Vol.2013-HPC-140 No.39
2013/8/2

GPU クラスタ HA-PACS における核融合シミュレーションコードの性能評価

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 はトカマクプラズマ中における乱流現象を対象として，核融合シミュレーションを行うプログラムである．これまでの研究で，GT5D を 1 ノードに複数の GPU を搭載するクラスタに向けて最適化を行い，GT5D の時間発展を行う部分のおよそ 80%の部分について GPU 化と，一部通信において計算とのオーバーラップを実装した．大規模マルチ GPU クラスタ HA-PACS を用いて実験を行い，CPU のみの実行と比較してオーバーラップを有効にした場合で 1.85 倍の性能が得られていた．本稿では，時間発展の中で GPU 化を出来ていなかった残りの 20%の部分について GPU 化を行なう．これまでの研究と同様に HA-PACS を用いた性能測定で，オーバーラップありの場合において，CPU のみを用いる場合に対し 2.03 倍の性能が達成された．

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．

一般的に，CPU と GPU は PCI Express を介して接続されており，GPU に対する指示や GPU のメモリに対する読み書きは PCI Express を経由して行なわれる．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．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．

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

世界のスーパーコンピュータのランキングである Top500[1] リストの 2013 年 6 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，2 位 Titan, 10 位 Tianhe-1A, 16 位 Nebulae の 4 システムが GPU によるアクセラレータを搭載しており，2 位の Titan は Linpack 性能で 17.5PFLOPS を記録している．

一方，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，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．

本研究では，核融合シミュレーションコード GT5D[2] を大規模 GPU クラスタ向けに最適化する．GT5D のプログラムは Fortran で書かれており，非常に巨大で複雑であるため，Fortran を用いて GPU プログラムを開発できる環境を使用する．また，CPU と GPU の間のデータの移動コストを最小にすることを目標とし，一部のノード間通信において，通信と計算のオーバーラップを実現する．本研究の目的は，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ことである．

© 2013 Information Processing Society of Japan

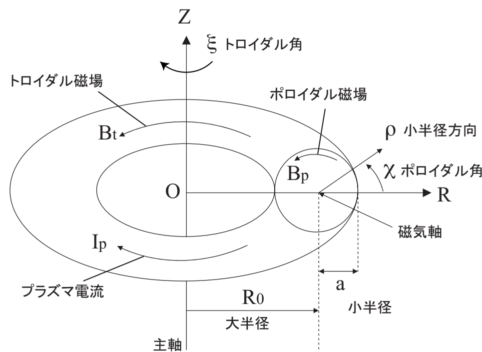

図 1 一般的なトーラス配位．図の中で GT5D で座標変数として使われているものは (R, Z, ζ)．ただし，$\zeta = -\xi$ である

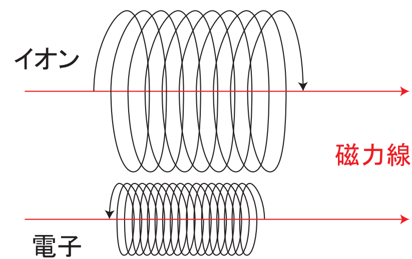

図 2 磁力線に沿って運動する荷電粒子の図．荷電粒子はローレンツ力によって磁力線の周りを旋回移動するため，粒子の速度を磁力線に平行方向の速度 $v_{\parallel}$，磁力線の垂直方向の速度 $v_{\perp}$, 旋回位相 ϕ の 3 変数で表わすことができる．このうち旋回位相は旋回平均によって消去でき，計算量を削減できる

先行研究 [9], [10] では，CPU 版の GT5D の実行プロファイルに基づき，コードをインクリメンタルに GPU 化してきたが，本稿では時間発展全体の GPU 化が完了したのでこれを報告する．

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) [2] 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．

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なくて済むが，ITER[3] や DEMO[4] といった次世代の大型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 [5]，より高速な計算機が求められている．

2.1 NVIDIA CUDA 環境

CUDA[6] 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．

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 [7] 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 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．

