Vol.2011-ARC-197 No.11
Vol.2011-HPC-132 No.11
2011/11/28

GPU 向け並列計算フレームワークの提案とGA を用いた性能評価

蔵野裕己[†1] 吉見真聡[†1]
三木光範[†1] 廣安知之[†2]

近年，画像処理専用ハードウェアとして用いられてきた GPU が汎用計算へと応用されるようになり，GPU を用いた多くのソフトウェア開発や研究が盛んに行われている．GPU は多数のコアを備え，それらを用いて並列計算を行うため体積あたりの計算量が高く，消費電力が低いという特徴を持つ．GPU による汎用計算向けの開発環境も多数提供されているが，並列計算は高度なプログラミング技術や専門知識を要するため，開発コストが高い．そこで我々は，GPU による並列計算を容易な手段でユーザに提供する，GPU 向け並列計算フレームワークを提案する．このフレームワークを用いることで，GPU に関する高度な知識をもたないユーザも容易に並列計算を行えるようになり，GPU プログラミングをより一般化することが可能である．なお実装には NVIDIA 社が提供する GPU 向け開発環境である，CUDA を用いた．本研究報告ではこのフレームワークを利用して GA を実装し，並列計算が可能であることを確認した．本フレームワークは複数の GPU 利用や並列化粒度の変更も可能であり，そのような条件と処理速度の関係について検討し，フレームワークに基づく性能評価と議論を行った．

A Proposal of Parallel Computing Framework for GPU and an Evaluation with Genetic Algorithms

Yuki Kurano,[†1] Masato Yoshimi,[†1] Mitsunori Miki[†1]
and Tomoyuki Hiroyasu[†2]

Graphic Processing Unit (GPU), which was traditionally used for image processing, has been widely applied to general computation called GPGPU. Nowadays, a lot of studies using GPUs are progressing and various products are developing. The most famous feature of GPU is hundreds of processing cores lead to low energy consumption compared to their physical volume. Even several developing environments are already provided, software developing cost remains high. Implementation of GPGPU program of the target algorithm exploiting parallelism requires not only realization of the target algorithm, but also knowledge of architecture such as memory hierarchy. We propose a framework, which enables easy implementation of parallel computation using GPU. The framework can lead communization of GPU programming. This paper reports evaluations of the simple genetic algorithms (SGA) implemented on the framework to confirm achieving parallel computation on GPUs. As the framework can customize parallel granularity and the number of GPU, relationship between computational speed and execution condition is also discussed.

1. はじめに

GPU（Graphics Processing Unit）は，従来より画像処理専用のハードウェアとして利用されてきたが，汎用計算へ応用されるようになり多数のソフトウェアが開発され，現在でも多くの研究が進められている[1)2)3)]．従来，コンピュータの計算性能は CPU の演算コアのクロックアップと共に向上していた．しかしクロックアップが頭打ちになり，性能向上の主軸はメニーコアへと移った．一方，GPU の従来からの用途である画像処理では，高度な 3 次元 CG を処理する際に並列性の高い処理を必要とするため，GPU は多数の演算コアを持つ．GPU はこれらのコアを利用した並列処理において高い計算性能を発揮し，また性能当たりの消費電力が低いといった利点も併せ持つ．そのような性能を生かしてスーパーコンピュータに GPU を計算資源として組み込む例が増加しており，2011 年 6 月発表の TOP500 の上位 5 位のマシンの内 3 台が GPU を組み込んだシステムである[4)]．ただしスーパーコンピュータは価格が非常に高く，多くの人が容易に利用することができない．その他の並列計算環境として多数の PC を繋いで構築する PC クラスタがあり，スーパーコンピュータに比べてイニシャルコストは低いが，電気代や整備費などのランニングコストが高く，GPU は価格当たりの性能の点で優れている．

GPGPU の従来の開発環境ではシェーダプロセッサ専用言語を用いていたが，新たに C 言語などの拡張言語である CUDA，OpenCL 等が開発され，GPGPU の開発コストは小さくなった．しかし，並列処理やデバイス（GPU とビデオメモリ）のメモリ構造，コアの構成などのアーキテクチャに関する専門知識が必要であり，高い高速性を得るためにはさらに

