Vol.2012-HPC-135 No.41
2012/8/3

# 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

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

**概要**：GT5D は核融合シミュレーションを行うプログラムであり，トカマクプラズマ中の乱流現象を対象とする．本研究では，1 ノードに複数 GPU が搭載されている GPU クラスタ向けに GT5D の最適化を行う．GT5D の実行時間を分析することにより，GPU 化の対象を時間発展部分と定め，時間発展の約 75%の計算に対して GPU 化を行なった．関数単位の性能評価では，CPU と比べて最大 3.37 倍の高速化が得られ，時間発展全体の性能評価では，CPU と比べて 1.21 倍の高速化が達成できた．しかしながら，一部の関数や CPU~GPU 間の通信の最適化が不十分であり，計算性能の向上のボトルネックとなっており，今後の改善が課題である．

## 1. はじめに

従来は 3D グラフィックスを描画するための装置としてのみしか利用されていなかった 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．

GPU 単体では，プログラムの実行やデータ転送といった動作をすることはできない．CPU と GPU 間は PCI Express インターフェイスによって接続されており，CPU からの命令発行やデータ転送は PCI Express を通じて行われる．PCI Express Gen2 16 レーンの帯域は上り 8GB/sec，下り 8GB/sec の全二重通信であり，CPU のメモリ帯域や，GPU のメモリ帯域と比較して細いため，ボトルネックとなりやすい．また，CPU から GPU への命令も PCI Express を通じて送られるため，GPU の操作はオーバーヘッドを伴う．

本研究では，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) [3] の GPU 化のための事前評価を行い，それを元に GPU 化を行う．GT5D は，独立行政法人日本原子力研究開発機構で開発されたプログラムであり，Fortran で記述されている．MPI と OpenMP によるハイブリッド並列化が既に成されており，それらを基に GPU 化を進める．また，GPU クラスタを効率よく利用するために，1 ノード複数 GPU の利用を行う．

## 2. GT5D

GT5D 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．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 $(\rho, \chi, \xi)$(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．

[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] 独立行政法人日本原子力研究開発機構
Japan Atomic Energy Agency
[a)] fujita@hpcs.cs.tsukuba.ac.jp

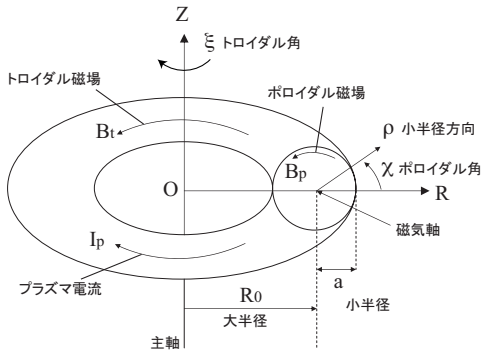


**図 1** GT5D におけるトーラス配位.

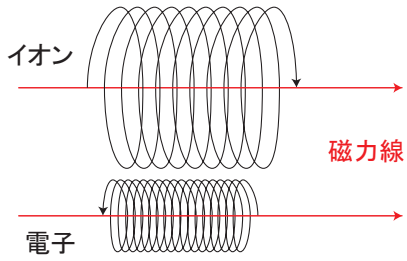


**図 2** プラズマ粒子の運動.

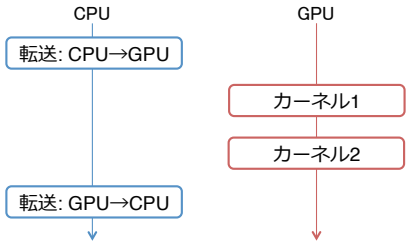


**図 3** 一般的な CUDA プログラミングの流れ.

## 3. NVIDIA CUDA 環境

### 3.1 CUDA プログラミング

NVIDIA 社の GPU で汎用計算を行うための開発環境を CUDA と呼ぶ [4]. 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.

