Vol.2013-HPC-138 No.21
2013/2/22

GPUクラスタにおける核融合シミュレーションコードの実装

藤田 典久[1,a)]　奴賀 秀男[2]　朴 泰祐[1,2]　井戸村 泰宏[3]

概要：我々は大規模トカマクプラズマ中の乱流現象をシミュレーションする，核融合シミュレーションコード GT5D の高速化を進めるべく，大規模 GPU クラスタ向けのコード開発を行っている．本稿では，各計算ノードに複数の GPU を搭載する大規模マルチ GPU クラスタ HA-PACS における GT5D コードの高速化について述べる．これまでの開発では同コードの時間発展部分の中で計算時間の重い部分を中心に GPU 化を進めて来たが，本稿では特にステンシル計算部分における通信レイテンシ隠蔽について述べる．通信と計算のオーバラップの実装により，関数 1 回あたり 21ms の高速化が得られ，時間発展全体の性能評価では，同じ条件で CPU のみの場合の 1.85 倍の高速化が達成できた．また，通信と計算のオーバラップの実装により，時間発展 1 回あたりさらに 1.44 倍の高速化が得られた．

1. はじめに

GPU (Graphics Processing Unit) は従来 3D グラフィックスを描画するためにしか利用されていなかったが，GPU に汎用的な計算をさせる General Purpose computing on GPU (GPGPU) が高性能計算分野で脚光を浴びている．GPU は CPU と比較して高い並列演算性能とメモリバンド幅を持ち，NVIDIA 社の Tesla M2090 では，倍精度演算性能で 665 GFLOPS，メモリバンド幅で 177GB/sec に達する．

近年の GPGPU の普及と，1 台のマシンが接続できる PCI Express のレーン数の増加に伴い，1 台のマシンに 3 台や 4 台の GPU を搭載するシステムも登場しているため，効率的なメモリ転送の戦略や，GPU の制御方法が重要視されている [1], [2]．また，計算を全て GPU に任せ，CPU は GPU 制御やノード間通信のみを行う計算モデルだけでなく，GPU が計算を行ないつつ CPU も計算を行う協調計算型のモデルも用いられている．

世界のスーパーコンピュータのランキングである Top500 リスト [3] の 2012 年 11 月付けのランキングによると，上位 20 個のシステムの中では，1 位 Titan, 8 位 Tianhe-1A, 12 位 Nebulae, 17 位 TSUBAME 2.0 の 5 システムが GPU によるアクセラレータを搭載しており，1 位の Titan は Linpack 性能で 17.5PFLOPS を記録している．

一方，核融合実験炉を対象とした核融合シミュレーションは計算工学の中で重要アプリケーションである．この中で，トカマクプラズマ中の乱流現象をシミュレーションするためには，核融合プラズマ中の乱流現象のシミュレーションでは第一原理的な計算が行われるため，高い計算性能が必要とされる．また乱流のシミュレーションには高い空間解像度が要求されるため，今後の装置規模の大型化に応じて計算量の著しい増大が見込まれる．現在，日本学術振興会が全世界の他機関と共同で推進する，G8 多国間国際研究協力事業研究課題「エクサスケール規模の核融合シミュレーション」では，同種のシミュレーションコードの大規模化・高速化を進めており，本研究もその一貫として行われている．

本研究の目的は大規模 GPU クラスタで高速に実行できる核融合シミュレーション用プログラムを開発することである．一般的に，GPU を用いて計算を行うプログラムは CPU のプログラムをそのまま流用することはできず，計算部分を中心に GPU 用に専用のプログラムを開発しなければならない．また，1 台のマシンに複数の GPU を搭載されているような環境を想定し，最適化を施していく．本研究では，GPU 向けの核融合シミュレーションコードを新たに開発するのではなく，次世代実験炉を対象として日本原子力機構で開発が進んでいるシミュレーションプログラムコードである GT5D を対象とし，これを GPU 向けに

[1] 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[2] 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[3] 日本原子力機構
Japan Atomic Energy Agency
[a)] fujita@hpcs.cs.tsukuba.ac.jp

改変することで大規模 GPU クラスタでの実行と速度向上を目指す．

2. GT5D

核融合シミュレーション用プログラム GT5D (conservative global gyrokinetic toroidal full-f five-dimensional Vlasov simulation) [4] は，旋回平均された速度分布関数の時間発展を計算するコードであり，トカマクプラズマ中の乱流現象を記述するものである．プラズマ中の乱流現象は，プラズマ輸送などのより大きな時間・空間スケールの現象にも影響を及ぼし，例えば，異常輸送や，乱流駆動不安定性などの原因となる．

GT5D の扱う空間を**図 1** と**図 2** で示す．GT5D はトーラス配位の実空間 3 次元 (R, Z, ζ)(図 1) と，粒子の速度空間 2 次元 $(v_\parallel, v_\perp)$ を位相空間変数としている．ここで，$v_\parallel, v_\perp$ はそれぞれ磁力線に平行方向の速度，垂直方向の速度である．荷電粒子は磁力線に巻き付くように運動するが，磁力線を旋回する速度は GT5D が対象とする乱流現象に比べて十分速い．このため，旋回平均によって速度空間変数から旋回位相を消去できる．