3. 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 [8]．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 dual rail の Inifiband HBA が搭載され，図 4 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 のメモリが接続され，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

```
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 An example of PGI CUDA Fortran

表 1 Machine Environment

CPU	Intel Xeon E5-2670 × 2 (2.6GHz)
	CPU (8 cores/CPU) × 2 = 16 core
CPU Memory	128GB, DDR3 1600MHz
GPU	NVIDIA Tesla M2090 × 4
GPU Memory	6GB/GPU
OS	CentOS 6.1
Intel MKL	Intel Compiler 13.0 付属
CUDA Toolkit	ver. 4.1
PGI Compiler	ver. 12.10
PGI Compiler Options	`-fastsse -Mcuda=cc20,4.1,flushz`
	`-Mipa=fast,inline`
MPI	MVAPICH2 1.8
Interconnect	Infiniband QDR 4 Lanes, 2 Rails

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．

4. GT5D の GPU 化

4.1 GT5D の GPU 化の方針

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．

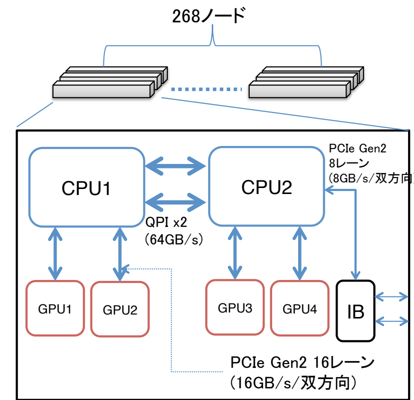

図 4 ノード内のコンポーネント間接続の概念図

表 2 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		

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 2** に示す．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るが，計算時間を関数単位で計測したため，それらのループが表 2 のその他の部分に該当する．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．

これまでの研究 [10] では，表 2 にある関数の中で，その他に相当する部分と，`l4dx_s`, `l4dx_r`, `l4dx_l`, `l4dx_nl` の 4 つの関数について，GPU 化を行なった．これは，GT5D 全体を最初から GPU 化するのは大変なため，計算時間の重い部分からインクリメンタルに GPU 化を行なったためである．また，`bcdf` 関数で行なわれる MPI 通信について計算とのオーバーラップを行い性能改善を確認した．本稿では，さらに GPU 化する範囲を広げて，性能改善を計ることとし，新たに `lfp`, `fld_sfls`, `drift_nl`, `dn3d`, `bcv` の 5

© 2013 Information Processing Society of Japan

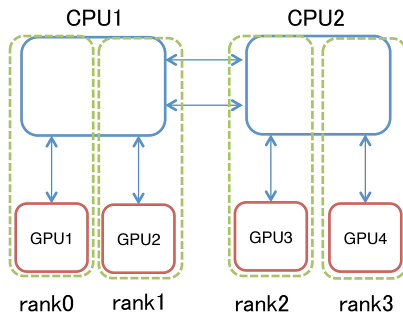

図 5 MPI プロセス毎の CPU コアと GPU の割り当ての方法

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

図 6 numactl コマンドの例．0 番の CPU を利用し，メモリは CPU のローカルメモリに優先して割り当てる設定で GT5D コマンドを利用する

関数について GPU 化を行い，MPI 通信やデバッグ用の計算などを除くほぼ全ての計算について GPU 化を行なった．

4.2 プロセスと GPU の割り当て

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．numa 環境を考慮しており，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，全体の性能向上が期待できる．CPU の指定には numactl コマンドを使用し，GPU の選択には CUDA API である `cudaSetDevice` を使用する．numactl コマンドの例を**図 6** に示す．

4.3 FFT と Lapack ライブラリ

GT5D は `fld_sfls` 関数で FFT ライブラリおよび Lapack ライブラリを使用している．FFT ライブラリは，CPU の実装を切替えられ，いくつかのライブラリに対応しており，コンパイルタイムに切替えられるようになっている．