CUDA プログラミングにおいて，GPU で行う処理は関数単位で記述しカーネルと呼ばれる. CPU と GPU はメモリ空間が分れているため，CPU から GPU のメモリ，あるいは GPU から CPU のメモリへ直接アクセスできない. したがって，計算や通信に必要なデータは，`cudaMemcpy` といった API を用いて，CPU と GPU の間でデータを転送する. **図 3** の様に，計算用のデータを GPU へ送り，カーネルを起動して計算を行い，結果を GPU から転送するという手順が基本的な CUDA プログラミングの流れとなり，転送に伴うオーバーヘッドが存在する.

```
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 4** PGI CUDA Fortran の例.

### 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 [5] を利用する.

PGI CUDA Fortran は，CUDA C/C++のように，Fortran 2003 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 4** 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. 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 [2]. HA-PACS 1 ノードの性能諸元を**表 1** 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 Inifiband HBA が搭載され，**図 5** 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

**表 1** 計算機環境.

| | |
|---|---|
| CPU | Intel Xeon E5-2670 × 2 (2.6GHz) |
| CPU メモリ | 128GB |
| GPU | NVIDIA Tesla M2090 × 4 |
| GPU メモリ | 6GB/GPU |
| OS | CentOS 6.1 |
| CUDA Toolkit | 4.1 |
| PGI Compiler | 12.2 |
| PGI Compiler Options | `-fastsse -Mcuda=4.1` `-Mipa=fast,inline` |
| MPI | MVAPICH2 1.8 |
| インターコネクト | Infiniband QDR 4 レーン, 2 レール |

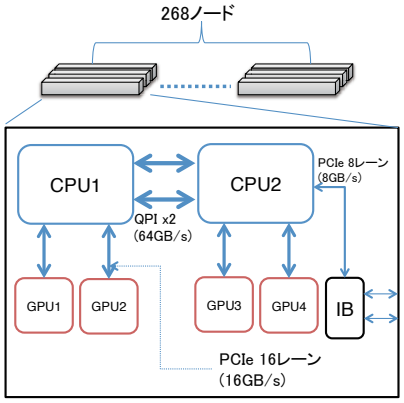


**図 5** ノード内のコンポーネント間接続の概念図.

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 Architecture) 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 QDR2 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，268 ノードからクラスタが構成されている．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．

## 5. GT5D の GPU 化

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

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

**図 6** numactl コマンドの例.

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．

### 5.1 プロセス毎の GPU の割り当ての方針

GT5D の GPU 化にあたり，1 つの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 7** に示す割り当て方針を採用する．1 つのプロセスに対して 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．HA-PACS では，1 ノードにつき，2 つの CPU が搭載され，各 CPU に対して 2 つの GPU が接続されている．したがって，1 ノードあたり 4 プロセスを起動し，MPI 並列におけるプロセス番号と 4 の剰余を取り，操作する GPU を決定する．また，HA-PACS はノードあたり 16 コアを持つため，プロセス当りの OpenMP のスレッド起動数を `OMP_NUM_THREADS` 環境変数を使用し 4 に設定する．

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 6** 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．

本方針では，GT5D が持つ既存の MPI 並列化のコードを再利用でき，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 の対応を取りやすいこと，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．

