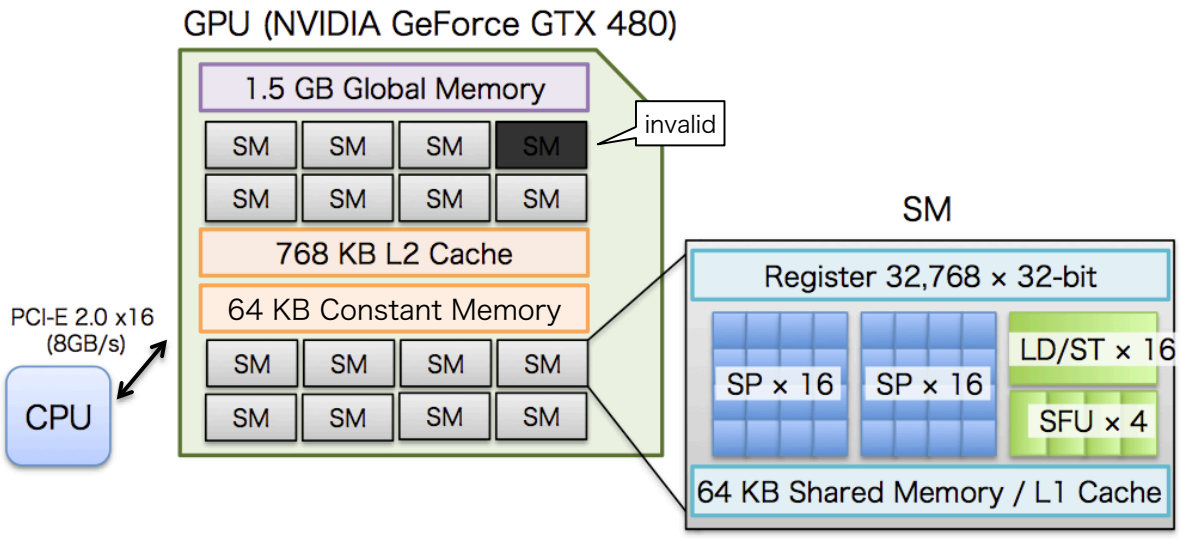

Fig. 3.1: NVIDIA GeForce GTX 480 の演算器とメモリ構成

GeForce GTX 480 の構成を Fig.3.1 に示す．GeForce GTX 480 にはグローバルメモリ，

Streaming Multi-Processor（SM）と呼ばれるクラスタ（演算器群）が16基，L2キャッシュなどが搭載されている．1つのSMには32個のStreaming Processor（SP）が搭載されている．SPには様々な呼び名があり，従来は汎用シェーダユニットと呼ばれていたが，近年，NVIDIAでは「CUDA Core」，AMDでは「Stream Core」と呼ばれているが，本稿はSPで統一する．

SM内には32基のSPに加え，4基の超越関数ユニット（Special Function Unit;SFU）があり，さらにレジスタやL1キャッシュ兼Sheardメモリといったメモリ領域を有する．また，メモリアクセスを行うためのロード/ストアユニット（LD/ST）も16基実装している．各SPには1基の32bit浮動小数点（FP32）スカラ演算器と，32bit整数（Int32）スカラ演算器が内包され，それぞれ並列（1サイクル，2OP）で処理できる．またSPは，2サイクルかけて64bit浮動小数点（FP64）演算と整数演算（Int64）を行うことができる（2サイクル,2OP）．またSFUは指数，対数，三角関数などを取り扱う専用ユニットで1サイクルで4つの浮動小数点演算（1サイクル,4OP）を行える．GeForce GTX 480に搭載されているSM数は16基だが，実際は1つ無効化されていて有効なSM数は15基である．これはGeForce GTX 480が搭載トランジスタ数が30億個と膨大で，かつ汎用GPUである故の歩留まり向上のためとされている．

ここでGeForce GTX480の単精度での理論性能値は次のように計算できる．

$$1.401\text{GHz} \times 15\text{SM} \times (32\text{SP} \times 2\text{OP} + 4\text{SFU} \times 4\text{OP}) = 1681[\text{GFlops}]$$

構造上ではSPとSFUは独立で設置されており同時実行が可能な形になっている．しかし数学関数を使用しない限り同時実行の状態になることはなく，結局，単純な四則演算における単精度ピーク性能は

$$1.401\text{GHz} \times 15\text{SM} \times (32\text{SP} \times 2\text{OP}) = 1345[\text{GFlops}]$$

となり，また倍精度の場合は

$$1.401\text{GHz} \times 15\text{SM} \times (32\text{SP} \times 2\text{OP} \times 1/2) = 672[\text{GFlops}]$$

となる．またGeForce GTX480はHPC向けの上位機種，NVIDIA Tesla 20シリーズとの差別化のため，倍精度性能が単精度性能の1/8になるようにドライバで制限され168[GFlops]程度となる．

このピーク性能を現在のCPUと比較したものをTable 3.1に示す．
理論性能通りならば，単精度ではGPUはCPUの1345/42.56=31.6倍の浮動小数点演算能力を有する．倍精度においても168/42.56=3.9倍の性能である．しかしGPUの場合480個ものコア（SP）があるために，メモリ帯域がCPUに比べ逼迫され，メモリアクセスが律速になりやすい．そのため最近のGPUはCPUと同じように高速にアクセス可能なL1/L2キャッシュやレジスタを設けることにより，レイテンシの大きいグローバルメモリへのアクセスを出来るだけ少なくする構造がとられる．

Table 3.1: 各演算器の理論性能の見積もり

	コア周波数 (GHz)	コア数 cores	ピーク性能 (GFlops)
CPU (Intel Core i7 920)	2.66	4	42.56
GPU (GTX480) Single precision	1.401	480	1345
GPU (GTX480) Double precision	1.401	240	672(limited 168)

3.2 プログラミング

GPGPU のプログラミング言語には CUDA，もしくは OpenCL が用いられる．本研究では前者の CUDA を用いた．

一般的に GPGPU のプログラムは CPU 側で実行されるホストコードと GPU 側で実行されるカーネルコードとに分けて記述する．今の段階ではホストコードは C/C++言語のみがサポートされているが，CASINO のコードは Fortran90 で記述されているため，一度 Fortran プログラムから C（または C++）言語で記述したプログラムを呼び出し，カーネルコードを実行するという方策と取る．CUDA の場合，ホストコード，及びカーネルコードのコンパイルには NVIDIA から無償で配布されている nvcc コンパイラを用いてプログラムをコンパイルする．また商用の PGI コンパイラを用いれば，Fortran のプログラムからでもカーネルコードを呼び出すことが可能である．

GPGPU のプログラミングは次節で述べるスレッドモデルの構造上，アーキテクチャやメモリ特性を理解した上でプログラムを構築する必要がある．このことが GPGPU の専門知識を持たない開発者が躊躇してしまう原因の一つとなっている．本研究では扱わないが，CUDA の場合，カーネルの特別なプログラミングをしなくとも数値計算を GPU で行なってくれるライブラリ API 群「CUDA SDK」が備わっている．例をあげると，BLAS に相当するベクトルと行列の線形計算を行う「CUBLAS」，高速フーリエ変換を行う「CUFFT」などが存在する．いずれのライブラリも，GPU のアーキテクチャにあわせた最大限のチューニングがなされており，CPU 側で実行するホストコードに 1-2 行の API 使用命令を書き加えるだけで CUDA を扱うことが可能となる．GPGPU はハードウェアだけでなく，ソフトウェアの著しい進歩により，研究者にとっての敷居は格段に下がっており，GPGPU の活用の場はさらに広がると予測される．

3.3 スレッディングモデル

GPU はメモリアクセスが非常に律速となるため，この影響を小さくなるように効率的な演算実行モデルが行なわれている [26]．いま，Fig.3.2 のような単純な多重ループの演算

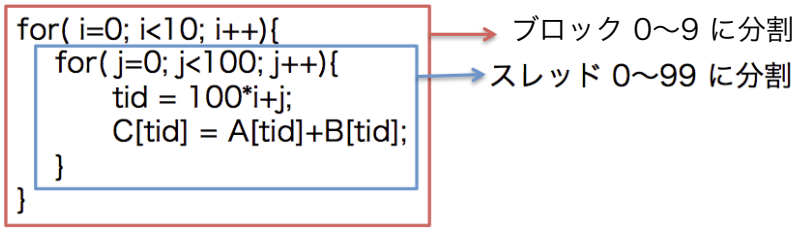

Fig. 3.2: ブロックとスレッドモデル

を GPGPU を用いて並列演算することを考える．

CUDA では他のマルチスレッドプログラミングモデルと同様，依存性のない並列可能な箇所をスレッド（Thread）に分割し，各コアで並列実行を行う．上記のプログラムの場合，内側 j のループを 0〜99 の 100 個のスレッドに分割し，さらにその外側 i のループを 0〜9 の 10 個のブロック（Block）と呼ばれる単位に分割，という方法が，CUDA での単純な実装方式である．このスレッドとブロックの階層を図で表すと Fig.3.3 のようになる．ブロックの集合をグリッドと呼び，実際ブロックは 2 次元，スレッドは 3 次元で管理されるが，ここでは 1 次元で表すことにする．

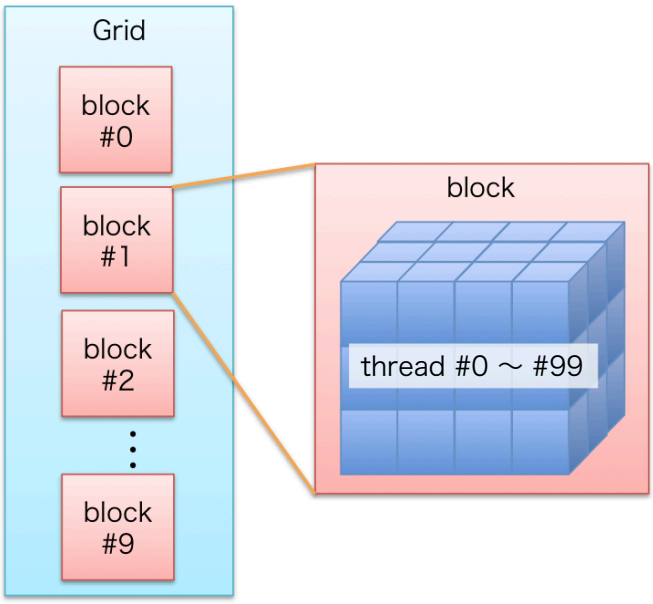

Fig. 3.3: ブロックとスレッドの階層イメージ

演算の際は各ブロックが GPU 上の SM に割り当てられ，SM 内の大量のプロセッサ（SP）でスレッドが並列処理される．このとき，32 スレッドずつ束ねたスレッドバッチ「Warp」

を，ベクタプロセッサとしての実行単位としている．実行可能な状態になっている Warp は，2 組の Warp スケジューラと Warp ディスパッチャによって，順次ストリームに登録されていく．その様子を Fig.3.4 に示す．SP は 16 個ごとにグループ化されているため，