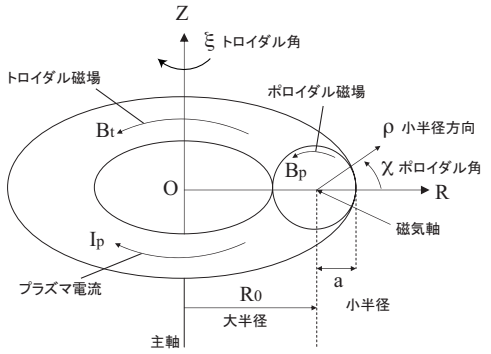

図 1 一般的なトーラス配位．このうち GT5D で座標変数として使われているものは (R, Z, ζ)．ただし，$\zeta = -\xi$ である

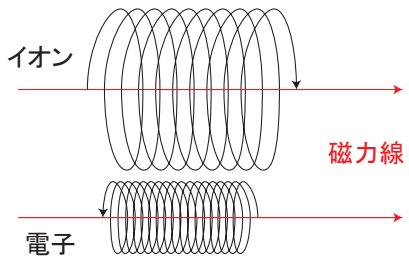

図 2 磁力線に沿って運動する荷電粒子の図．荷電粒子はローレンツ力によって磁力線の周りを旋回移動するため，粒子の速度を磁力線に平行方向の速度 $v_\parallel$，磁力線の垂直方向の速度 $v_\perp$, 旋回位相 ϕ の 3 変数で表わすことができる．このうち旋回位相は旋回平均によって消去でき，計算量を削減できる

GT5D は「京」及び大規模高性能クラスタに移植され，国内最大規模の並列シミュレーションが実現されている．GT5D の計算量は，シミュレーションの対象とする装置の規模に依存する．小規模な装置のシミュレーションは計算量が少くて済むが，ITER[5] や DEMO[6] といった次世代の実験炉の乱流現象を計算するためには，現在のスーパーコンピュータでは計算能力が不足しているため [7]，より高速な計算機が求められている．

3. NVIDIA GPU の開発環境

3.1 CUDA 開発環境

CUDA[8] は NVIDIA 社の GPU で汎用計算を行うための開発環境である．CUDA Toolkit には，C/C++コンパイラ，ドライバ，ランタイムライブラリ，プロファイラ，CUDA 用 BLAS (Basic Linear Algebra Subprograms) ライブラリである CUBLAS などが含まれる．

3.2 PGI CUDA Fortran

GT5D は Fortran で記述されているが，NVIDIA 社の提供する GPGPU 用開発環境 CUDA では，C 言語および C++言語のコンパイラのみ提供されており，そのままでは GT5D のソースコードを再利用できない．そのため，本研究では PGI 社の提供する PGI CUDA Fortran コンパイラ [11] を利用する．

PGI CUDA Fortran は，CUDA C/C++のように，Fortran の仕様に CUDA のために文法を拡張したコンパイラと，CUDA ランタイムライブラリを Fortran から呼び出すためのライブラリから構成される．PGI CUDA Fortran コンパイラは，Fortran コードを C コードに変換し，バックエンドとして CUDA C/C++コンパイラ (nvcc) を呼び出し，GPU 向け実行ファイルを作成する．PGI CUDA Fortran のソースコード例を**図 3** に示す．CUDA C/C++における `__global__` と同等の意味を持つ `attributes(global)` や，Shared Memory に領域を確保することを示す `shared` 属性，カーネル起動時のスレッド，ブロックの次元数を指定する “<<< >>>” といったものが，Fortran に対する CUDA 拡張である．また，CUDA ランタイムの関数は，ほぼ全て Fortran から呼べるようにバインディングが提供されている．

4. GT5D の GPU 化

本章では GT5D をどのようにして GPU 化するのかについて述べる．GT5D のおおまかな計算内容は，初期化部，時間発展部，後処理部から成る．初期化部では，初期値の計算やリスタートの処理などを行い，時間発展部でシミュレーションを行い，そして，後処理部で各種リソースの解放などを行う．初期化部は時間発展部の反復回数に依らず，一定の時間がかかるが，時間発展部は時間発展の反復回数に比例して計算時間が延びる．したがって，本研究では GT5D の時間発展部分を GPU 化の対象とする．

```
attributes(global) &
subroutine saxpy_kernel(alpha, x, y)
  real, value :: alpha
  real :: x(256), y(256)
  real, shared :: tmp(256)

  tmp(threadIdx%x) = y(threadIdx%x)
  y(threadIdx%x) = &
    alpha * x(threadIdx%x) + tmp(threadIdx%x)
end subroutine saxpy

subroutine saxpy(alpha, x, y)
  real :: alpha
  real, device :: x(256), y(256)

  call saxpy_kernel<<<1, 256>>>(alpha, x, y)
end subroutine saxpy
```