### 5.2 時間発展部の流れ

時間発展 1 回の時間と呼び出し回数測定結果を**表 2** 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図を**図 8** に示す．時間発展の中には，内部ループ (図 8 の波線部) が 2 つあり，収束判定が満されるまで繰り返される．`l4dx_s` 関数は内部

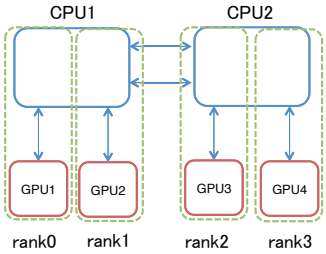


**図 7** プロセス毎の CPU コアと GPU の割り当ての方法.

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 2 のその他の部分に該当する．

**図 9** 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 2 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く，時間発展内におけるほとんどの通信を占める．`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．ただし，GT5D の MPI 並列の分割数は $n_R, n_Z, n_\mu$ の 3 変数で表わせられ，$n_R$ 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．

### 5.3 GT5D のプロファイリング

時間発展部の中で，どの処理に時間がかかっているかを調査するために，各処理にかかる時間の計測を行う．関数に入る前と出た後に，MPI 関数の 1 つである `MPI_Wtime` を用いて時間計測を行う．

時間測定は以下の条件で行う．測定は HA-PACS 1 ノードを使用し，4 MPI プロセスを立ち上げプロセス当り 4 スレッドの設定を用いて，CPU のみで計算を行う．GT5D のメッシュ分割数は $(N_R, N_\zeta, N_Z, N_{v_\parallel}, N_\mu) = (64, 64, 64, 64, 4)$，MPI 分割数は $(n_R, n_Z, n_\mu) = (1, 1, 4)$ とする．$n_R = 1, n_Z = 1$ であるため，`bcdf` 関数は通信を行わないことに注意が必要である．また，計算結果の入出力を除いて，各 MPI プロセスの仕事量は均一であるため，ランク 0 で時間計測を行う．

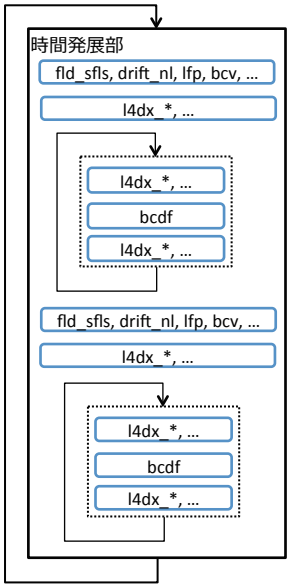


**図 8** GT5D の時間発展部の概要図．ただし，波線部は内部ループを表す．

**表 2** GT5D の時間計測結果,

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

### 5.4 GT5D の GPU 化の方針

GT5D の GPU 化の方針としては，時間発展部分を GPU 化する対象とする．そして，時間発展部分の中で，特に重い関数である `l4dx_s`, `l4dx_r`, `l4dx_l`, `l4dx_nl` の 4 関数を中心に考える．`l4dx_r`, `l4dx_l`, `l4dx_nl` の 3 つの関数は，合計で実行時間の 8.26%しか占めないが，MPI 通信を含まないため GPU のみで処理が完結すること，`l4dx_s` 関数と処理内容が似ており GPU 化するコストが低いこと，CPU~GPU 間のデータ転送を削減する必要があるといった理由により GPU 化の対象とする．CPU と GPU の間の通信速度はあまり早くなく，時間発展全体の高速化のために CPU と GPU の間のデータ移動は，必要最低限に抑えなければならない．

前章で延べた通り，`l4dx_s`, `l4dx_r`, `l4dx_l`, `l4dx_nl` の呼出の周辺に，小さな DO ループがいくつかあり，時間発展の中で約 40%の時間を占めている．個々のループは

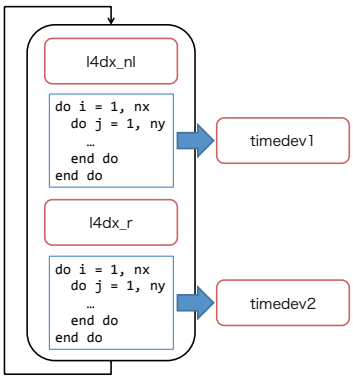


**図 9** GT5D の時間発展部分の GPU 化の概要図．ただし，青で囲まれた部分は CPU で処理を行うことを示し，赤で囲まれた部分は GPU で処理を行うことを示す．

計算量としては多くないが，CPU で計算するとなると，`l4dx_r` 関数の場合と同様に CPU~GPU 間のデータ転送が発生し，性能に悪影響を及ぼすため，図 9 のように関数化 (`timedev1`~`timedev9`) し，GPU で計算する．ある計算を GPU 化するかどうか決定する際は，計算量だけでなく，CPU~GPU 間のデータ転送量も重要となる．また，GPU よりも CPU で計算する方が速い処理の場合でも，データ移動の時間を含めて比較検討しなければならない．

## 6. 性能評価

### 6.1 関数毎の性能評価

関数毎の性能評価では，各種パラメータを次のように設定し測定を行った．本測定では GT5D を使用せず，開発用のテストプログラムを用いて性能評価を行なっている．開発用のテストプログラムとは，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ものであり，MPI 並列を使用せず OpenMP 並列のみを使用する．実行時のメッシュ分割数は $(N_R, N_\zeta, N_Z, N_{v_\parallel}) = (64, 64, 64, 64)$，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 と性能を比較した結果を**表 3** と**図 10** 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 と性能を比較した結果を**表 4** と**図 11** 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 倍高速と，GPU で実行する方が遅くなってしまった．