†1 同志社大学 理工学部
Faculty of Science and Engineering, Doshisha University
†2 同志社大学 生命医科学部
Faculty of Department of Life and Medical Science, Doshisha University

© 2011 Information Processing Society of Japan

Vol.2011-ARC-197 No.11
Vol.2011-HPC-132 No.11
2011/11/28

深い知識によるチューニングが必要である．

そこで GPU による並列処理に関する知識を有さない人も，容易に GPU を利用可能になるフレームワークを提案する．本フレームワークは関数として呼び出すことで利用可能であり，ユーザの代わりにメモリの確保や開放，デバイスとのデータ通信などを行う．本研究報告では，並列性の高い処理として GA（Genetic Algorithm:遺伝的アルゴリズム）を採用し，一部の処理を GPU 上で並列実行した．その際，パラメータを変更することでフレームワークにもとづく性能評価を行い，またフレームワークについて議論する．

2. CUDA

2.1 CUDA について

本研究報告では，NVIDIA 社が無料で配布している GPU 向け統合開発環境である，CUDA を用いた．CUDA を用いることで，対応する GPU を並列計算機として利用可能になる．また CUDA は，C 言語の構文を用いて記述することができ，デバイス（GPU とビデオメモリ）のメモリ確保，解放，データ転送等，デバイスの操作に関する機能の拡張も行われている．

CUDA では，ホストとデバイスの処理を明確に分けて記述する．デバイスの処理は kernel 関数と呼ばれる特殊な関数に記述して行い，それ以外をホストで処理する．デバイスで処理を行う際の，大まかな処理の流れは以下の通りである．またその様子を**図 1** に示す．

- ホストでデバイスのメモリ確保を行う．
- ホストのメモリからデバイスのメモリへ，計算に用いるデータを転送する．
- デバイスで，転送されたデータを処理する．(kernel 関数)
- デバイスのメモリからホストのメモリへ，計算結果のデータを転送する．
- ホストでデバイスのメモリ解放を行う．

このように，ホストで処理をしつつデバイスに処理を与える．

また CUDA を用いた並列性の高いアプリケーションは数多く開発されている．その例として，多体問題[5]，分子動力学[6] など，並列性が高い問題が挙げられる．

2.2 デバイスの構造と計算資源

デバイスは，計算を行う GPU と，計算に必要な各データを保存するメモリに分割できる．GPU は並列計算を行うための複数の計算資源の集合と考えられる．

計算資源は，**図 2** のように grid，block，thread の 3 つの単位で管理される．grid は複数の block をまとめた単位，block は複数の thread をまとめた単位として定義される．1 つ