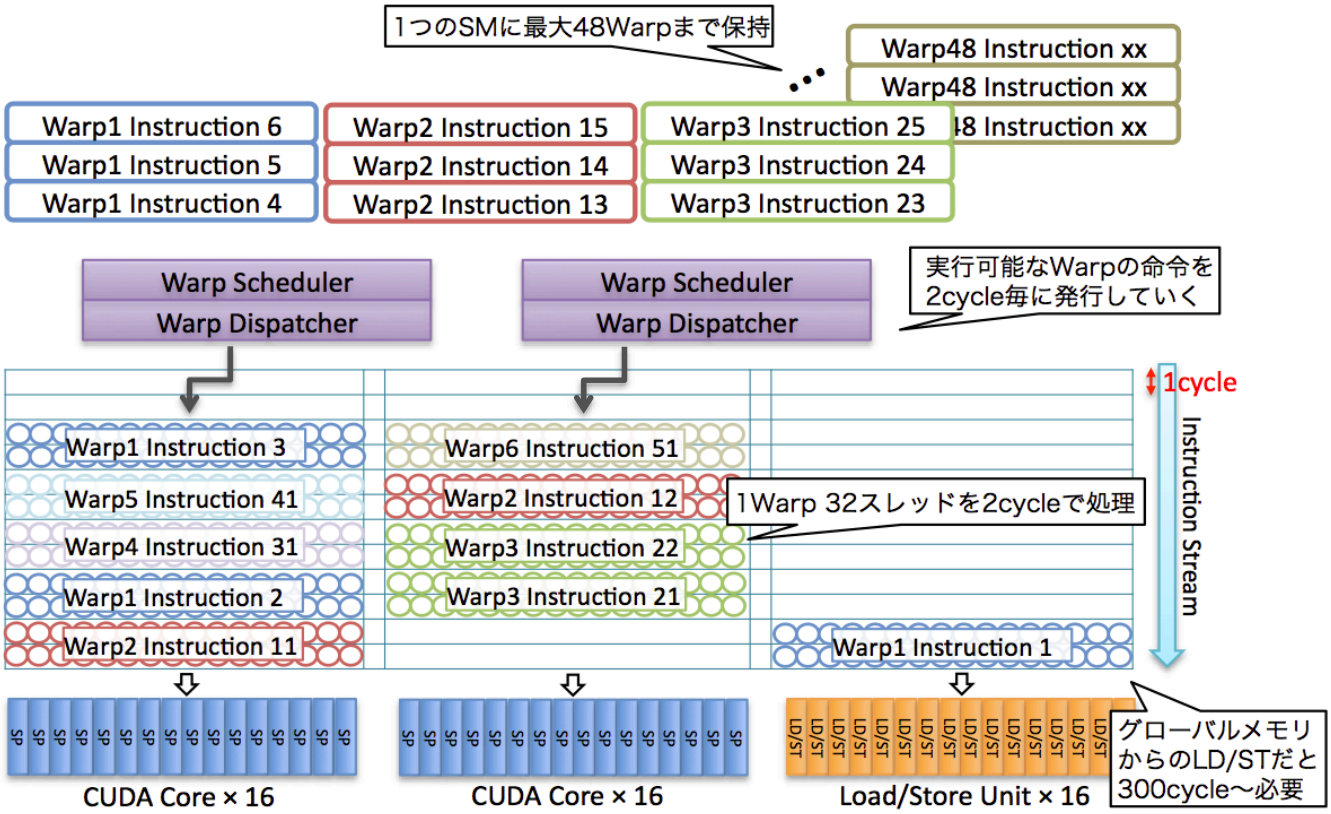

Fig. 3.4: Warp スケジュールの仕組み

2Warp（64 スレッド）を 2cycle かけて処理する形式になる．ここで注目すべきは，先ほど述べたメモリのロード/ストアにおけるレイテンシの問題である．グローバルメモリへのアクセスが発生した場合，最低でも 200cycle，通常で 400〜600cycle もの時間を要する．この間に，SP が使用出来ないのは非常に無駄である．そこで，他の実行可能になっている Warp を先に処理することで，遅いメモリアクセスを隠蔽する仕組みが取られる．したがって，GPGPU のプログラミングには「大量に Warp（スレッド）を発行して，実行可能状態になっている Warp を常に保持しておく」だけの並列度を見出すことが重要である．また，SP や SM の数，ブロックあたりの最大スレッド数などは GPU のアーキテクチャによって変化する．GPGPU で最大の効率を引き出すには GPU アーキテクチャに合わせたプログラミングが必要となってくる．

3.4 GPUの各種メモリ特性と最適化手法

§3.3で述べたとおり，GPUはグローバルメモリへのアクセスが非常に遅く，如何にこれを少なくするかというのが高速化の鍵になる．GPUはその点が考慮されており，高速にアクセス可能な小容量メモリが数種類搭載されている．これらのメモリ群の特性に応じたプログラムにチューニングすることで，GPGPUの更なる高速化が見込める．この節ではメモリの特性と，チューニング手法について述べる．

Table 3.2: CUDAで扱えるGPUのメモリの種類

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	不要	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	L1/L2	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	不要	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	L1/L2	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

CUDAで扱えるGPUメモリをTable 3.2に示す．GPUのメモリは大きくオンチップとオフチップの二種類に分けられる．前者はGPUコア内に内蔵されているメモリで，非常に高速にアクセスできるが，小容量である．後者はGPUの基板上に搭載されているメモリで，オンチップメモリに比べ非常に低速であるが大容量である．またメモリの種類によって，ブロックやスレッドがアクセスできる範囲が異なる．Fig.3.5にその概略を示す．

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（GPU上で処理される関数）上で宣言した変数がここに格納される．SM毎に128KB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倍程度低速であるが，GTX 480のアーキテクチャではL1/L2キャッシュが効き，キャッシュ上にデータがある場合，高速に読み込みができる．しかしながら，すべてのデータが上手くキャッシュに乗るとは限らないので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ごとに実装されているメモリで，ブロック内の全て

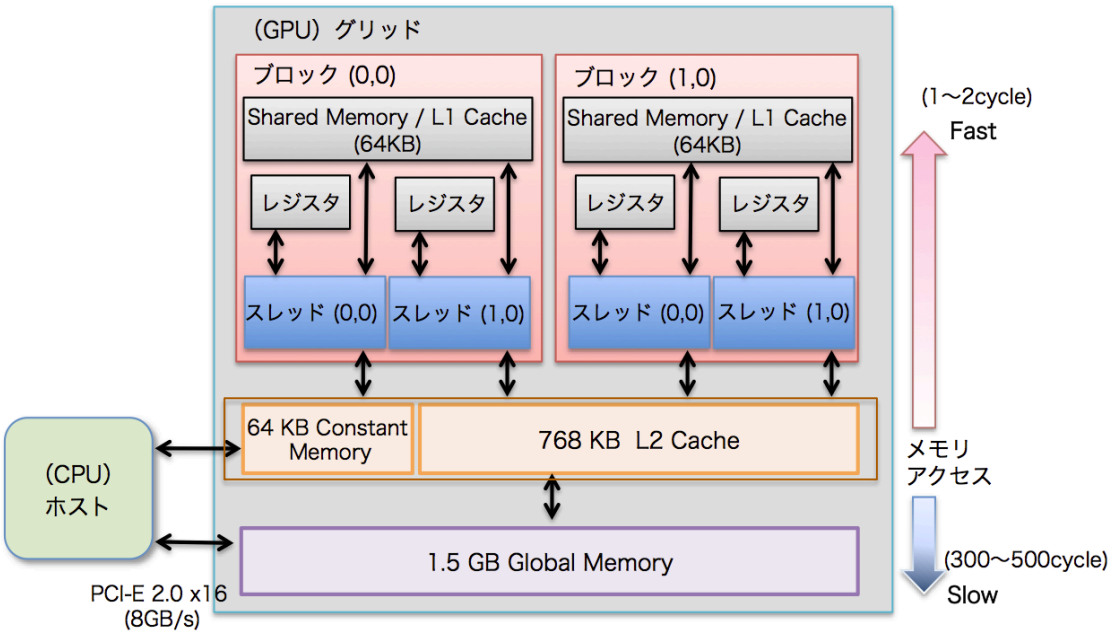

Fig. 3.5: GPU メモリモデルの概要

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．スレッド間でリダクション処理が必要な場合はシェアードメモリが効力を発揮する．シェアードメモリは L1 キャッシュを兼ねた構造になっており，両者のどちらのメモリサイズを優先するか，ユーザー側から指定することが可能である．方法としてはプログラム上で CUDA の API を呼び出し，「シェアードメモリ 48KB/L1 キャッシュ16KB」，もしくは「シェアードメモリ 16KB/L1 キャッシュ48KB」を選択でき，何も記述しない場合はコンパイラの一存になる．これを利用し，シェアードメモリを多く使うプログラムの場合はそちらを優先し，シェアードメモリを全く使わない場合は，L1 キャッシュを優先させることで，グローバルメモリにアクセスする際，L1 キャッシュヒットの向上に期待できる．

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，オンチップメモリと比べて数百倍も低速なメモリであるが L1 や L2 キャッシュが効く場合には高速にアクセスできる．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 1GB〜3GB と非常に大容量である．

コンスタントメモリはチップ外に 64KB 実装されており，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

などで利用されるが本研究では利用しない．

上述した高速にアクセス可能なオンチップメモリは小容量で，プログラム構造上，どうしてもグローバルメモリへ大量にアクセスが発生する場合がある．その際,「コアレッシング」という連続メモリアクセスを用いることで，短時間でのグローバルメモリへのアクセスが可能になる．コアレッシングによるメモリアクセスの概要を Fig.3.6 に示す．

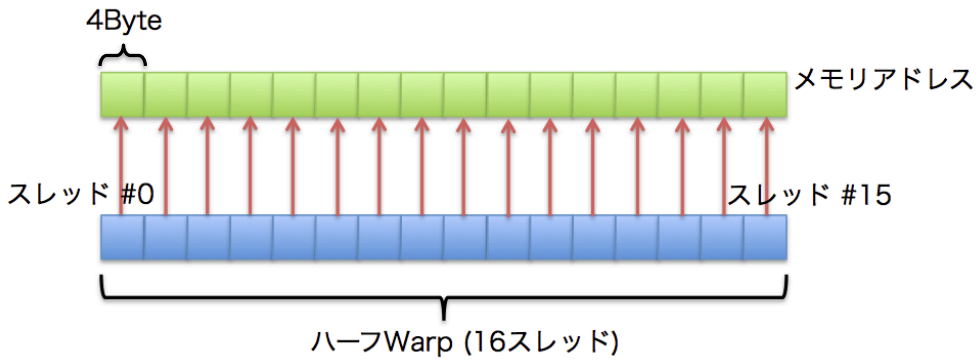

Fig. 3.6: 連続メモリアクセス（コアレッシング）

メモリアクセスは SM 内の Load/Store ユニットによって，ハーフ Warp（=16 スレッド）ごとに行われるが，このとき，16 スレッドのメモリアクセスが連続したメモリアドレスになる場合，単精度では 64Byte（=4Byte × 16 スレッド）を一括して Load/Store することが可能である．もし，ハーフ Warp のメモリアクセスが連続ではなく，ランダムアクセスになる場合，64Byte のメモリを読む動作を最大 16 回処理しなければならず，メモリアクセスが非常に律速になってしまう．このコアレッシングは GPGPU だけでなく，CPU のみで計算する際にも重要な技法であるが，メモリアクセスが律速になりやすい GPGPU では特に意識する必要がある．したがって，GPGPU プログラミングの際はこのコアレッシングを十分に考慮してスレッドやブロックの構造を立てることが重要となる．