図 3 PGI CUDA Fortran の例

4.1 並列化の方針

GT5D の GPU 化にあたり，1 つの MPI プロセスがいくつの GPU を制御するかを考える．本研究では，**図 5** に示す割り当て方針を採用する．1 つの MPI プロセスに対して 1 つの GPU を割り当て，プロセスは担当している GPU のみを制御する．また，スレッド数もコアを全て使い切れるように設定する．例えば，8 つのコアを持つ CPU に対して 2 つの GPU が接続されている環境では，OpenMP のスレッド数を 1 プロセスあたり 4 に制限する．スレッド数の設定は `OMP_NUM_THREADS` 環境変数を通じて行う．

HA-PACS は NUMA 構成となっているため，他の CPU にあるメモリへのアクセスは速度面でペナルティがある．また，GPU も同様に，他の CPU の配下にある GPU へのデータ転送は，避けなければならない．前述した 2 つの条件を満たすために，`numactl` コマンドを使用し，あるプロセスが実行される CPU を固定する．`numactl` は NUMA 環境でのリソースを制御するために用いるコマンドであり，プロセスが利用する CPU コアとメモリを限定できる．例えば，**図 4** の様にコマンドを実行すると，GT5D をノード 0 番の CPU で実行し，0 番 CPU に接続されているメモリ (ローカルメモリ) を利用するという意味になる．

```
$ numactl --cpunodebind=0 --localalloc -- ./GT5D
```

図 4 numactl コマンドの例

本方針では，GT5D が持つ既存の OpenMP 並列および MPI 並列のプログラムを再利用でき，開発が容易であること，また，プロセス毎にデータ参照の局所性があり，NUMA によるメモリアクセスのペナルティを受けにくいこと，あるプロセスが操作する GPU が，`numactl` コマンドによって設定された CPU と直接接続されていることを保証できること，といった利点があるが，一方で，同じノードに接続されている GPU 間のデータ交換でさえ，MPI を経由せねばならず，オーバーヘッドが発生するという欠点を持つ．

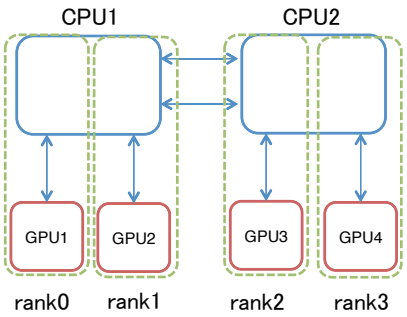

図 5 MPI プロセス毎の CPU コアと GPU の割り当ての方法

4.2 GT5D の時間発展部の流れ

GPU 化の方針を立てるため，まずオリジナルの GT5D コードを CPU のみを用いて実行時間を測定した [9] 時間発展 1 回の時間と呼び出し回数測定結果を**表 1** に示す．時間発展中で，最も時間のかかる関数は `l4dx_s` であり，以降，`lfp`，その他と続くことがわかる．

時間発展部の処理の流れの概要を**図 6** に示す．時間発展の中には，内部ループ (図 6 の波線部) が 2 つあり，収束判定が満たされるまで繰り返される．例えば `l4dx_s` 関数は内部ループ内で呼ばれているため，実行パラメータによって呼び出し回数が変化する．また，`l4dx_s`, `l4dx_r`, `l4dx_l`, `l4dx_nl` の 4 つの関数は，計算のみを含んでおり，MPI 通信を行わない関数であるが，`lfp` 関数は MPI 通信を含んでいる．したがって，`l4dx_s`, `l4dx_r`, `l4dx_l`, `l4dx_nl` 関数の方が GPU のみで計算が完結し，GPU 化が行いやすい．また，時間発展中に，関数として分離されていない，小さな DO ループがいくつかあり，それらのループが表 1 のその他の部分に該当する．

図 6 の波線部からわかるように，内部ループに `bcdf` という関数が含まれている．表 1 からわかるように，`bcdf` 関数は内部ループに含まれているため，呼び出される回数が多い．`bcdf` 関数は袖領域の交換のための関数であり，MPI 通信を含んでいる．MPI 通信に用いるデータは CPU のメモリに存在しなければならない．したがって，関数を GPU 化することはできず，必ず CPU で実行しなければならないため，前後の CPU~GPU 間の通信を回避できない．

GT5D の MPI 並列の分割数は n_R, n_Z, n_μ の 3 変数で表わされる．このうち，n_R と n_Z はそれぞれ図 1 における R 方向と Z 方向への分割数であり，n_μ は $v_\perp$ の分割数である．`bcdf` 関数は，R 方向と Z 方向の袖領域を交換する関数であり，$n_R = 1$ の場合は R 方向への通信は行なわれず，$n_Z = 1$ の場合は Z 方向への通信を行わない．

表 1　GT5D の時間計測結果

関数名	時間 [ms]	割合 [%]	回数
その他	2703.237	39.22	
l4dx_s	1934.259	28.06	30
lfp	1283.132	18.62	2
l4dx_nl	288.847	4.19	2
bcdf	225.992	3.28	32
l4dx_l	167.247	2.43	2
fld_sfls	124.050	1.80	2
l4dx_r	113.089	1.64	2
drift_nl	38.424	0.56	2
dn3d	14.260	0.21	2
bcv	0.430	0.01	2
合計	6892.967		