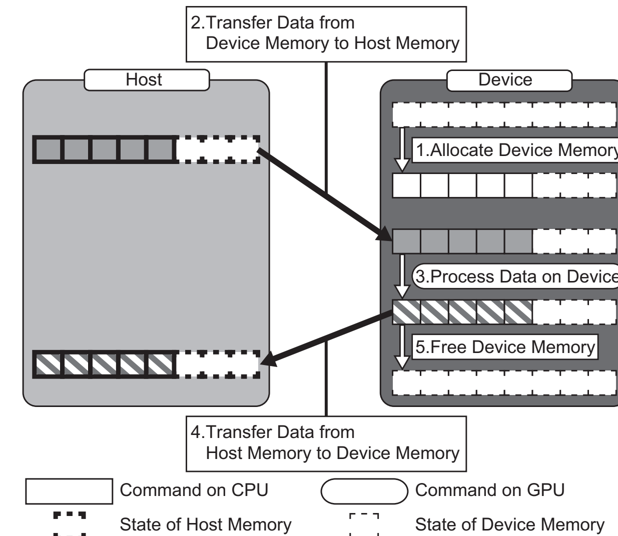

図 1 ジョブオフロードの手順
Fig. 1 The order of Job off-load

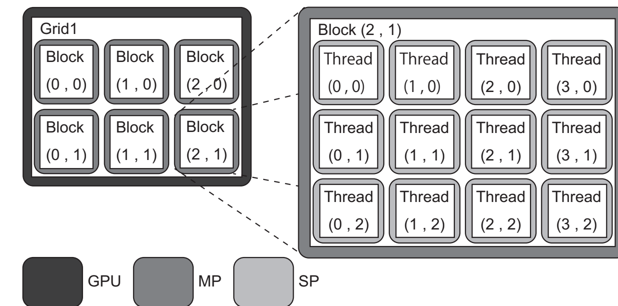

図 2 計算資源の管理方法
Fig. 2 The way to manage the calculation resources

© 2011 Information Processing Society of Japan

の grid は 1 つの GPU に対応する．GPU は複数の Streaming Multi Processor（以降 SM）からなり，1 つの SM は 1 つの block に対応する．SM は複数の Streaming Processor（以降 SP）からなり，1 つの SP は 1 つの thread に対応する．そして各計算資源に割り当てられた処理は同時に実行され，並列処理が実現される．また SM の数を超える block，SP の数を超える thread を宣言すると，すべてを計算資源に割り当てることができないため，一度に並列処理できない．そこで 1 つの block が計算を終えると，待機していた次の block を空いた SM に割り当て，thread も同様に，計算を終えた SP に次の thread を割り当てる．このとき計算資源の割り当てを待つ時間，計算資源が使用されない空き時間が発生するが，これらは thread 数，block 数を調整することで減らすことが可能である．

また実行の際には warp という単位で同じ命令が発行され，処理される．1warp は 32 個の thread からなるため，thread 数は 32 の倍数の場合に効率よく処理される．

3. 関連研究

処理の一部を GPU で行うことで高速化した結果は多数報告されている．しかしそれらは，豊富な経験や知識を有する専門家が多数の調査や実験，調整を繰り返すなど試行錯誤し，チューニングして得た結果である．そのような背景のもと，PGI 社が提供する PGI Accelerator は，GPU 上で容易に並列計算を行うための仕組みとして注目されている[7]．PGI Accelerator は，既存の C 言語，または Fortran 言語のコードに特殊なコメントであるディレクティブ（並列化指示子）を挿入し，データ並列性を持つループ処理を CUDA による GPU 上での並列処理に変換するコンパイラである．また GPU 以外の並列計算機向けにも，同様のディレクティブ挿入型の規格である OpenMP[8] や XcalableMP[9] が存在し，ディレクティブ挿入による自動並列化が一定の支持を得ていることがわかる．

また，PC クラスタによる並列計算環境において，GA の一部の処理を PC クラスタにオフロードし並列に処理するフレームワークに関する研究が行われている[10]．GA を対象とする理由は，並列性の高い処理が計算の大部分を占めるからである．このフレームワークを用いると，関数を呼び出すだけでネットワークで接続されたノードに処理がオフロードされる．検証の結果，フレームワークによって並列処理が容易に実装ができたことと処理の高速化が確認された．このようなフレームワークを用いた並列計算機へのジョブオフロードは，並列計算機上のコードを記述しチューニングする開発者と，フレームワークを利用する開発者がそれぞれの専門分野に集中できる．

ディレクティブ挿入型の規格とフレームワークを用いる手法は，並列処理を容易に実現することを目的としているという点で共通している．以上のように，既存の計算を高速化する研究や報告だけでなく，並列処理を容易に実現しようとする研究も盛んに行われており，その期待は大きいと考えられる．

4. 提案フレームワーク

4.1 フレームワークの概要

本フレームワークはタスクの一部を GPU にオフロードし，並列計算を行うことで処理を高速化する．そしてそのような GPU を利用しやすくするための仕組みを容易に提供することで GPU プログラミングをより一般化することを目的とする．

4.2 フレームワークの構造

フレームワークの構造を**図 3** に示す．図 3 に示すように，フレームワークは 2 つに分けられており，フレームワークの前半はデバイス上のメモリ確保，デバイスへのデータ転送，kernel 関数呼び出しを行う．後半はデバイスからの計算結果取得とメモリの解放を行い，呼び出し元に計算結果を返す．