HA-PACS では，CPU の FFT ライブラリおよび Lapack ライブラリとして，Intel MKL (Math Kernel Library) [11] を使用する．GPU での実行の際は CUDA Toolkit に付属しているライブラリを使用する．CUDA Tookit の同梱ライブラリとして，CUFFT[12] という FFT ライブラリが提供されているが，Lapack ライブラリについては提供されていない．しかしながら，BLAS ライブラリである CUBLAS[13] は提供されているため，CUBLAS を用いて必要な Lapack ルーチンを実装して利用している．

5. 性能評価

5.1 `lfp` 関数におけるデータアンパックの最適化

`lfp` 関数において，MPI 通信で受け取ったデータを GPU に書き込む処理がある．MPI 通信が細切れになることを避けるために，データはパッキングされた状態で通信されるため，GPU に戻す際にパッキングを解くという処理が必要になる．

表 3 `lfp` 関数におけるデータアンパック手法の比較

処理手法	処理時間 [ms]
ケース 1	438.4
ケース 2	37.9
ケース 3	28.9

データをアンパックする際の手法として考えられるのは，以下の 3 つである．

ケース 1 cudaMemcpy を細切れに行い，データを転送とアンパックを兼 る方法．

ケース 2 CPU 側でアンパックを行い，アンパックしたデータを cudaMemcpy で GPU に送る方法．

ケース 3 まず cudaMemcpy で GPU にデータを送り，GPU 側でアンパックする方法．

3 つの手法について，それぞれの性能評価の結果を**表 3** に示す．ケース 1 は cudaMemcpy を利用してデータアンパックを行う方法であるが，cudaMemcpyAsync および cudaMemcpy2DAsync が合計で 163840 回呼ばれてしまう．cudaMemcpy のオーバーヘッドは一回あたり約 3μs であるが，cudaMemcpy のオーバーヘッドの蓄積により性能が悪化している．ケース 2 とケース 3 を比較すると，CUP~GPU 間のデータ転送量は両者で差はないため，ケース 2 とケース 3 の差はアンパックにかかる時間の差である．

結果として，ケース 3 の GPU 側でアンパックを行う方式が最も性能が高いことがわかる．したがって，時間発展全体の性能評価を行う際はケース 3 の実装を採用する．

5.2 `fld_sfls` 関数における最適化

これまで，全ての計算を GPU にオフロードするという方針で実装を行なったが，しかしながら，`fld_sfls` 関数の中で 4 次元の 2 つの配列 (fmat, dfem) のデータを 2 次元の配列 (phw) へリダクション (加算) している部分があり，phw への書き込みがスレッド間で衝突するため，GPU 向けの実装が難しい．実測の結果，この部分の処理を GPU で行うと，CPU で処理する場合より大幅に時間がかかり，GPU から CPU に逆にデータを送るオーバーヘッドを含めても，CPU で実行した方が高速であることがわかった．したがって，この部分については CPU 側で処理を行うこととする．CPU 側で処理を行う理由について，以下に詳述する．

CPU 向けの実装では，OpenMP によるスレッド並列を用いてマルチコア対応が行なわれている．その際，OpenMP parallel ディレクティブの reduction 節を利用して phw への加算が壊れない様に制御されている．すなわち，OpenMP を用いる場合は，書き込み先である配列 phw をスレッド

表 4 `fld_sfls` 関数におけるリダクション処理の比較

実装	処理時間 [ms]
CPU	0.5
GPU (A)	28.2
GPU (B)	178.1

毎に複製し，スレッドローカルな演算が終了した後に，スレッド間の加算を行うため性能面での障害とはならない．

GPU における実装で，CPU と同じ戦略を取ることはメモリの制約などから困難である．GPU におけるスレッド数は数千になることが一般的であり，phw をスレッド毎に複製してしまうと大量のメモリが必要になってしまうためである．また，スレッド毎にアクセスするメモリ領域が大きく離れてしまうため，キャッシュヒットも期待できない．

以上の理由より，この処理を GPU 化するためには，CPU における実装を流用したものではなく，GPU 向けの最適な実装が必要である．実験結果を**表 4** に示す．

表 4 における GPU(A) は，少ないスレッド数で GPU カーネルを起動し計算させる方式である．1 ブロック 34 スレッドのパラメータでカーネルを起動し，途中まで 34 スレッドで計算させた上で，代表となる 1 つのスレッドが phw への更新を行い書き込みの衝突を回避する．表 4 における GPU(B) は，phw への書き込みに atomic 命令を利用する方式である．Fermi アーキテクチャは倍精度浮動小数点数への atomic 加算をサポートしていないため，CAS (Compare and Swap) を用いて擬似的に atomic 加算を実現している．

2 つの実装を行い性能比較を行なったが，どちらも性能が十分ではなく，必要なデータを CPU 側に転送し，CPU でリダクションを行なう方が速いという結果となった．実際には，本処理の後で MPI 通信があり，GPU 側で処理を行なったとしてもデータを CPU に戻さなければならず，転送が増えることによる性能への影響は小さいと考えられる．

5.3 時間発展全体の性能評価