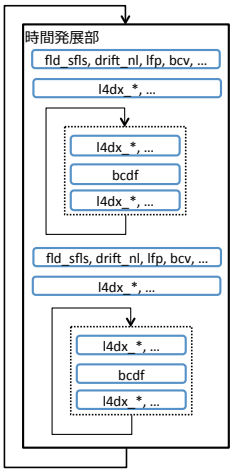

図 6　GT5D の時間発展部の概要図．ただし，波線部は内部ループを表わす

5. 通信と計算のオーバーラップ

これまでの研究 [9] では，時間発展部分の中で表 1 の項目の内，l4dx_s, l4dx_r, l4dx_l, l4dx_nl, lfp, bcv およびその他の範囲について GPU 化と性能評価を行なったが，MPI 通信および CPU~GPU 間の通信については最適化を行なっていなかった．

MPI 通信と計算をオーバーラップさせて，通信の時間を隠蔽する最適化手法は CPU で実行するプログラムの場合でも一般的に行なわれている [10]．GPU 上にあるデータを直接 MPI で通信できないため，MPI 通信を行う際は MPI 通信の前に CPU~GPU 間通信も行いデータを CPU 側に転送する必要があり，CPU のみで計算を行う場合よりも通信オーバーヘッドが大きくなる．したがって，通信と計算のオーバーラップを行なう際の効果も大きい．本研究では，bcdf 関数内で行なわれている通信に着目し，通信と計算のオーバーラップを行なう．

5.1 bcdf 関数における通信と計算のオーバーラップ

bcdf 関数は領域のデータの交換を行うための関数である．bcdf 関数が扱うデータは $(R, Z, \zeta, v_{\parallel})$ の 4 つの次元で構成されている．4 つの次元それぞれに袖領域処理が必要であるが，MPI プロセスを跨いでの分割の有無で，MPI による分割がある (R, Z) の 2 次元と，MPI による分割がない $(\zeta, v_{\parallel})$ の 2 次元に分けて考える．そして，(R, Z) はそれぞれ n_R, n_Z 個のプロセスに分割されている．各次元の袖交換作業はそれぞれ独立しているため，MPI による通信を行なっている最中に，プロセス内で閉じている作業を並行して行えるため，通信時間を隠蔽できる．

袖領域の処理だけでなく，前後で行う計算部についても，通信に影響を受ける部分の計算と受けない部分の計算に分離すれば，通信と計算のオーバーラップが行える．計算を (R, Z) 次元の袖領域の値を必要とする領域 (境界) と，(R, Z) 次元の袖領域の値を必要としない領域 (内点) に分ける．内点の部分の計算は MPI 通信を必要とする袖領域のデータを使わずとも計算できるため，MPI 通信と計算のオーバーラップが行える．

袖領域の処理と前後の計算の 2 つのオーバーラップをまとめた図が**図 7** である．2 の計算カーネル k_1, k_2 の間で bcdf 関数で袖領域を交換している場合を表している．GPU カーネルの起動，CPU~GPU 間通信，MPI 通信をそれぞれ非同期に実行する．GPU の非同期操作は CUDA Stream を用いて行い，MPI の非同期通信には MPI_Isend と MPI_Irecv を用いる．ただし，同じ CUDA Stream に関連付けられた命令は直列にしか実行されないため，通信用と計算用の 2 つの CUDA Stream を用意する．これにより，それぞれの CUDA Stream に関連付けられた命令は並列して実行されるため，通信と計算を同時に行える．

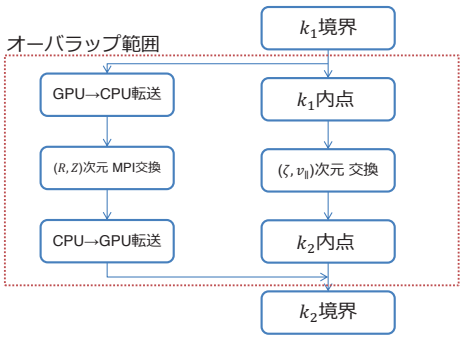

図 7　通信と計算のオーバーラップの概念図

6. 性能評価

6.1 計算機環境

本研究では，筑波大学計算科学研究センターの超並列 GPU クラスタである HA-PACS を実験に用いる [1]．HA-PACS 1 ノードの性能諸元を**表 2** に示す．1 つのノードに，Intel Xeon E5-2670 が 2 台，NVIDIA Tesla M2090 が 4 台，

および dual rail の Inifiband HBA が搭載され，**図 8** のように接続されている．CPU1 と CPU2 の間は，Intel の CPU 相互接続用シリアルバスである QuickPass Interconnect (QPI) で接続され，CPU と各 GPU 間は PCI Express 16 レーンで接続され，CPU1 つにつき GPU が 2 つ接続されている．CPU1 と CPU2 はそれぞれ 64GB のメモリが結合され，ノードあたり 128GB のメモリを持つ NUMA (Non Uniform Memory Access) を構成している．したがって，CPU1 から CPU2 のメモリ，CPU2 から CPU1 のメモリへのアクセスは，自 CPU の持つメモリより若干時間がかかる．ノード間インターコネクトとしては，Infiniband QDR ×2 レールを用いるマルチレール環境を構成している．ノード全体の接続はファットツリー型となっており，総ノード数は 268 台である．CPU と GPU 間の接続関係は，Linux の sysfs を通じて提供されている情報を用いて取得でき，CPU0 番ノードに，デバイス番号 0 と 1 の GPU が接続され，CPU1 番ノードに，デバイス番号 2 と 3 の GPU が接続されている．ただし，GPU デバイス番号は `cudaSetDevice` 関数で GPU を指定する際に用いる数字のことを指す．