3.5 ハイブリッド MPI

本研究の最終的な目標は，MPI 並列計算時における各コアの計算律速箇所に対し，GPGPU を用いた高速化を行うことである．GPGPU の適用手順としては，第一に MPI 並列を用いずに単一 CPU コアのみで GPU への実装を行い，そこで性能が発揮された場合にのみ MPI への拡張を図る方が得策である．したがって，本研究の対象となる量子モンテカルロ計算は MPI 並列を用いずに「1CPU/1 コア/1GPU」という条件で議論を進めている．

先の話として GPGPU 化した量子モンテカルロコードをハイブリッド MPI に対応させ

ていくことになるが，GPU を用いる以外の類似手法として背景で述べた OpenMP がある．この OpenMP を用いたハイブリッド MPI 手法について簡単に述べておく．

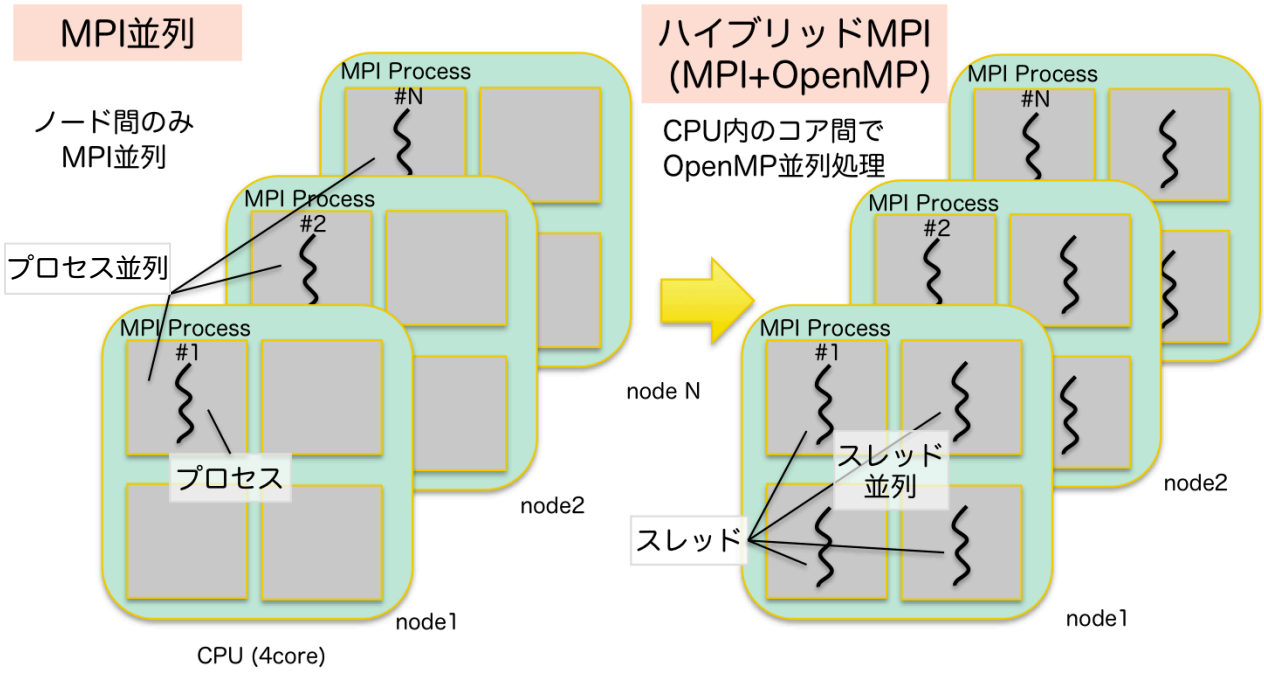

Fig. 3.7: MPI+OpenMP を用いたハイブリッド MPI

スーパーコンピュータにおいてハイブリッド並列が注目されているのはノード間で生じる通信によるものである．というのも数百〜数千ノード間で MPI 通信が生じると通信帯域が逼迫され，ネットワークが律速となってしまう．したがって並列数を増やしても演算時間に比べ通信時間が大きくなり，CPU の性能を引き出せないことになる．そこで Fig.3.7 のように MPI 並列はノード間のみで行い，一つのプロセスの中でさらに並列化できる部分については OpenMP を用いてプロレスをスレッドと呼ばれる実行単位に分ける．そして CPU 内の各コアを用いて並列計算（スレッド並列）を行う．このスレッド並列の際に発生する通信はノード内通信のみであり，高速な通信が可能である．これにより，ノード間に発生する通信量は，従来での全 CPU コアを用いた MPI 並列を行う手法に比べて大幅に削減できる．また，近年の CPU は「同時マルチスレッド処理」に特化している．これは Intel 社の CPU で言えば，1 コアで 2 つのスレッドを同時に処理できる「ハイパースレッディング・テクノロジー（HT）」に相当する．つまり，この同時マルチスレッド処理に対応している CPU ならば，1 コア/2 スレッド（Fig.3.7 の例ならば 4 コア/8 スレッド）の並列処理を行うことで高い演算性能を出すことが出来る．現在，多くのスーパーコンピュータや一般的なコンピュータでは，同時マルチスレッド処理が行える CPU が搭載されている．したがって OpenMP を用いた場合，多くのアプリケーションで高速化が見込

める．

本研究で扱う量子モンテカルロ計算コード「CASINO」は OpenMP に対応しており，問題としている律速箇所もスレッド並列計算が行うことができる．律速箇所について 1CPU で OpenMP を用いた並列計算を行った結果を Table 3.3 に示している．計算対象は 1536 電子系で，使用した CPU は Intel Core i7 920 2.66GHz（4 コア・HT 対応）である．

Table 3.3: OpenMP を用いた際の律速箇所の計算時間

スレッド数	CPU time（ms）	性能加速比
1	210.33	–
2	114.19	1.84
4	67.15	3.13
8	178.55	1.18

結果から 4 スレッド並列時に最大性能を出せることが分かったが，性能加速比は 1 スレッド時の 3.13 倍であり，CPU コア数分の高速化には至っていない．OpenMP を用いた並列計算の場合，スレッド並列箇所について大きな高速化が見込めないことはよく知られている [27]．その点で，OpenMP を超えるの高速化が見込まれる GPGPU を用いたハイブリッド MPI が注目されている．

第4章　実装

変分モンテカルロ計算で律速となっている blip 基底で記述された軌道関数 $\{\phi_j(\vec{r_i})\}$ の再評価ルーチン:blip3d の演算を GPU 側で行なうよう，CUDA への実装を試みた．初期の段階では計算の整合性の確認も兼ねた単純な実装を行ない，次第にチューニングしていくことで高速化を図った．

実装に用いた計算機環境を Table 4.1 に，GPU（Geforce GTX 480）のスペックを Table 4.2 に記載する．またプログラムのコンパイルには Table 4.3 のオプションを用いた．

Table 4.1: 計算ノードの詳細

CPU	Intel Core i7 920 2.66 GHz
GPU	NVIDIA GeForce GTX480 × 1
Motherboard	MSI X58M
Memory	DDR3-10600 2GB × 6
OS	Linux Fedora 13
Fortran/C Compiler	Intel Fortran/C Composer XE 12.0.0
CUDA	CUDA version 4.0

4.1　逐次更新の換装

問題としている blip3d サブルーチンは任意の電子位置 $\vec{r_i}$ が更新された後，§2.3.6 で述べた基底関数展開により次式の積和算を行なうことで軌道関数 $\{\phi_j(\vec{r_i})\}$ を得ている．

$$\{\phi_j(\vec{r_i})\}_{j=1}^{L} = \left\{ \sum_{s=1}^{64} a_{js} \Theta_s(\vec{r_i}) \right\}_{j=1}^{L} \tag{4.1}$$

L は電子の軌道数を表し，電子数 $N = 216$ では $L = 56$，N =1536 では $L = 384$ 程度である．また，軌道関数 $\{\phi_j(\vec{r_i})\}$ は複素数であり，a_{js} は複素数，Θ_s は実数である．上式の処理を CPU のコードから CUDA に単純換装し， 同じ計算結果が保たれることを確認した．プログラミングの際，ブロックとスレッドは Fig.4.1 のように指定した：相異なる格子点

Table 4.2: GPU のスペック

Compute Capability	2.0
Clock of CUDA cores	1401 MHz
Global Memory	1536 MB
Memory Bandwidth	177.4 GB/s
Number of SM	15
Number of CUDA Cores	480
Constant Memory	64 KB
Shared / L1 Memory	16 or 48 KB per block
Register	32 KB per block
Max number of Threads	1024 per block

Table 4.3: 使用コンパイラとコンパイルオプション

Fortran90	ifort -O3 -no-prec-div -no-prec-sqrt -funroll-loops -no-fp-port -ip-complex-limited-range
C, C++	icc -O3
CUDA	nvcc -O3 -gencode arch=compute_20,code=sm_20

s での積算をスレッドにアサインし，軌道のインデックス j を同じくするスレッドが同一ブロックにまとめられる．

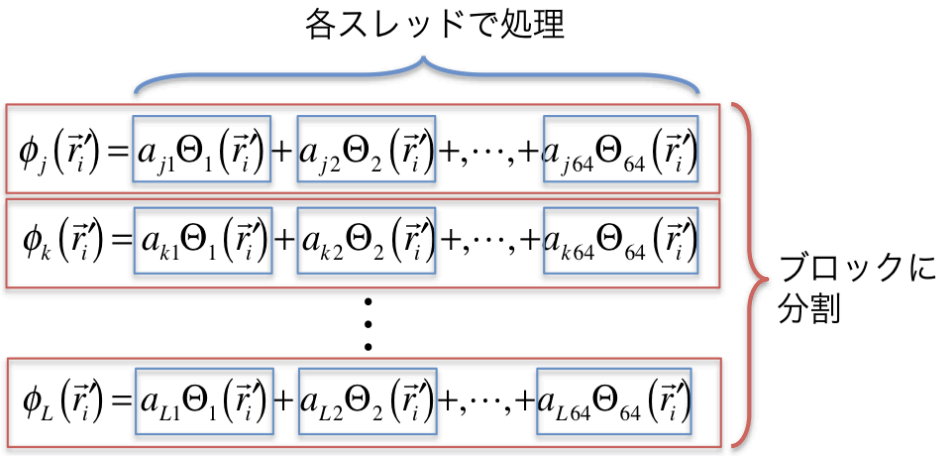

Fig. 4.1: 逐次更新におけるブロックとスレッドの割り当て

まずは，同じ計算結果を保つ GPU 換装を確立することが本節の目的である．そのため，この時点で達成される高速化の結果は Table 5.2 のとおり乏しいものである．1536 電子系に対しては 1.47 倍，216 電子系に対しては 0.41 倍となり却って遅くなっている．この原因として次の問題が挙げられる：