本節では，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 $(n_R, n_Z, n_\mu) = (4, 4, 4)$ とする．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．HA-PACS 上の各ノードの 4 台の GPU を，それぞれ 1 つの MPI プロセスで制御するため 1 ノードあたりのプロセス数は 4，よって 64 プロセス = 16 ノードとなる．

時間発展 1 回あたりの全計算時間を表 5 に示す．「オーバーラップのありなし」は，`bcdf` 関数における MPI 通信と GPU 内演算のオーバーラップのありなしを示す．また，“GPU Old” はこれまでの研究 [10] による実装を表し，時間発展のおよそ 80%の計算が GPU で行なわれていた場合のものである．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 1.33 倍，オーバーラップありで 2.04 倍の高速化が得られていることがわかる．さらに，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 1.53 倍高速であることがわかる．また，本稿で GPU 化の範囲を広げたことにより，以前の実装と比較して 1.85 倍から 2.04 倍 (ともにオーバーラップあり) と，より高速になっている．

表 5 時間発展 1 回あたりの計算時間

	Time[s]	Ratio to CPU
CPU	15.7	
GPU Old (overlap なし)	12.2	×1.29
GPU Old (overlap あり)	8.5	×1.85
GPU (overlap なし)	11.8	×1.33
GPU (overlap あり)	7.7	×2.04

5.4 処理時間の内訳

GPU 化を行った後のオーバーラップありの場合のプロファイル結果を**表 6** に示す．Total は時間発展部全体にかかった時間を表し，オーバーラップありの場合を測定しているため，`bcdf` 関数の前後の計算の違いで 5 つの種類があり，それぞれ前後の計算を含めた時間となっている．また，`bcdf` 関数の時間計測と，オーバーラップで呼ばれる関数の時間計測は重複している．例えば，“bcdf overlap: timedev2 + l4dx_s” の項目は，timedev2 inner, timedev2 boundary, l4dx_s inner, l4dx_s boundary, l4dx_s reduce の 5 項目を含んでいる．したがって，Total を除く全ての行の時間の和は Total と一致しない．計測は時間発展一回分であり，パラメータは**表 5** を計測した際に用いたものと同じである．ただし，表 5 は 5 回の時間発展ループの計算を行い，1 回あたりの平均時間を求めたものに対し，表 6 は初回のループについてのみ計測をしたものであるため，両者の計算時間は一致しない．時間発展のループは，実行状態によって内部ループの反復回数が異なり，それがループ 1 回にかかる時間が変化する要因である．表 6 の中で，“bcdf overlap: timedev4 + l4dx_s” と timedev5 の項目が内部ループ内で呼ばれているため，呼び出し回数が変化する可能性を持つ．

表 6 より，“bcdf overlap: timedev4 + l4dx_s” と lfp の 2 項目で，全体の 71%を占めていることがわかる．各関数単位でみると，timedev4 と l4dx_s の比較では l4dx_s の方が占めている時間が多く，l4dx_s の 3 種 (boundary, inner, reduce) をあわせると全体の 34%に相当し，timedev4 の 3 種をあわせると全体の 9%に相当する．また，lfp も 15%の時間を占めている．

© 2013 Information Processing Society of Japan

表 6 GPU 化後のプロファイル結果

要素	時間 [ms]	回数
Total	6608.37 ms	1
bcdf overlap: timedev2 + l4dx_s	155.95 ms	2
bcdf overlap: timedev3 + l4dx_s	143.55 ms	2
bcdf overlap: timedev4 + l4dx_s	3651.28 ms	51
bcdf overlap: timedev4 + timedev6	47.11 ms	1
bcdf overlap: timedev4 + timedev8	47.48 ms	1
bcv	9.98 ms	2
dn3d	14.56 ms	2
drift_nl	26.85 ms	2
fld_sfls	165.59 ms	2
l4dx_l	217.91 ms	2
l4dx_nl	261.95 ms	2
l4dx_r	38.21 ms	2
l4dx_s boundary	773.69 ms	55
l4dx_s inner	1491.86 ms	55
l4dx_s reduce	1.57 ms	55
lfp	1031.90 ms	2
timedev1	5.45 ms	1
timedev2 boundary	3.68 ms	2
timedev2 inner	13.93 ms	2
timedev3 boundary	4.23 ms	2
timedev3 inner	14.93 ms	2
timedev3 reduction	0.04 ms	2
timedev4 boundary	123.37 ms	53
timedev4 inner	465.77 ms	53
timedev5	546.09 ms	51
timedev6	7.19 ms	1