本研究では CPU および GPU とプロセスの割り当てを図 5 の様に行うため，CPU で行う処理や CPU~GPU 間のデータ転送を行う際に用いるメモリは，それぞれのソケットが持つメモリを使用する．したがって，QPI を経由するデータ転送の発生を抑えられ，全体の性能向上に繋る．

表 2 計算機環境

CPU	Intel Xeon E5-2670 × 2 (2.6GHz)
	CPU Core 数 8 core/CPU × 2 = 16 core
CPU メモリ	128GB, DDR3 1600MHz
GPU	NVIDIA Tesla M2090 × 4
GPU メモリ	6GB/GPU
OS	CentOS 6.1
CUDA Toolkit	ver. 4.1
PGI Compiler	ver. 12.10
PGI Compiler Options	`-fastsse -Mcuda=4.1,flushz`
	`-Mipa=fast,inline`
MPI	MVAPICH2 1.8
相互結合網	Infiniband QDR 4 レーン， 2 レール

6.2 通信を含まない関数の性能評価

まず，これまでの研究 [9] で実装した通信を含まない計算のみの関数の性能評価を行う．本評価では，各種パラメータを次のように設定し測定を行う．測定対象の CPU 版 GPU 版それぞれの関数を呼び出し計算結果が一致しているかどうかと，処理時間を計測するテストプログラムを作成し，本測定で使用する．ただし，測定用のテストプログラムは MPI 並列を使用せず 1 ノードのみで動作する．実行時のメッ

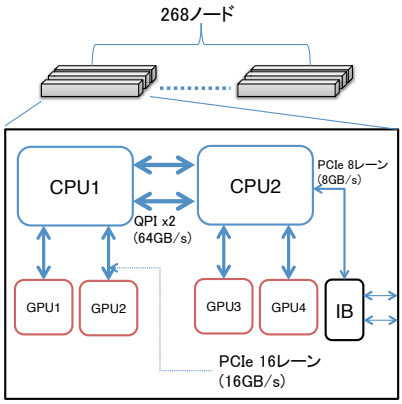

図 8 ノード内のコンポーネント間接続の概念図

シュ分割数は $(N_R, N_\zeta, N_Z, N_{v_\parallel}, N_\mu) = (64, 64, 64, 64, 1)$，CPU 側の OpenMP スレッド数は 4，GPU 使用数は 1 とする．

`timedev1`~`timedev9` 関数を GPU 化し，CPU(4 コア) と性能を比較した結果を**表 3** と**図 9** に示す．最も性能が改善した関数は `timedev1` のケースで，CPU と比べ 3.37 倍高速になった．また，`timedev1`~`timedev9` 関数の平均では，CPU と比べ 2.66 倍高速になった．

`l4dx_r`, `l4dx_s`, `l4dx_l`, `l4dx_nl` 関数を GPU 化し，CPU(4 コア) と性能を比較した結果を**表 4** と**図 10** に示す．最も性能が改善した関数は `l4dx_r` 関数のケースで，CPU と比べ 2.16 倍高速になった．`l4dx_s`, `l4dx_nl` 関数でも速度向上がみられるものの，`l4dx_l` 関数は CPU と比べて 0.77 倍と，GPU で実行する方が遅くなってしまった．

表 3 `timedev1`~`timedev9` 関数の性能評価

関数名	CPU(4 コア)[ms]	GPU[ms]	Speedup
`timedev1`	17.2	5.1	3.37
`timedev2`	18.2	8.3	2.19
`timedev3`	22.1	9.5	2.33
`timedev4`	22.4	10.5	2.13
`timedev5`	22.2	10.5	2.11
`timedev6`	21.8	6.7	3.25
`timedev7`	16.9	5.1	3.31
`timedev8`	26.4	8.3	3.18
`timedev9`	22.6	10.9	2.07

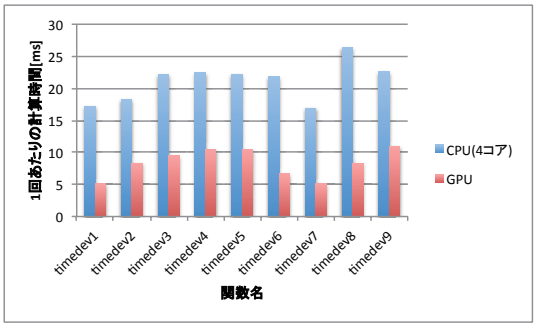

図 9 `timedev1`~`timedev9` 関数の性能評価のグラフ

表 4　l4dx_r, l4dx_s, l4dx_l, l4dx_nl 関数の性能評価

関数名	CPU(4 コア)[ms]	GPU[ms]	Speedup
l4dx_r	38.9	18.0	2.16
l4dx_s	47.0	33.0	1.42
l4dx_l	82.6	106.6	0.77
l4dx_nl	148.0	123.9	1.19