- 並列数が少ない（ブロック数＝軌道数なので L =384 程度，スレッド数は 64 である）
- 各スレッドでの演算量が乏しく，乗算 1 度のみである
- GPU 上のグローバルメモリへのランダムアクセスが発生し，メモリの Load/Store に時間がかかる

中でも最も効果的な改良方策は §3.4 で述べたコアレッシングを適用し，グローバルメモリへのランダムアクセスを減ずることである．これはスレッドを軌道のインデックス j で取ることで，a_{js} 配列への連続メモリアクセスを実現できるが，対するブロックを総和の s で取らなければならなくなり，ブロック間のリダクションを行う必要がある．この場合，遅いグローバルメモリを使ったリダクション処理になり，結果的にパフォーマンスがより悪化した．またブロック数を 1 として，各スレッド内のループ処理で s の総和をとるという手法も考えられるが，ブロック数=1 というのは GPU 内に 15 ある SM のうち 1 つしか使用しないため，極端にパフォーマンスが低下する．残る実現可能な改良手法は，並列数を増やし，メモリアクセスの律速を隠蔽することである．これらの理由から，更なる並列数を得るべく計算アルゴリズムの改変を模索した．

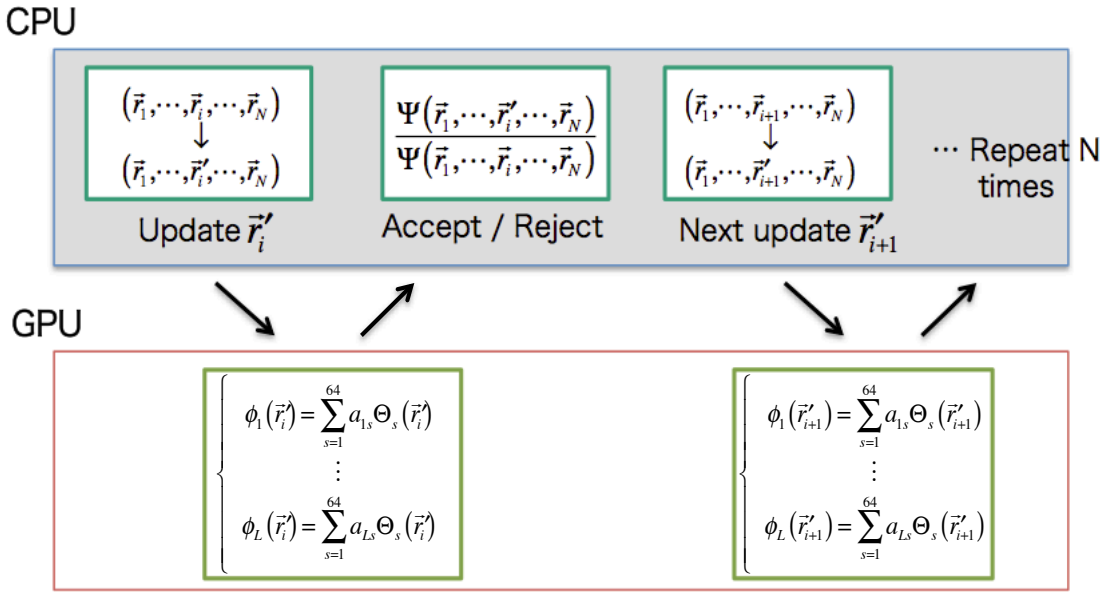

Fig. 4.2: 逐次更新アルゴリズム

単純実装（逐次更新の換装）段階における計算の流れを Fig.4.2 に示す．任意の電子位置を CPU 上で更新後 $(\vec{r}_i \to \vec{r}_i')$, 試行関数を構成する $\phi_j(\vec{r}_i')$ の再評価のために，それを得るのに必要なデータを GPU に転送し，$\phi_j(\vec{r}_i')$ を GPU 上で構築する．構築された $\phi_j(\vec{r}_i')$ は CPU へ戻し，試行関数を構成後，更新された電子位置の棄却/採択を行い，また次の電子位置を更新する，という逐次更新の流れになる．上述の流れにおいて，$\vec{r}_i$ と $\vec{r}_{i+1}$ の更新はランダムで行なわれるため無関係である．したがって，$\vec{r}_1$ から $\vec{r}_N$ までの全電子の軌道関数 $\phi_j(\vec{r}_1')$ から $\phi_j(\vec{r}_N')$ を GPU 上で一度に処理する構造（斉次更新）に変更できれば，GPGPU 並列数を逐次更新の電子数 N 倍にすることが可能である．次節ではこの斉次更新の実装について述べる．

4.2 斉次更新の実装

4.2.1 斉次更新処理

GPU 上の並列処理を増やすため，Fig.4.3 に示す斉次更新の実装を考案した．この実装では，事前に CPU 上で $\vec{r}_1'$ から $\vec{r}_N'$ の全ての電子位置に関する試行更新を準備し，まとめて GPU にデータを転送，$\phi_j(\vec{r}_1')$ から $\phi_j(\vec{r}_N')$ を GPU 上で一度に構築する．

逐次更新と斉次更新では，サンプリングのシーケンスが異なるため，両者の結果は数値としては一致せず，統計誤差の範囲内での一致となる．デバッグを含めた換装を段階的に

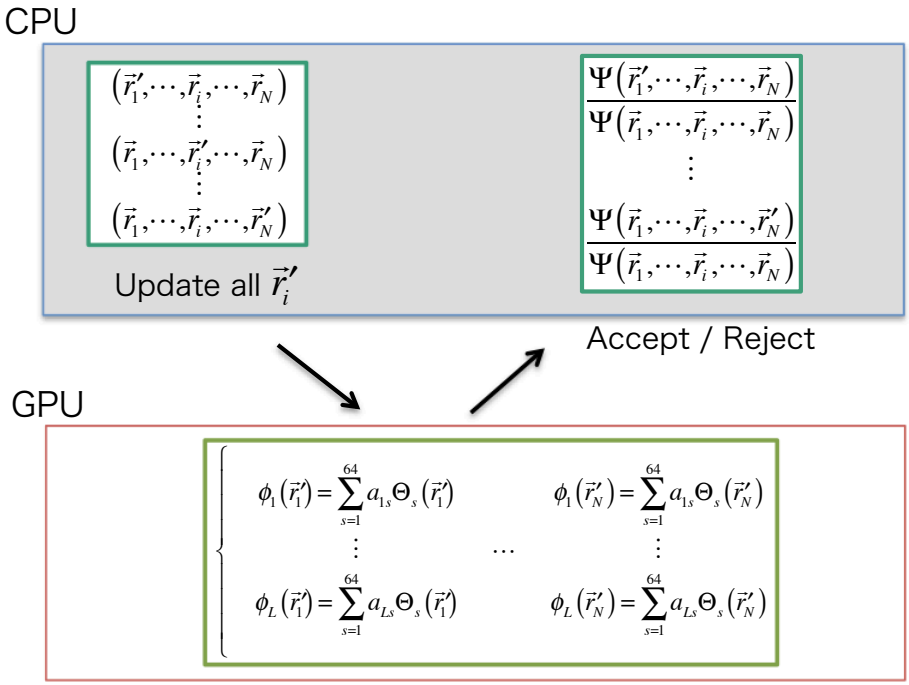

Fig. 4.3: 斉次更新アルゴリズム

進めるために，まずは斉次更新の CPU 処理版を構築し，統計誤差範囲での一致を確認の上，斉次更新 CPU 版を段階的に GPU に換装した．

4.2.2 斉次更新の GPU 換装

次に斉次更新処理を以下のように GPU 上に換装した：この段階では，§4.1 で行なったのと同様に，相異なる格子点 s での積算をスレッドにアサインする（スレッド内の演算は唯一つの積算のみ）．斉次更新への変更に伴い，軌道のインデックス j に加え，電子のインデックス i に関する並列性が加わる．これを (j,i) という二次元構造のブロックで処理する（→ Fig.4.4）．

結果が Table 5.2 に示されている．スレッド数は 64 で，逐次更新の場合と変わらないがブロック数が（電子数 N ×軌道数 L）に増加したことで，並列度が大幅に向上し，216 電子の場合，6.39 倍，1536 電子の場合には 5.61 倍の高速化が達成された．

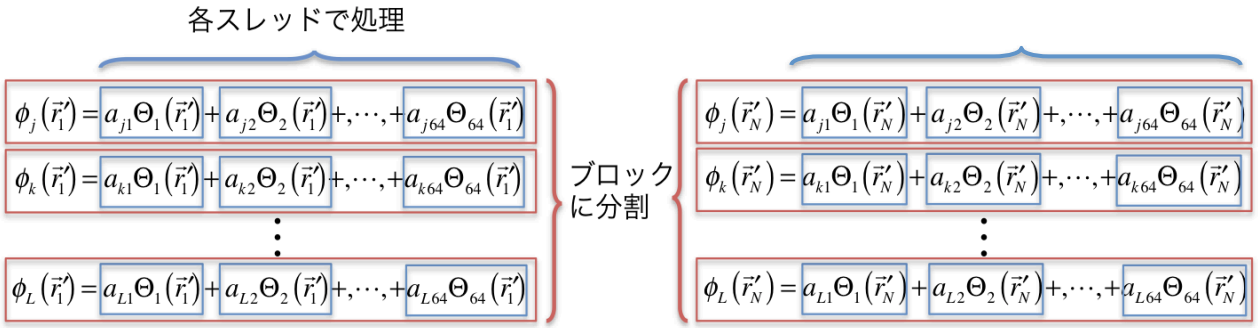

Fig. 4.4: 斉次更新におけるブロックとスレッドの割り当て

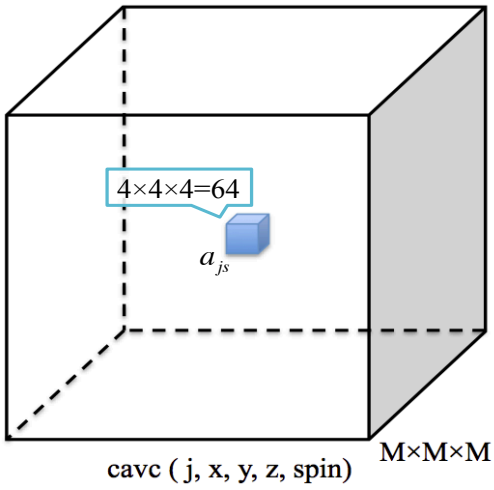

Fig. 4.5: cavc と a_{js} の関係

4.2.3 スレッディング構造の改変と最適化

前節までの逐次更新および斉次更新では，積和算

$$\sum_{s=1}^{64} a_{js}\Theta_s\left(\vec{r}_i\right) \tag{4.2}$$

において，各スレッドで $a_{js}\Theta_s\left(\vec{r}_i\right)$ を計算し，スレッド間のリダクションにより総和が計算された．係数 a_{js} のデータ構造に着目し，ランダムアクセスを解消するようなスレッディング構造の組み換えを試みたところ，大きな性能向上を達成することが出来た：係数 a_{js} は $cavc$ という配列名で，$cavc(j,x,y,z,spin)$ の 5 次元配列に格納されている．j は軌道のインデックス，x,y,z は格子点のインデックス，spin は電子スピンに関するインデックスである．本研究では非偏極系を取り扱うため，スピンのインデックスは退化・消失する．x,y,z に関する要素数は $M\times M\times M$ で，216 電子系の場合 $M=60$，1536 電子系では $M=50$ である．