timedev6 boundary	2.96 ms	1
timedev6 inner	4.32 ms	1
timedev7	5.46 ms	1
timedev8	8.83 ms	1
timedev8 boundary	3.64 ms	1
timedev8 inner	5.36 ms	1
timedev9	10.71 ms	1

表 7 lfp 関数の 2 回の呼出における内訳

要素	時間 [ms]	割合 [%]
MPI 通信	173.51	16.8
cudaMemcpy	471.53	45.7
計算カーネル	386.89	37.5

timedev4 と l4dx_s は計算のみからなる関数であるが，lfp は MPI 通信を主に行う関数であり，通信に関する時間が大半を占める．lfp 関数内での処理内容に応じた内訳を**表 7** に示す．計算にかかる時間よりも，MPI および CPU∼GPU 間の通信に時間がかかっていることがわかる．lfp は計算と通信のオーバーラップが可能な構造をしており，オーバーラップを実装すれば，最大で 386.89ms の実行時間削減が可能であると考えられる．timedev4 と l4dx_s を構成するカーネルの改善と，lfp 関数における通信最適化の 2 点を改善することで，時間発展全体のさらなる高速化ができると期待できる．これらは今後の課題である．

6. まとめ

本研究では，核融合シミュレーションコード GT5D の完あ GPU 化を行った．大規模 GPU クラスタ HA-PACS 上の性能評価の結果，全体を GPU 化したことにより，CPU のみを使用する場合と比べて時間発展ループ 1 回あたりで 2.04 倍の計算速度を実現した．また，MPI 通信と GPU 内計算のオーバーラップによる性能向上は 1.53 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．GPU 化した後のプロファイル結果より，timedev4 と l4dx_s の 2 つのカーネルによって全体の約 43%の時間がかかっていることが判明した．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 JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」，研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」，および日本学術振興会・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 25 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] Top500 Supercomputer Sites(online), 入手先 ⟨http://top500.org/⟩.
[2] 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).
[3] ITER(online), 入手先 ⟨http://www.iter.org⟩.
[4] 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).
[5] 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).
[6] NVIDIA Developer Zone: CUDA Toolkit(online), 入手先 ⟨http://developer.nvidia.com/cuda-toolkit⟩.
[7] PGI: CUDA Fortran(online), 入手先 ⟨http://www.pgroup.com/resources/cudafortran.htm⟩.
[8] HA-PACS Base Cluster — Center for Computational Science, University of Tsukuba(online), 入手先 ⟨http://www.ccs.tsukuba.ac.jp/CCS/eng/research-activities/projects/ha-pacs/base-cluster⟩.
[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).

© 2013 Information Processing Society of Japan

[10] 藤田典久, 奴賀秀男, 朴泰祐 and 井戸村泰宏: 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, 情報処理学会研究報告, Vol.2012-HPC-138, (2013).
[11] Intel(R) Math Kernel Library (Intel MKL) 11.0 — Intel Developer Zone(online), 入手先 ⟨http://software.intel.com/en-us/intel-mkl⟩
[12] CUBLAS — NVIDIA Developer Zone(online), 入手先 ⟨https://developer.nvidia.com/cublas⟩
[13] CUFFT — NVIDIA Developer Zone(online), 入手先 ⟨https://developer.nvidia.com/cufft⟩

© 2013 Information Processing Society of Japan