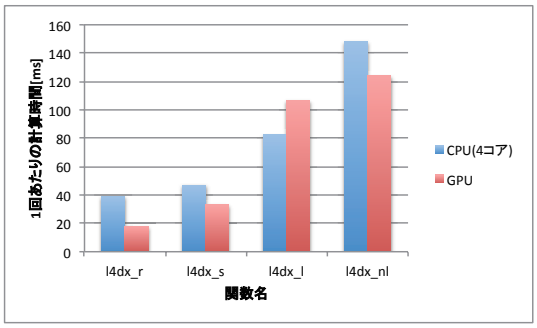

図 10　l4dx_r, l4dx_s, l4dx_l, l4dx_nl 関数の性能評価のグラフ

6.3　bcdf 関数の通信と計算のオーバーラップ評価

bcdf 関数の通信と計算のオーバーラップによる通信時間の隠蔽評価を行う．通信と計算のオーバーラップは前後の計算を行う関数によって以下の 5 パターンが存在する．

(1) timedev_2 → bcdf → l4dx_s
(2) timedev_3 → bcdf → l4dx_s
(3) timedev_4 → bcdf → l4dx_s
(4) timedev_4 → bcdf → timedev_6
(5) timedev_4 → bcdf → timedev_8

以上 5 つのパターンの内，本評価では 3 番のパターンについて評価を行う．3 番のパターンは，図 6 の波線部で示す内部ループで用いられているパターンで，実行回数が他のパターンより多いため評価対象とする．

なお，CPU 側の処理時間の測定は MPI_Wtime 関数を使用し，GPU 側の処理時間の測定は cudaEvent を使用する．オーバーラップを行う際は，CUDA カーネルは非同期に実行されるため，MPI_Wtime などの CPU 側で時間を計測する手段では処理時間を求められない．cudaEvent は GPU の処理の開始や終了を検出するために用いる機構であるが，cudaEvent は 2 つのイベントの間の時間を求める cudaEventElapsedTime 関数があり，それを用いて処理時間を計測する．また，ジッタなどの影響を軽減するために，関数の入口で MPI_Barrier 関数を用いて全ての MPI プロセスの待合せを行う．

関数全体の計算時間を**表 5** に示す．GT5D のパラメータは $(N_R, N_\zeta, N_Z, N_{v_\parallel}, N_\mu) = (128, 128, 64, 128, 1)$，MPI プロセス数は $(n_R, n_Z, n_\mu) = (4, 4, 1)$ で性能評価を行う．オーバーラップなしの場合は 1 回あたり約 66ms かかっていたが，オーバーラップを行うことで 1 回あたり約 45ms と，21ms (x1.46) の短縮の効果が得られた．また，オーバーラップありの場合の詳細な処理時間を**図 11** に示す．Calc, Transfer の 2 つの縦線はオーバーラップに用いる 2 つの CUDA Stream を表し，CPU の縦線は MPI の通信状況を表わす．また，線の上に書かれている箱は各処理の時間を表わす．ただし，図の構成上，箱の高さと実際の処理時間の比率は一定ではない．それぞれの処理の内容は以下の通りである．

timedev4 bonudary　timedev4 カーネルの境界部の計算．

timedev4 inner　timedev4 カーネルの内点部の計算．

bcdf pack　bcdf MPI exchange で MPI 通信を行うために timedev4 boundary で計算した境界部の計算結果を CPU 側に転送する．

bcdf MPI exchange　MPI を用いて隣接プロセスと袖領域のデータを交換する．

bcdf exch. inner　周期境界条件となっている次元の袖領域の交換処理を行う．ノード内でデータ移動が完結するため MPI 通信は発生しない．

bcdf exch. boundary　bcdf MPI exchange で更新されたデータを GPU 側に書き戻す．

l4dx_s boundary　l4dx_s カーネルの境界部の計算．

l4dx_s inner　l4dx_s カーネルの内点部の計算．

図 11 の赤線は l4dx_s inner の処理の終りを示し，緑線は bcdf exch. boundary の処理の終りを示す．計算 (緑) が通信 (赤) よりも早く完了しているため，GPU が約 2.2ms 遊んでいることがわかる．l4dx_s boundary は bcdf exch. boundary の処理が終わらなければ計算できないためである．

表 5　bcdf 関数の計算時間

オーバーラップ	時間 [ms]	Speedup
なし	66.327	-
あり	45.337	1.46

6.4　時間発展全体の性能評価

時間発展全体の性能評価を行う．測定の際のメッシュ数は $(N_R, N_\zeta, N_Z, N_{v_\parallel}, N_\mu) = (128, 128, 128, 128, 4)$，MPI プロセス数は $(n_R, n_Z, n_\mu) = (4, 4, 4)$ を使用する．1 ノードあたり 4 プロセスを起動するため 64 プロセス = 16 ノードとなる．