グローバルメモリ上に置かれた $cavc(j,x,y,z,spin)$ に対し，各電子位置 $\vec{r}_i$ を担当するスレッドは，$\vec{r}_i$ 近傍の $4\times4\times4=64$ 点の格子点での a_{js} の値にアクセスする（→ Fig.4.5）．前節までの逐次更新，および，斉次更新の実装では相異なる s がスレッドにアサインされているため，s を足とする連続メモリアクセスが発生するが，$cavc$ 配列はメモリ構造上，j を足にとる 1 次元配列である．したがって，$cavc$ へのアクセスはメモリアドレス上，飛び飛びのランダムアクセスとなってしまう．この場合，§3.4 で述べたとおり，レイテンシの大きいグローバルメモリへのアクセスが大量に発生し，高速化を妨げる．これを避けるには，各スレッドが，軌道のインデックス j を足とした 1 次元配列 $cavc$ への連続アクセスを構成するようにスレッディングを組み替え，コアレッシングを行なう必要がある．この場合，相異なる軌道 j がスレッドにアサインされることになる．スレッド内の演算は，ある一つの軌道関数を構築する (4.2) 式の 64 項の積和算となる．残されたインデックスである i（電子インデックス）についてブロック化を行なった．すなわち，同一の電子インデックス i を有する複数のスレッド（L 本）がブロックにまとめられ処理される（→ Fig.4.6）．

前節までの実装ではスレッドで並列処理されていた s =1〜64 の積和算は，本実装では，各スレッドのループ内で処理される．この構造により，従来ではスレッド間で行なっていたリダクション処理の必要がなくなった．j をスレッド並列にしたことで，以前の実装での $\Theta_s\left(\vec{r}_i\right)$ に関する連続アクセスが失われる．しかしながら，一つのブロック（電子インデックス i が固定）は，高々，64 要素の同一の $\Theta_s\left(\vec{r}_i\right)$ を参照するため，再利用性があり，キャッシュによる効果が期待される：ある Warp が一度，グローバルメモリ上にある $\Theta_s\left(\vec{r}_i\right)$ にアクセスすると，その近傍の配列は，L1 や L2 キャッシュに保持され，キャッシュヒットが起きやすく，$\Theta_s\left(\vec{r}_i\right)$ に関する連続アクセスの喪失はメモリアクセス遅延上の問題にはならないと期待される．

結果が Table 5.2 に示されているが，この実装で最終的に 216 電子の場合，16.58 倍，1536 電子の場合には 30.67 倍の高速化が達成された．

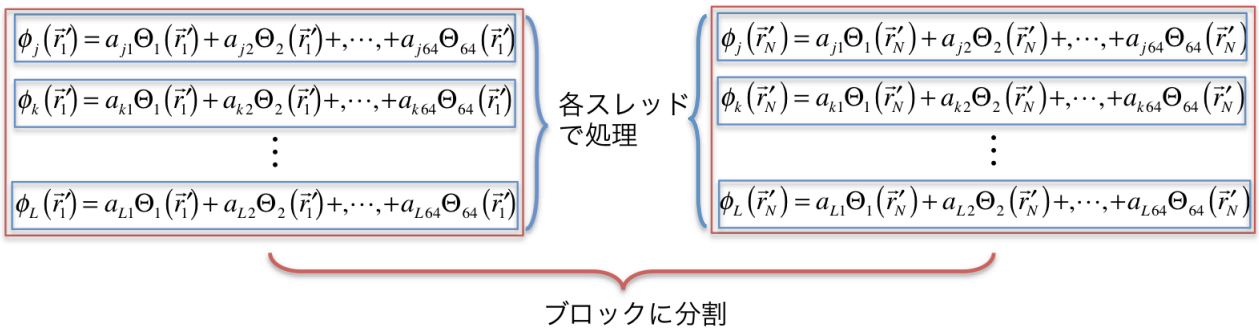

Fig. 4.6: 最適化によるブロックとスレッドの割り当て

第5章　結果と考察

5.1　実装結果

5.1.1　演算時間の比較

Table 5.1 には本研究で開発した各種カーネルでの，216 電子系と 1536 電子系について実行時間が示されている．また Table 5.1 に基づき，性能比較を行ったものを Table 5.2 に示す．これらの比較は，GPU 上の演算コアへの並列タスクを異なる割り当ての仕方で行なった場合の比較に相当する．

Table 5.1: 計算律速箇所の演算時間

	$N = 216$ (ms)		$N = 1536$ (ms)	
	CPU	GPU	CPU	GPU
(1) 逐次更新	2.77	6.76	100.00	68.03
(2) 斉次更新		0.43		17.83
(3) 斉次更新+最適化		0.17		3.26

Table 5.2: 計算律速箇所の性能加速比

	Index		性能加速比	
	block	thread	$N = 216$	$N = 1536$
(1) 逐次更新	j	s	0.41	1.47
(2) 斉次更新	j, i	s	6.39	5.61
(3) 斉次更新+最適化	i	j	16.58	30.67

(1) 逐次更新から (2) 斉次更新への変化では，GPU 上で処理される総スレッド数が電子数の N 倍増えた事で演算性能が向上している．(2) 斉次更新から (3) 斉次更新+最適化への

変化では，スレッド割り当ての設計を変更した事で，(i) スレッドが処理する演算数を増やしたこと，(ii) グローバルメモリへの連続アドレスでのアクセスを可能とし，コアレッシングが効くようになったこと，の 2 点で性能が伸びている．

5.1.2 システムサイズの変化

GPU 換装版を Table 5.2 の (3) 斉次更新+最適化の版に固定し，性能向上が対象系のシステムサイズにどのように依存するかを調べた．$N = 216$ ($Si/2\times2\times2$), 648 ($TiO_2/3\times3\times3$), 1536($TiO_2/4\times4\times4$) に対する結果が Table 5.3，および Fig. 5.1 に示されている．

Table 5.3: システムサイズの変化による性能加速比

電子数	軌道数	CPU time (ms)	GPU time (ms)	性能加速比
216	56	2.77	0.17	16.58
648	168	20.21	0.82	24.46
1536	384	100.00	3.26	30.67

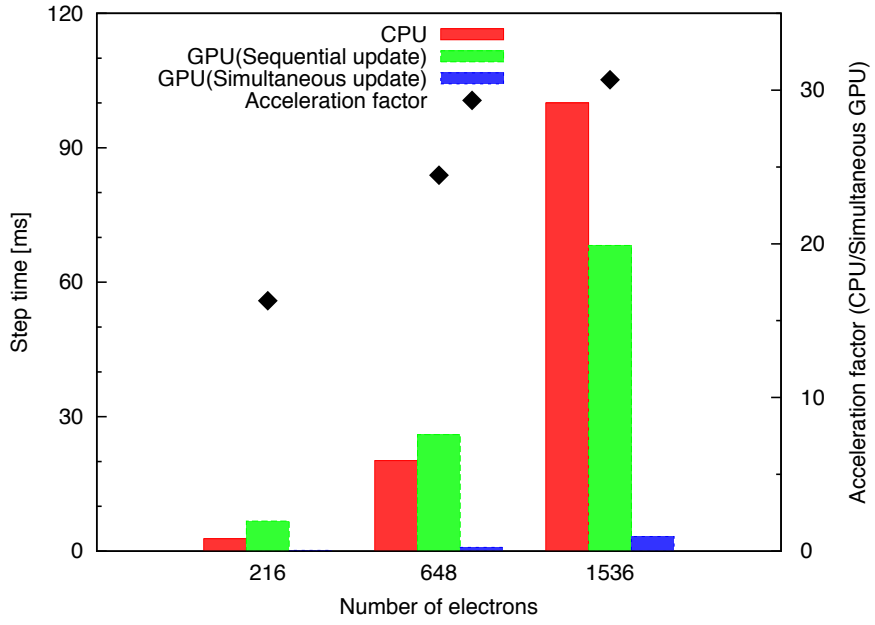

Fig. 5.1: CPU/GPU time と性能加速比

電子数 N が増えると，一般に軌道数 L もほぼ比例して増える．したがって，GPU 上で処理される総スレッド数は，$N \times L \sim N^2$ で増え，GPU 処理はより効率的となる．ただ

し，演算処理後に GPU から CPU に返される $\left\{\{\phi_j(\vec{r}_i)\}_{j=1}^{L}\right\}_{i=1}^{N}$ の転送量も増大するから，トータルの演算性能は N^2 では増えていない．計算コードの用途を考えると大電子数でほど性能向上が見込める換装は好ましいものである．ただし，軌道数 L がブロック内のスレッド容量となっており，これが 1,024 を超えると現行実装では取り扱う事が出来なくなる．非磁性固体の場合，大凡，$N \sim 4L$ なので，電子数にして 4,096 電子となる．磁性物質の場合，スピン多重度が復活し，軌道を識別するスピンインデックスの退化が解消するため，$N \sim 2L$ となり，2,048 電子，更に時間反転対称性のない系では軌道インデックスに成立するある種の符号反転対称性が破れるため，$N \sim L$ で 1,024 電子となる．当該計算手法で扱える電子数は，現行でせいぜい 1,000 電子程度であり，且つ，対称性のよい非磁性固体が主要な対象となっているから 4,000 電子程度まで扱える実装は十分現実的なものである．ブロックあたりに収容されるスレッド数はデバイス技術の進展で倍々で増大していくものと期待されるので，この点は重大な懸念とはならないだろうと予測される．

5.1.3 演算性能の考察

GPU に換装した各カーネルがどの程度の演算性能を出しているか次の条件で調査した：実性能を計測するために CPU と GPU 間のデータ転送時間は含めず，カーネル部の時間のみ測定した．各カーネルで行われる浮動小数点演算回数は次のように求められる．
逐次更新：

$$2OP(\text{積和}) \times 64 \times L \times 2(\text{実部・虚部})$$

斉次更新：

$$2OP(\text{積和}) \times 64 \times L \times N \times 2(\text{実部・虚部})$$

CPU の理論性能（Flops）は Table 3.1 に記載されている CPU の 1 コア分の性能（42.56/4=10.64 GFlops）とし，GPU は単純演算における理論性能（1345 GFlops）の値を用いて算出した．計算は 1536 電子の TiO_2 で行った．その結果を Table 5.4 に示す．

Table 5.4: 律速箇所の Flops と理論性能比

	演算性能 (GFlops)	理論性能比
(1) CPU/逐次更新	1.50	14.10 %
(2) GPU/逐次更新	4.62	0.34 %
(3) GPU/斉次更新	9.05	0.71 %
(4) GPU/斉次更新+最適化	73.98	5.50 %

カーネルの改良に伴い，演算性能が向上していく様子が分かる．ところが，30.67 倍の高速化を達成した (4) の最終的なカーネルでさえ，GPU のピーク性能には遠く及ばず，未だ改良の余地があるかのように見える．