### 6.2 時間発展全体の性能評価

時間発展全体の性能評価では，各種パラメータを次の

**表 3** `timedev1`~`timedev9` 関数の性能評価．

| 関数名 | CPU[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 |

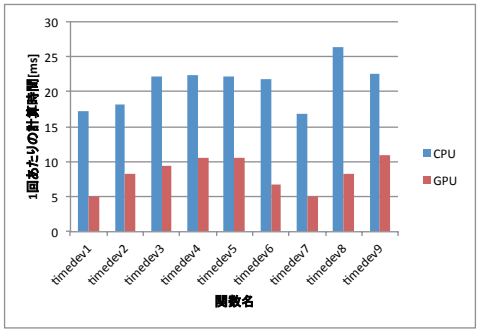


**図 10** `timedev1`~`timedev9` 関数の性能評価のグラフ．

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

| 関数名 | CPU[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 |

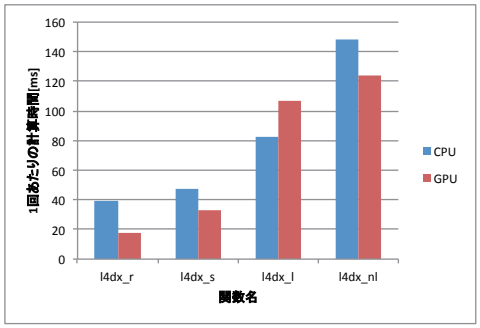


**図 11** `l4dx_r`, `l4dx_s`, `l4dx_l`, `l4dx_nl` 関数の性能評価のグラフ．

ように設定し測定を行った．GT5D のメッシュ分割数は $(N_R, N_\zeta, N_Z, N_{v_\parallel}, N_\mu) = (64, 64, 64, 64, 4)$，MPI 分割数は $(n_R, n_Z, n_\mu) = (1, 1, 4)$ とし，HA-PACS 1 ノードを使用した．プロセス，スレッド，GPU の割り当ては前節で延べた通りである．

時間発展 1 回あたりの計算時間は，CPU が 7.05 秒，GPU が 5.81 秒と，GPU の方が 1.21 倍高速に計算できるという結果が得られた．なお，上記計算時間は MPI 通信および CPU~GPU 間の通信時間を含んでいる．

bcdf 関数周辺での CPU∼GPU 転送時間を計測し，GPU から CPU への転送には 26.1ms，CPU から GPU への転送には 27.9ms かかることがわかった．なお，CPU∼GPU 間の転送は，非同期転送 API cudaMemcpyAsync を用いて行なっており，CPU 上での時間経過を計測する MPI_Wtime 関数では，正しい値が得られないため，CUDA Toolkit に含まれている CUDA Compute Profiler を用いて計測した．bcdf 関数は袖領域の交換を行うための関数であり，本来は GPU 側にある配列の一部があれば通信できるが，現在の実装では配列全体を転送しているため無駄が多く，改善が必要である．本評価で用いたパラメータでは，転送している配列のサイズは 163MB となるが，袖領域はその中の 16MB 分にしか過ぎず，最適化を行うと通信量がおよそ 10 分の 1 になる．しかしながら，不連続な領域の転送となるため，最適化を行う前に事前評価を行う必要がある．

## 7. まとめと今後の課題

本研究では，核融合シミュレーションコード GT5D の GPU 化を行なった．関数単位では，CPU と比較し，最大で 3.37 倍の高速化が得られたものの，CPU よりも遅い関数があり，最適化が不十分である．

1 ノードあたり 4GPU を利用し，時間発展部全体を実行した場合，1.21 倍の高速化が達成できた．しかしながら，bcdf 関数の周辺で通信と計算のオーバーラップなど，さらなる最適化を行う余地は残っており今後の課題である．

**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，本研究の遂行に当たり，HA-PACS を利用させて頂いた筑波大学計算科学研究センターに謝意を表する．

### 参考文献

[1] TSUBAME2 ハードウェア構成 — TSUBAME 計算サービス. http://tsubame.gsic.titech.ac.jp/hardware-architecture
[2] HA-PACS ベースクラスタ — 筑波大学 計算科学研究センター. http://www.ccs.tsukuba.ac.jp/CCS/research/project/ha-pacs/cluster
[3] Y.Idomura, M.Ida, T.Kano, N.Aiba, S.Tokuba, Conservative global gyrokinetic toroidal full-f five-dimensional Vlasov simulation, Computer Physics Communications, 179, 391-403, 2008
[4] CUDA Toolkit — NVIDIA Developer Zone http://developer.nvidia.com/cuda-toolkit
[5] PGI — Resources — CUDA Fortran http://www.pgroup.com/resources/cudafortran.htm