時間発展 1 回あたりの計算時間を**表 6** に示す．オーバーラップのありなしは，bcdf 関数におけるオーバーラップのありなしを示す．CPU と比較して，GPU を用いる場合，オーバーラップなしの場合で 1.29 倍，オーバーラップありで 1.85 倍の高速化が得られていることがわかる．また，オーバーラップのありとなしで比較すると，オーバーラップありの方が 1.44 倍高速であることがわかる．

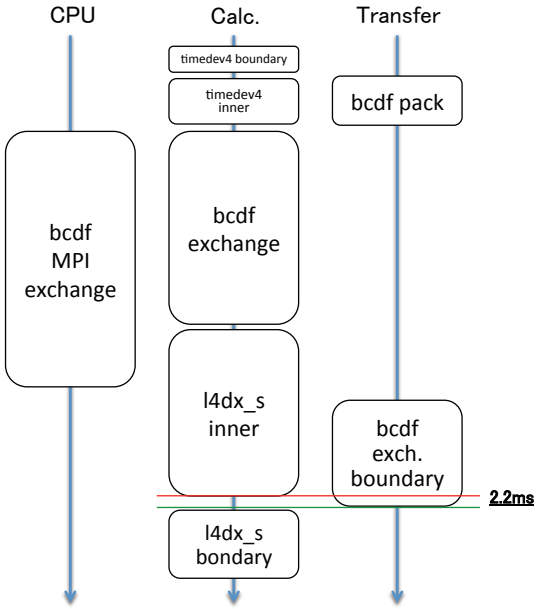

図 11 オーバーラップありの場合の bcdf 関数の処理時間の詳細

表 6 時間発展 1 回あたりの計算時間

	計算時間 [s]	CPU 比
CPU	15.7	
GPU (オーバーラップあり)	12.2	1.29 倍
GPU (オーバーラップなし)	8.5	1.85 倍

7. 考察

本研究ではいくつかの関数の GPU 化を行なったが，一部の関数について高速化が達成できていない．`timedev` 系の関数や `l4dx_r` 関数については GPU の方が 2 倍以上高速であるが，`l4dx_s`, `l4dx_r` 関数については 2 倍未満の高速化しか得られず，`l4dx_nl` 関数については GPU の方が遅いという結果が得られた．それぞれのカーネルについて，実行時の情報を**表 7** に示す．これらの情報は Compute Profiler を用いて取得している．また，パラメータについては性能評価の際に使用したものと同じであり，1 つの関数名の項目に対して複数のカーネル名の項目がある関数は，複数のカーネルから関数が構成されていることを表わす．

GPU の方が遅い関数 `l4dx_l` は Occupancy が低いことがわかる．CUDA のアーキテクチャは，メモリアクセスやその他の要因によってスレッドの実行ができない状態になると，他のワープの実行に切替えて，演算器が遊ばないようする制御が行なわれている．なお，切替え単位がワープなのは，SM の最小制御単位がワープなためである．

Occupancy が低いということは，SM が実行ワープを切替える際の切替え先の候補が少ないということであり，特にメモリアクセスの比率が高いカーネルにおいて，全てのワープが実行不能になり，結果として性能が低下する可能性が高くなる．一方で Occupancy が低いが性能が高いカーネルは，演算比率が高いカーネルであると考えられ，そのようなカーネルでは実行が停止するワープが少なく，SM が保持するワープの数が少なくとも実行効率が高い状態になっていると考えられる．

`l4dx_l` と `l4dx_nl` 関数に含まれる `reduce` カーネルは，計算カーネル本体で求めた結果の総和を取る補助カーネルである．現在の CUDA では，ブロックを跨いでの同期命令が存在しないため，複数のブロックの計算結果の総和を求める場合などには，総和を求めるだけのカーネルを作らなければならない．総和を求めるカーネルが実行される時には，前に実行されている計算カーネルが終了していることが保証されるが，総和計算は並列度が高くないため，1 ブロック ×12 スレッドといった少ないスレッド数のパラメータでカーネルを起動しなければならず，Occupancy が低くなってしまう問題が避けられない．

CUDA では 1 スレッドが使えるレジスタ数の上限は 63 であるが，`l4dx_nl` カーネルはレジスタの使用数が 63 となっており，制限に抵触している．そのようなカーネルでは，プログラムで必要になるレジスタの数が 63 よりも多いため，ローカルメモリにデータを退避させることでプログラムを実行している．ローカルメモリはレジスタよりもアクセスに時間がかかるため，レジスタが溢れている状態は性能に悪影響を及ぼす．`l4dx_nl` カーネルでは，8 バイトのデータがローカルメモリに置かれる状態になっており，プログラムの修正等でレジスタの使用量を減らせれば，性能が改善されると考えられる．

GPU の方が遅いカーネルが存在すると計算速度の面では不利になるが，そのカーネルの計算を CPU で行うとした場合，カーネルの前後で CPU~GPU 間のデータ移動を行うことは避けられない．したがって，計算時間の増加よりもデータ移動の時間の方が長くなるような場合は，トータルで考慮すると GPU で実行した方が高速となる場合が考えられる．

8. さいごに

8.1 まとめ

GT5D の時間発展部分を GPU 化するにあたり，CPU 版でプロファイルを取り，どの箇所の計算に時間がかかっているのかを測定した．プロファイルの結果を基に，いくつかの関数を GPU 化し，およそ 80%の計算を GPU で行えるようになった．