ここで他の GPGPU パッケージの理論性能比の例を挙げる．線形代数計算ライブラリ「BLAS」を CUDA 用にチューニングした「CUBLAS」がある．これの BLAS Level3（行列積）の場合，NVIDIA Tesla C2050 での演算性能は 400 GFlops に達しており，GPU 演算の手本とされているが，理論性能の観点では 38.8%しか出ていない [28]．つまり GPGPU の場合，実用的な演算における性能はどんなにチューニングを行なっても理論性能の近傍に到達することは難しいと言える．これはグローバルメモリへのメモリアクセスが律速になる事が大きい．上記の理由に加え，本研究の場合，いずれのカーネルにおいても §4.2.3 で述べたように，$\vec{r}'$ の位置から，cavc 内の 64 点の格子点 $\{a_{js}\}$ を求める必要がある．そのため，GPU カーネルでは積和算以外にも幾らかの演算を行なっている．これらの実情から，本研究で最終的に達成した 5.50 %の理論性能比は十分なものだと考えられる．

5.1.4 エネルギー値の比較

各カーネル実装に対して計算されたエネルギー値の比較を Table 5.5 に示す．216 電子のシリコン固体の単位胞あたりの基底エネルギー値が示されている．統計蓄積条件は全て共通に 10,000 ステップ × 100 ブロックとし，エネルギー値のサンプリングは 10 ステップおきに行なった．

Table 5.5: 各実装カーネルにおけるエネルギー値の比較

	Energy (hartree/cell)
(1) CPU/逐次更新	-7.9590865 ± 0.0001749
(2) GPU/逐次更新	-7.9589381 ± 0.0001754
(2) CPU/斉次更新	-7.9591560 ± 0.0001755
(4) GPU/斉次更新	-7.9592106 ± 0.0001698

「CPU/逐次」の結果がオリジナルの実装での値であり，換装に際しての参照標準となる．「GPU/逐次」との比較は，アルゴリズム構造の変更なく GPU で単精度化した時の誤差に対応するが，以下に述べるように注意が必要である：結果は，有効桁数 3 桁目まで一致している．電子状態計算の実用上要求される精度はこの単位系で小数点以下 3 桁目までで，統計計算の誤差範囲を考えると実用には耐える精度を保っている．ただし，単精度の有効桁数が 6 桁程度であることから考えると非常に一致が悪いという印象を与える．これは総計 1,000,000 回のステップに亘る誤差の蓄積によるものである．実際，1 ステップで比較

すると Table 5.6 に示すように有効桁数 9 桁までの一致を示す.

Table 5.6: 1 更新ステップにおけるエネルギー値

	Energy (hartree/cell)
(1) CPU/逐次更新	-7.95075065699
(2) GPU/逐次更新	-7.95075065360

僅かな誤差によって，棄却採択に一回でもズレが生じると，乱数のシーケンス自体がずれをきたす．したがって,「確率論的なアルゴリズム構造」は変更がないものの，事実上の算法構造は変わってしまっている点に注意が必要である.

次に,「CPU/逐次」と「CPU/斉次」との比較に注目する．これは CPU 上の倍精度の扱いに固定したまま，モンテカルロ更新の算法構造を変えた場合の変化に相当している．この場合に期待されるのは統計誤差の範囲内での一致となるが，結果は，この一致を保証している.「CPU/斉次」と「GPU/斉次」との比較が，単精度化に伴う誤差に相当する．この場合の誤差蓄積の要因についても，上述の逐次更新法での比較の場合と同様である．総じて，単精度化や斉次更新化に伴う誤差は，実用上問題ないことが分かる.

5.1.5 最適化の効果

§4.2.3 に述べた斉次更新版の最適化について，その効果を検証する．解析には CUDA 用のプロファイラ「Compute Visual Profiler」を使用し，次の要点に絞って分析した：

- コアレッシングによるグローバルメモリのロードの影響
- 演算コア群（SM）の占有率

これらの解析結果を Table 5.7 に示す.

(1) 逐次更新版では，GPU 演算の並列数が少なく（軌道数 $L \times 64$），グローバルメモリのロード回数が斉次更新版に比べ少ない．$\{a_{js}\}$ に対するランダムアクセスが発生するので，グローバルメモリのロードスループットも 13.5 GB/s と理論性能値 (177.4 GB/s) に届いていない．SM activity は演算コア群（SM）の占有率を意味し，次式で計算される.

$$\text{SM activity} = \frac{\text{実行待ち状態の Warp がある cycle 数}}{\text{全体の cycle 数}}$$

つまり SM activity が高い程，グローバルメモリのアクセス隠蔽が可能であるということ

Table 5.7: GPU プロファイラによる性能解析

	グローバルメモリ		SM activity
	32bit ロード回数	スループット (GB/s)	
(1) 逐次更新	24,576	13.5	88.1 %
(2) 斉次更新	188,744,000	18.1	99.4 %
(3) 斉次更新+最適化	7,077,890	153.0	100.0 %

を示す．(1) 逐次更新版の SM activity は 88.1%で，残りの 11.9%の時間はメモリのアクセス待ちのため，演算コアが全く使われていない状態になっていることが分かる．

対して，(2) 斉次更新版の結果では，さらに電子数 N の並列数を乗じたため，SM activity は 99.4%に向上している．しかし，ランダムアクセスの問題からやはりグローバルメモリアクセスのスループットはほとんど改善されていない．$\{a_{js}\}$ に対する連続アクセスを考慮した (3) 斉次更新の最適化版ではグローバルメモリへのアクセス回数が大幅に減っており，コアレッシングが有効になっていることが分かる．また，スループットも理論性能値に近い値を達成しており，最適化が非常に有効であったと言える．

5.1.6 CUBLAS

CUDA では，線形計算に佳くチューニングされた CUBLAS が提供されている．これを効率的に利用する余地についても考察を行った．(4.1) 式の演算は，

$$\begin{pmatrix} \phi_1(\vec{r}_i') \\ \phi_2(\vec{r}_i') \\ \vdots \\ \phi_L(\vec{r}_i') \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1,64} \\ a_{21} & \ddots & & \vdots \\ \vdots & & & \\ a_{L1} & \cdots & & a_{L,64} \end{pmatrix} \begin{pmatrix} \Theta_1(\vec{r}_i') \\ \Theta_2(\vec{r}_i') \\ \vdots \\ \Theta_{64}(\vec{r}_i') \end{pmatrix} = A \cdot \begin{pmatrix} \Theta_1(\vec{r}_i') \\ \Theta_2(\vec{r}_i') \\ \vdots \\ \Theta_{64}(\vec{r}_i') \end{pmatrix} \tag{5.1}$$

とまとめることができ，一見，斉次更新の場合

$$\begin{pmatrix} \phi_1(\vec{r}_1') & \phi_1(\vec{r}_2') & \cdots & \phi_1(\vec{r}_N') \\ \phi_2(\vec{r}_1') & \ddots & & \vdots \\ \vdots & & & \\ \phi_L(\vec{r}_1') & \cdots & & \phi_L(\vec{r}_N') \end{pmatrix} = A \cdot \begin{pmatrix} \Theta_1(\vec{r}_1') & \Theta_1(\vec{r}_2') & \cdots & \Theta_1(\vec{r}_N') \\ \Theta_2(\vec{r}_1') & \ddots & & \vdots \\ \vdots & & & \\ \Theta_{64}(\vec{r}_1') & \cdots & & \Theta_{64}(\vec{r}_N') \end{pmatrix} \tag{5.2}$$

と行列積の形にまとまるように見えてしまう．実際には，上記の行列 A は

$$A = \{a_{js}\} = \left\{a_{j,s(\vec{r}_i')}\right\} = A(\vec{r}_i')$$

である．上式の $s\left(\vec{r}_i'\right)$ は $\vec{r}'$ の位置から求められる部分格子の座標 (x,y,z) 配列で，$\left\{a_{j,s\left(\vec{r}_i'\right)}\right\}$ は総体 $cavc\,(j,x,y,z,spin)$ から組まれる．したがって

$$\begin{pmatrix} \phi_1\left(\vec{r}_i'\right) \\ \phi_2\left(\vec{r}_i'\right) \\ \vdots \\ \phi_L\left(\vec{r}_i'\right) \end{pmatrix} = A\left(\vec{r}_i'\right) \cdot \begin{pmatrix} \Theta_1\left(\vec{r}_i'\right) \\ \Theta_2\left(\vec{r}_i'\right) \\ \vdots \\ \Theta_{64}\left(\vec{r}_i'\right) \end{pmatrix} \tag{5.3}$$

であり，(5.2) 式のようにまとめることは出来ない．

そこで，(5.3) 式の行列ベクトル算に CUBLAS を適用し高速化を図る方策についても検討した．この場合，CUBLAS の呼び出しに際し，CPU 上で行列 A を構築する必要が生じる．この A に相当する配列を構成するのに結局，相当の計算時間を割いてしまい，十分な高速化が望めないことがわかった．

別の方策として，$cavc\,(j,x,y,z,spin)$ にマスク処理を掛け，大きな定数行列にして線形計算に持ち込む事も可能であろう．$cavc$ は定数で，原理的には GPU 上に常駐し転送は 1 回で済むはずのものだが，CUBLAS の使用法では，線形計算のたびに GPU への転送を行うような使い勝手となり，性能向上に資するという見込みは薄い．

5.2 将来的な課題

現実的に可能な性能向上方策を具体的に述べる．

5.2.1 実性能の向上

本研究では，カーネルとしては 30 倍超の高速化を達成した．実用上の実効性能としては，1.06 倍程度しか達成されない．これは，換装対象とした「blip3d」につき，実計算における全ての呼び出しを換装していないからである．CPU 版での実計算では，本研究が換装したサブルーチン「blip3d」は 3 カ所から呼び出される：「(1; equil) メトロポリス法での統計分布関数の平衡化を行う部分．(2; trial) 配位の試行更新 $\{\vec{r}_i \to \vec{r}_i'\}$ の棄却採択を評価する部分からの呼び出し．(3; eval) 採択が確定した新配位 $\{\vec{r}_i'\}$ に対して，エネルギー値をサンプリング評価する部分」．3 カ所とも，軌道関数値 $\{\phi_j\left(\vec{r}_i'\right)\}$ が必要となる箇所である．(3; eval) に関しては更に，軌道関数の微分値 $\left\{\vec{\nabla}\phi_j\left(\vec{r}_i'\right)\right\}$, $\{\nabla^2\phi_j\left(\vec{r}_i'\right)\}$ が必要となる．(1; equil) は計算冒頭の 1000 ステップ程度の呼び出される．(2; trial)，(3; eval) は，モンテカルロステップのループ中から引用され，ユーザが指定した統計蓄積回数分（典型的には数億回）呼び出される．

オリジナルの CPU 版では，N 電子の位置セットは逐次更新法で更新される．GPU 換装の逐次更新版であれば，上記 3 カ所からの呼び出しを全て GPU 版に置き換えることは

容易である．我々の最終的な換装では，斉次更新法を採用しているため，呼応して，CPU側からの呼び出し構造にも変更を施す必要がある．本研究では，カーネル開発と性能評価にまずは集中するため，換装がたやすく，律速に関わる (2; trial) の部分のみの換装を取り扱った．換装のたやすさは，(1; equil) と (2; trial) では同程度である．(1; equil) は律速部分ではないので，今回は換装していないが，この部分の換装はたやすい．(3; eval) の部分は gradient と laplacian を更に評価する換装が必要となるが，算法構造上，原理的な難点はなく，拡張は straight forward である．これを換装すると，GPU 上での演算量が増えるから，換装による性能はより上がると期待される．