`bcdf` 関数の計算と通信のオーバーラップによる性能改善では，計算の方が早くおわってしまうため，通信の完了を 2.2ms 待ってしまうが，関数一回あたり 21ms の性能改善が達成できた．時間発展全体で性能を評価すると，GPU を用いる場合の方が 1 回あたり 1.85 倍高速に計算できた．また，オーバーラップによる性能向上は 1.44 倍であり，通信時間が性能に大きな影響を与えていることがわかる．

表 7 各カーネルの実行時情報．ただし smem はシェアードメモリの使用量 (バイト)，倍率は CPU に対する速度差を示す

関数名	カーネル名	レジスタ	smem	Occupancy	倍率
timedev1	timedev1	18	0	1.000	3.37
timedev2	timedev2	20	0	1.000	2.19
timedev3	timedev3	21	2048	0.833	2.33
timedev4	timedev4	20	0	1.000	2.13
timedev5	timedev5	30	6144	0.667	2.11
timedev6	timedev6	21	0	0.833	3.25
timedev7	timedev7	19	0	1.000	3.31
timedev8	timedev8	21	0	0.833	3.18
timedev9	timedev9_1	19	0	0.093	2.07
	timedev9_2	20	0	1.000	
	timedev9_3	20	0	1.000	
	timedev9_4	20	0	1.000	
	timedev9_5	20	0	1.000	
	timedev9_6	20	0	1.000	
	timedev9_7	20	0	1.000	
	timedev9_8	20	0	1.000	
	timedev9_9	20	0	1.000	
l4dx_l	l4dx_l	56	27264	0.167	0.77
	reduce	22	0	0.021	
l4dx_nl	l4dx_nl	63	14336	0.333	1.19
	reduce	23	0	0.021	
l4dx_r	l4dx_r	29	4352	0.667	2.16
l4dx_s	l4dx_s	38	3200	0.500	1.42

8.2 今後の課題

現時点では，時間発展部分の 75%の範囲しか GPU 化できておらず，GPU 化できていない計算カーネルを CPU で実行しているため，フル GPU 化ならば不必要になる CPU と GPU 間のデータ転送が発生しており，性能に悪影響を与えていると考えられ，GPU 化の範囲を広げていくことが今後の課題の 1 つである．

GPU の方が CPU よりも遅いカーネルがいくつかあり，それらの性能を改善することで，さらなる高速化が得られると考えられる．また，`lfp` 関数は通信と計算のオーバーラップが可能な構造をしており，通信時間を隠蔽することで，さらなる高速化が得られると考えられ，今後改善して行きたい．

謝辞 本研究の一部は日本学術振興会・多国間国際研究協力事業（G8 Research Councils Initiative）プログラム研究課題「エクサスケール規模の核融合シミュレーション」による．また，本研究では筑波大学計算科学研究センター平成 24 年度学際共同利用プログラム課題「核融合シミュレーションの GPU 化と性能評価」により，大規模 GPU クラスタ HA-PACS を利用した．同センター並びに関係各位に謝意を表する．

参考文献

[1] 筑波大学 計算科学研究センター: HA-PACS ベースクラスタ, 入手先 ⟨http://www.ccs.tsukuba.ac.jp/CCS/research/project/ha-pacs/cluster⟩.
[2] TSUBAME 計算サービス: TSUBAME2 ハードウェア構成, 入手先 ⟨http://tsubame.gsic.titech.ac.jp/hardware-architecture⟩.
[3] Top500 Supercomputer Sites(online), 入手先 ⟨http://top500.org/⟩.
[4] Y.Idomura, M.Ida, T.Kano, N.Aiba and S.Tokuda: Conservative global gyrokinetic toroidal full-f five-dimensional Vlasov simulation, *Computer Physics Communications*, Vol. 179, pp.391-403, (2008).
[5] ITER(online), 入手先 ⟨http://www.iter.org⟩.
[6] S. Konishi, S. Nishio, K. Tobita and The DEMO design team: “DEMO plant design beyond ITER”, *Fusion Engineering and Design*, Vol.63, pp.11-17, (2002).
[7] S.Jolliet and Y.Idomura: Simulating Plasma Turbulence with the Global Eulerian Gyrokinetic Code GT5D, *Progress in NUCLEAR SCIENCE and TECHNOLOGY*, Vol.2, pp.85-89 (2011).
[8] NVIDIA Developer Zone: CUDA Toolkit(online), 入手先 ⟨http://developer.nvidia.com/cuda-toolkit⟩.
[9] 藤田典久, 奴賀秀男, 朴泰祐 and 井戸村泰宏: 核融合シミュレーションコードの GPU クラスタ向け最適化, 情報処理学会研究報告, Vol.2012-HPC-135, (2012).
[10] GPGPU におけるデータ転送とカーネル実行のヒューリスティックスケジューリング, 情報処理学会研究報告, Vol.2011-HPC-129, (2011).
[11] PGI: CUDA Fortran(inline), 入手先 ⟨http://www.pgroup.com/resources/cudafortran.htm⟩.