(2; trial) と (3; eval) の呼び出し回数の比率はユーザ指定で設定される．このパラメタ名を「dcorr (integer)」と書くことにする．'dcorr=4' とは，4 モンテカルロステップ毎にサンプリングを行う動作に対応する．1 回の試行更新が 1 モンテカルロステップの定義である．'dcorr $\neq$ 1' と採る理由は，サンプリングのタイミングにある程度のスパンを設けて，疑似乱数に起因する自己相関を除去するためである．実用上は'dcorr $= 5 \sim 10$' と採られる [29]．GPU 換装は (2; trial) に対してのみで，(3; eval) の部分は換装していないから，実計算性能は，dcorr の値に依存する．この dcorr の依存を詳しく調べるため，CPU 版においてプロファイラ（Intel VTune Amplifier [30]）を用いてプログラムの性能解析を行った：1536 電子の TiO_2 の系について，dcorr を 1,5,10 と変化させ，(1; equil）:100 step，(2; trial）:10,000 step の蓄積条件で評価を行なった．その結果を Table 5.8 に示す．

Table 5.8: blip3d ルーチンが占める計算時間の割合

dcorr	(1; equil)	(2; trial)	(3; eval)	total (blip3d)
1	0.2%	2.4%	36.8%	39.5%
5	0.1%	6.8%	20.5%	27.5%
10	0.1%	8.7%	13.2%	22.0%

dcrr が増加するとモンテカルロステップが増えるので，(2; trial）から呼び出される blip3d の割合も大きくなる．また blip3d だけでなく，その他のルーチンの律速も大きくなり計算全体を占める blip3d の割合は小さくなる．

次に GPU カーネルを適用した実性能の結果を Table 5.9 に示す．dcorr が大きくなると，GPU 換装化していない blip3d が呼び出される頻度が減るから，相対的に性能は向上している．但し，dcorr を大きくすることで，blip3d 以外の次律速ルーチンが占める割合が増加するという要因のため，上述の理由による性能向上が隠されてしまい，それほど大きな性能向上にはならないという点がある．なお，blip3d 以外の律速部分として現れる箇所は，BLAS のライブラリを用いて計算される箇所で，行列の内積 (ddot) やベクトル加算 (daxpy) 部分である．したがって，この部分に GPGPU を適用すれば，更なる計算全体の

Table 5.9: dcorr 値を変化させた際の実計算時間

dcorr	Total time (sec) with GPU	CPU	実性能向上比
1	503	515	2.33 %
5	881	923	4.55 %
10	1,345	1,430	5.94 %

高速化が図れるのではないかという予測が立つ.

5.2.2 転送の隠蔽

本研究での実装では，CPU-GPU 間のメモリ転送が結構大きなサイズとなり，その転送に Table 5.10 に示すように GPU 実行時間の半分程度を要している.

Table 5.10: 斉次更新版における CPU-GPU 間のメモリ転送時間

電子数	CPU → GPU (μs)		GPU → CPU (μs)	Total (μs)	GPU 実行時間の割合
	$s\left(\vec{r}_i'\right)$	$\Theta_{s\left(\vec{r}_i'\right)}\left(\vec{r}_i'\right)$	$\phi_j\left(\vec{r}_i'\right)$		
216	16.8	19.5	29.1	71.9	47.6 %
648	18.8	32.0	290.9	405.1	45.8 %
1536	25.5	131.1	1046.6	1203.2	36.9 %

現行実装では，軌道関数 $\left\{\left\{\phi_j\left(\vec{r}_i\right)\right\}_{j=1}^{L}\right\}_{i=1}^{N}$ が GPU から CPU へ転送される（→ Fig. 5.2）. この転送に 1536 電子系で 1203 μsec を要し，かつ対象系の電子数サイズが大きくなると，より転送時間が嵩む．この部分で CPU が返値待ちをしていると効率を低下させてしまう.

更なる高速化のために転送の内容を $\left\{\left\{\phi_j\left(\vec{r}_i\right)\right\}_{j=1}^{L}\right\}_{i=1}^{N}$ の情報容量以下に絞ることは出来ないであろうか？本研究での換装に範囲を限定した場合には，最終的な目標は各電子の試行更新 $\{\vec{r}_i \to \vec{r}_i'\}_{i=1}^{N}$ の棄却/採択の裁定だけである．この場合，N 個の論理型配列で済むから（→ Fig. 5.3）転送量は大幅に（$1/L$ よりも小さく）削減されるだろう.

しかしながら，前節に述べた拡張を見越して，このような実装は採用しなかった．(2; trial) の部分だけであれば，棄却採択の情報のみを返値する機能に持ち込めるが，当該カー

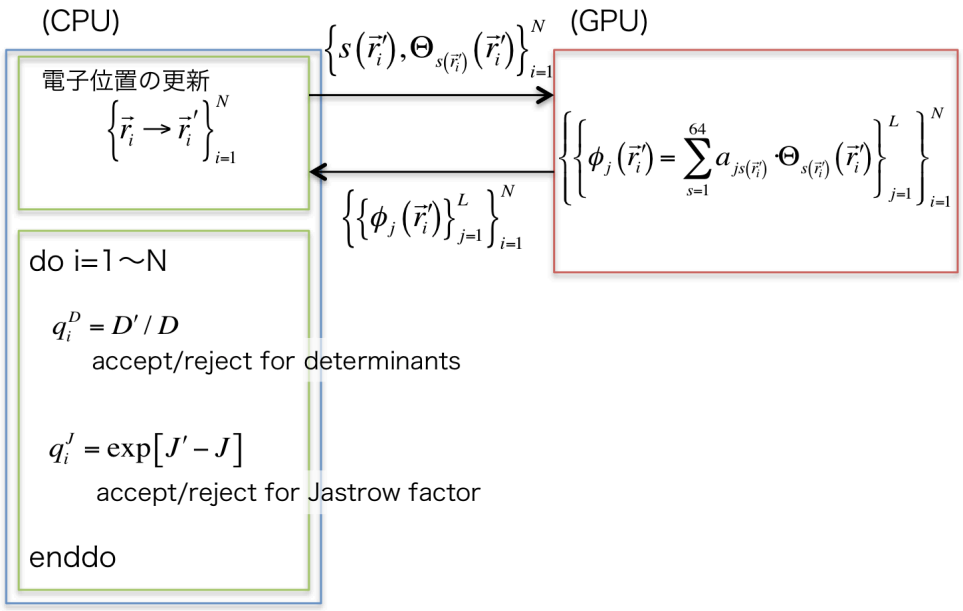

Fig. 5.2: CPU-GPU 間のデータ転送.

ネルを (3; eval) にも適用できるよう拡張する場合，CPU 側では，サンプル値 $E_L\left(\{\vec{r}_i'\}\right)$ の評価に，結局，軌道の関数値や微分値 $\left\{\partial^{(n)}\phi_j\left(\vec{r}_i'\right)\right\}$ $(n = 0, 1, 2)$ を必要とし，転送情報の縮約は不可能である．現在 CPU 側で処理している内容を更に GPU 側で処理させるようにすれば，これら中間量の転送も不要なのではないかという指摘をうけるかもしれない．棄却採択で確定した新配位に対してのエネルギー評価部分は，しかしながら，様々な分岐を含む複雑なルーチンの組み合わせである．GPGPU の部分換装の利点や趣旨からも，この部分は，CPU 側で処理させるという前提がある．

転送に関するオーバーヘッド対策としては，CPU 処理による隠蔽という方策がより現実的である．GPU からの返値待ち時間に CPU である程度重い処理をさせることができれば，待ち時間の隠蔽（オーバーラップ）が出来る．最近の CUDA では,「CUDA stream」の機能により，GPU へのデータ転送中にも CPU に処理が戻り，CPU で演算が可能になり，隠蔽効果を高めるような方策が取れる仕組みになっている．待ち時間中での CPU での処理内容としては，ジャストロー部分に関する棄却採択の評価が妥当な選択である：(2.26) 式に現れている確率比は，(2.23) 式のスレーター・ジャストロー型多体波動関数の比に帰着する．これは「行列式部分 Ψ_D の比による棄却採択決定」と「ジャストロー部分 e^J の比による棄却採択決定」の積となる．汎用の量子モンテカルロ計算コードでは，この部分を Fig.5.2 のように二段階の棄却採択決定（2 レベルサンプリング）に分け効率化が図られている [31]．一旦，試行更新 $\{\vec{r}_i \to \vec{r}_i'\}_{i=1}^{N}$ が振り出されれば，行列式部分の評価とジャストロー部分の評価は独立である．$\{\vec{r}_i \to \vec{r}_i'\}_{i=1}^{N}$ の情報を GPU に転送し処理と返値を待っている間に,「ジャストロー部分の棄却採択決定」を進めてしまう事で転送時間の隠蔽が可

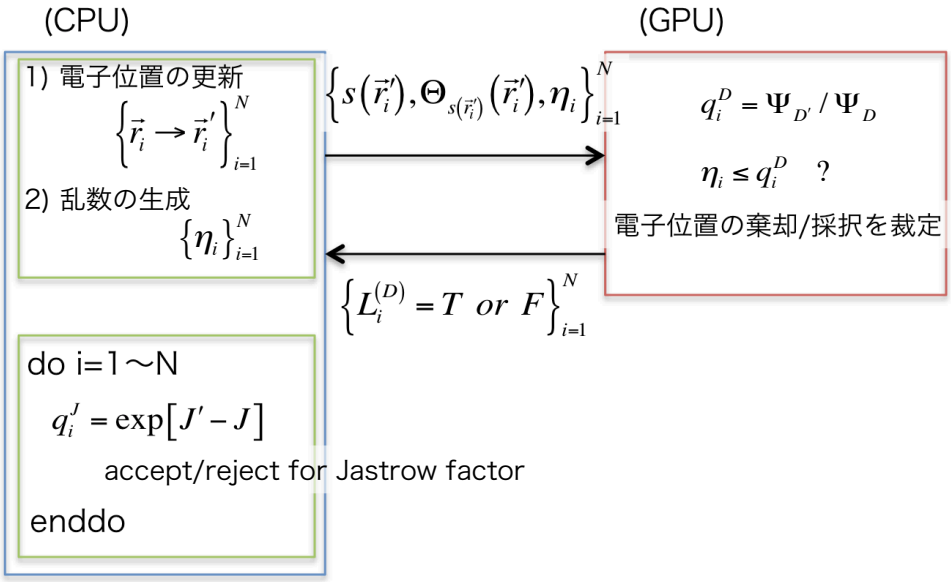

Fig. 5.3: 軽量化したデータ転送の考案

能である．ジャストロー部分の評価ルーチンは，全計算中でも重い律速部分として知られており [32]，大きな隠蔽効果が期待される．

5.2.3 ハイブリッド並列化

現行のコードでは，モンテカルロステップについて MPI 並列化がなされている．例えば，統計蓄積を 10,000 ステップ行うとき，CPU 演算コアが 2 つある場合には，5,000 ステップずつが分担されるため，理想的には 2 倍の高速化がなされる．現行実装での MPI 並列化効率は 99%程度となっている．一つのモンテカルロステップは一つの配位（N 個の電子位置のセット）の更新に相当し，本研究で GPU 換装したのは，この一配位の更新部分である．したがって，本研究のカーネルは，そのまま変更することなく現行の MPI 並列と共存が可能で，ハイブリッド MPI でのリニアな性能の伸びが期待される．

GPGPU を用いたハイブリッド MPI の際，問題になるのは CPU 内のマルチコア資源の利用である．GPU はマザーボードに 1 枚，最大でも 4 台までしか搭載できない．一方，マザーボード内（ノード内）の CPU には 4-8 個程度の演算コアが存在する．本研究では，1 マザーボードに 1CPU+1GPU 搭載，1CPU が 4 コアの構成を用いているが，実装と性能評価は 1CPU コアに対して 1GPU デバイスという構成で行った．1CPU コアで r ステップのモンテカルロ蓄積計算に要する計算時間を，

$$T^{1CPU} = r \times t^{CPU} \tag{5.4}$$

とする．t^{CPU} は 1 モンテカルロステップに要する時間である．1 モンテカルロステップ分の処理が，GPU 利用により β 倍高速化される：

$$t^{CPU} = \beta \times t^{(GPU/1CPUcore)} \ . \tag{5.5}$$

本研究では β =30.67 倍が達成された．CPU 内の演算コア数が n 個の場合，GPU を使わず n コア MPI を行うと，各コアが担当するステップ数は n 分割され，並列化効率が，ほぼ 100%であるから，演算時間は

$$T_{nMPI} = \left(\frac{r}{n}\right) \times t^{CPU} = \frac{T^{1CPU}}{n} \tag{5.6}$$

と書ける．一方，n コア CPU でもシングルコアのみの使用として，1CPU コア 1GPU という利用の場合，

$$T_{GPU/1CPUcore} = r \times t^{(GPU/1CPUcore)} = r \times \left(\frac{t^{CPU}}{\beta}\right) = \frac{T^{1CPU}}{\beta} \tag{5.7}$$

となる．両者の比

$$\frac{T_{nMPI}}{T_{GPU/1CPUcore}} = \frac{\beta}{n} \tag{5.8}$$

が実質的な性能向上比と据えられるべき量で，これは「n コア CPU の演算コアを遊ばせて，シングルコア利用すると，カーネルの β 倍加速が実質 $1/n$ に目減りする」事に相当する．

n コアを用いて 1GPU を共有利用すれば，β 倍までの演算加速は得られないだろうが，「$1/n$ の目減り」よりはましな性能向上が得られるであろう．これを簡単なモデルで見積もってみる：n コアが 1GPU を共有利用する場合のステップあたりの演算時間を $t_0^{(GPU/nCPUcore)}$ とする．これは 1 コアが 1GPU を使う場合の演算時間 $t_0^{(GPU/1CPUcore)}$ とは異なり，資源が共有される分，性能は目減りして，$t_0^{(GPU/nCPUcore)} > t_0^{(GPU/1CPUcore)}$ となるだろう．これは，以下のように見積もることが出来る．n 個のコアは，同一の内容を処理し，ほぼ同一のタイミングで GPU カーネルを呼び出す．この際，最初に GPU を呼び出したコアは全くの空き状態で GPU を利用するから，'GPU/1CPUcore' 利用と同一の $t_0^{(GPU/1CPUcore)}$ で処理を受ける．次に呼び出したコアは，最初のコアの処理が終わる $t_0^{(GPU/1CPUcore)}$ を待ってから，同じく $t_0^{(GPU/1CPUcore)}$ で処理を受けるから，合計で $2 \times t_0^{(GPU/1CPUcore)}$ の処理時間が見積もられる．その次のコアは，前 2 つ分の作業時間 $2 \times t_0^{(GPU/1CPUcore)}$ を待って後に，$t_0^{(GPU/1CPUcore)}$ で処理を受けるから合計は $3 \times t_0^{(GPU/1CPUcore)}$ である．このように考えると，コアあたりの平均の処理待ち時間は，

$$t_0^{nGPU} = \frac{t_0^{GPU} + 2t_0^{GPU} + 3 \cdot t_0^{GPU} + \cdots + n \cdot t_0^{GPU}}{n} = \frac{n+1}{2} t_0^{GPU} \tag{5.9}$$

と見積もることができる．但し，以下のような理想的状況を想定している：

1. データ転送などの遅延がなく，あるコアの処理の後，すぐに別のコアの処理が開始されると仮定．本研究での場合，最も大容量のデータ転送 $cavc\,(j,x,y,z,spin)$ は，計算の冒頭でマスタの演算コアのみによって唯一度だけ行われる．このため，現在考察しているモンテカルロステップ内の処理には関係しない．また，モンテカルロステップからの処理で生じるデータ転送遅延は，既に因子 β の算出に繰り込まれているので，これによるデータ遅延は勘案しなくてもよい．

2. GPU 演算資源が，1 コアからの呼び出しで 100%使われていると仮定．100%に満たない場合には，空いた資源を用いて，2 番目の呼び出しコアの処理も，その一部がすぐに開始するから，ここでの見積もりよりは，性能が向上すると期待される．

n コア MPI 演算で，コアあたりの処理ステップ数は $1/n$ に減じるから，この場合の演算時間は，したがって，

$$T_{nMPI\&1GPU} = \left(\frac{r}{n}\right) \times t_0^{GPU/nCPUcore} = \left(\frac{r}{n}\right) \times \frac{n+1}{2} t_0^{GPU} = \frac{n+1}{2n} \times \frac{T^{1CPU}}{\beta} \tag{5.10}$$

と見積もられる．n コア MPI 演算で GPU を利用する場合としない場合のメリットファクターは，したがって，

$$\frac{T_{nMPI}}{T_{nMPI\&1GPU}} = \frac{T^{1CPU}}{n} \frac{2n\beta}{(n+1)\,T^{1CPU}} = \frac{\beta}{(n+1)/2} \tag{5.11}$$

となる．

第6章　結論

本研究ではスプライン基底関数型を用いた量子モンテカルロ計算の律速箇所を GPGPU に一部換装した．その結果，1536 電子の固体系の計算で最終的に 30.67 倍の律速箇所の高速化を達成した．初期の逐次更新の実装では，1.5 倍程度の高速化しか見られなかったが，斉次更新アルゴリズムを適用し，GPU 演算部の並列度を増したことで，最終的な性能加速比を得られた．また GPU 換装部分の演算は単精度で行なったが，最終的に算出される統計エネルギー値にはその影響がなく，計算上，必要とされる精度を十分に達成していることが分かった．

今回の実装では，電子位置の棄却/採択ルーチン部（2; trial）にのみ GPGPU を適用したことで，計算全体では 1.06 倍程度の高速化であったが，さらに計算時間割合の大きい，エネルギー評価部（3; eval）への適用はストレートフォワードである．また，問題としていた blip3d ルーチンの律速を GPGPU を適用することで解消できることが本研究の成果から分かり，さらに次の律速部分が現れる状態になった．次の律速部分に対しても GPU が適用できる部分であることから，さらに計算全体の高速化を図ることができると考えられる．

謝辞

本研究を行うにあたり，終始，熱心なご指導を賜った前園涼准教授に深謝いたします．また的確なご指導をいただきました，松澤照男教授，敷田幹文准教授に深く感謝いたします．日頃から有益な議論や助言をいただき，本研究の支えとなっていただいた前園研究室，松澤研究室の皆様に厚く御礼申し上げます．

最後に，多くの方々のご支援，ご指導により充実した研究生活を送れたことに感謝いたします．

参考文献

[1] J. Huang, Supercomputing Conference (SC11)

[2] David B. Kirk, Wen-mei W. Hwu, CUDA プログラミング実践講座, ボーンデジタル, 2010.

[3] CUDA Community Showcase, http://www.nvidia.com/object/cuda-apps-flash-new.html

[4] A. Oshiyama, J. Iwata, J. Phys.: Conf. Ser. **302**, 012030, 2011.

[5] H. F. Schaefer, Electronic Structure of Atoms and Molecules, Addison-Wesley, 1972

[6] Y. Uejima, T. Terashima, R. Maezono, J. Comput. Chem. **32**, 2264-2272, 2011.

[7] D.A. マッカーリ, J.D. サイモン 著, 千原秀昭, 斎藤一弥, 江口太郎 訳, 物理化学〈上〉, 東京化学同人, 1999

[8] 前園涼, 固体物理, **39**, 779-790, 2004.

[9] 小川哲生, 量子力学講義, サイエンス社, 2006

[10] 里子允敏, 大西楢平, 密度汎関数法とその応用, 講談社, 1994

[11] 今田正俊, 常行真司, 寺倉清之, 日本物理学会誌, **64**, 241, 2009.

[12] M. Kamiya, T. Tsuneda, K. Hirao, J. Chem. Phys. **117**, 6010, 2002.

[13] 藤永茂, 入門分子軌道法, 講談社, 1990

[14] 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

[15] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, J. Chem. Phys. **21**, 1087, 1953.

[16] Y. Ge, T. C. Shore, Comp. Theor. Chem. **978**, 57-66, 2011.

[17] 前園涼, ナノ学会会報, **3**, 87-95, 2005.

[18] M. J. Gillan, M. D. Towler, D. Alfè , Psi-k Highlight of the Month, 2011.

[19] D. Alfè, M. J. Gillan, Phys. Rev. B. **70**, 161101(R), 2004.

[20] R maezono, J. Comput. Theor. Nanosci. **6**, 2474-2482, 2009.

[21] CASINO,
http://www.tcm.phy.cam.ac.uk/˜mdt26/casino2.html, accessed on 1 Feb. 2012

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

[23] CHAMP,
http://pages.physics.cornell.edu/˜cyrus/champ.html, accessed on 1 Feb. 2012

[24] QMCPACK,
http://qmcpack.cmscc.org, accessed on 1 Feb. 2012

[25] QWALK,
http://www.qwalk.org/wiki/index.php?title=Main_Page, accessed on 1 Feb. 2012

[26] NVIDIA Fermi のマルチスレッディングアーキテクチャ,
http://pc.watch.impress.co.jp/docs/column/kaigai/20091105_326442.html, accessed on 1 Feb. 2012

[27] ソフテック, マルチコア CPU 上の並列化手法、その並列性能と問題点,
http://www.softek.co.jp/SPG/Pgi/TIPS/public/general/multicore-para.html, accessed on 1 Feb. 2012

[28] Tesla C2050 Performance Benchmarks,
http://www.siliconmechanics.com/files/C2050Benchmarks.pdf, accessed on 1 Feb. 2012

[29] K. Hongo, R. Maezono, K. Miura, J. Comput. Chem. **31** 2186-2194, 2010.

[30] Intel VTune Amplifier XE,
http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe, accessed on 1 Feb. 2012

[31] M. Dewing, J. Chem. Phys. **113**, 5123, 2000.

[32] CASINO manual,
http://www.tcm.phy.cam.ac.uk/˜mdt26/casino_manual_dir/casino_manual.pdf, accessed on 1 Feb. 2012