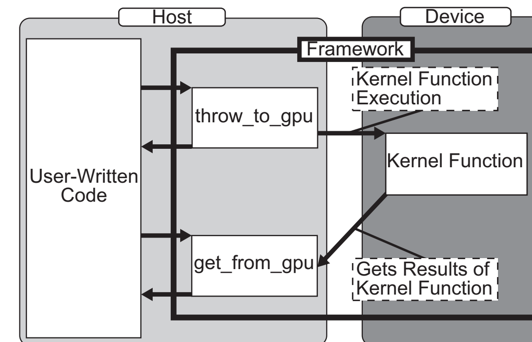

図 3 フレームワークの構造
Fig. 3 Structure of the framework

前半のフレームワークでは kernel 関数を呼び出した直後にホストに制御が戻るため，そこでフレームワークの呼び出し元に戻り kernel 関数実行中にホスト側で別の処理を行う．その後フレームワークの後半を呼び出すことで計算結果の取得を行うが，kernel 関数が完了していない場合は，完了を待ち，その後にデバイスからのデータ転送を行う．

© 2011 Information Processing Society of Japan

表 1 フレームワークを利用したコードの例（kernel 関数）
Table 1 Example of a code which uses the framework(kernel function code).

```
1 __global__ void KernelFunction(double* Data) {
2     int tid = threadIdx.x;
3     int bid = blockIdx.x;
4     int bdm = blockDim.x;
5     data[bdm * bid + tid] = tid * bdm;
6 }
```

表 2 フレームワークを利用したコードの例（ホスト）
Table 2 Example of a code which uses the framework(host).

```
1 double Evaluation(void* Data, int Number_of_Datas) {
2     throw_to_gpu (Data, Data_Type, Number_of_Datas,
      Number_of_Gpus, Parallel_Particle_Size);
3     get_from_gpu(Processed_Data, Data_Type,
      Number_of_Datas, Number_of_Gpus);
4     return Processed_Data;
5 }
```

kernel 関数は，ユーザが自由に記述する．よって CUDA の機能を生かした自由度の高いチューニングが可能である．

4.3 フレームワークの利用方法

表 1，**表 2** にフレームワーク利用時のコードの例を示す．ユーザは本フレームワークを関数としてプログラムにインポートし，kernel 関数の記述と，フレームワークの前半と後半の呼び出しを行う．

kernel 関数は表 1 に示すように，通常の CUDA と同様に記述可能である．これにより，細かなチューニングを行うことができる．また引数にデータが格納された宛先を示すポインタが渡されているので，これを用いて計算を行う．

ホストコードの中で表 2 に示すようにフレームワークの前半を throw_to_gpu 関数で呼び出し，後半を get_from_gpu 関数で呼び出す．throw_to_gpu 関数の引数 Data は，GPU 上で計算に用いるデータであり，void 型のポインタで表される．元のデータの型を Data_Type で渡すことで，配列のサイズ指定や演算時の型指定を行うことができる．また，Number_of_Datas でデータの総数を指定する．

残りの引数はオプションを指定する．まず Number_of_Gpus で使用する GPU の数を決定する．CUDA では複数の GPU を用いて計算を行うことができ，この機能を利用する．そして Parallel_Particle_Size は並列化粒度を指定するオプションである．並列化粒度とは，ある処理を並列処理に分割する際の 1 つの処理の大きさであり，細かくすると処理の数が増えて 1 つの処理が小さくなり，粗くすると処理の数が減って 1 つの処理が大きくなる．本フレームワークでは，並列化粒度を細かくすると block 数が増え，1block あたりの thread 数が減り，粗くすると block 数が減って 1block あたりの thread 数が増える．

get_from_gpu 関数の引数 Processed_Data は処理結果を代入する宛先を格納した void 型ポインタである．その他の引数は throw_to_gpu 関数と同様に指定する．

5. GA の実装

5.1 GPU 上での GA の計算手法

本フレームワークを用いて，GA の一部の計算を GPU 上で並列処理するプログラムを作成した．GA とは，発見的手法により最適解，またはその近似解を求めるアルゴリズムである．ランダムな解を持つ個体を多数用意し，それらの解に変化を与える操作を繰り返し，最適解またはその近似解を得る．それらの操作の中で，それぞれの解がどれだけ適しているか（適合値）を判断する必要があり，この操作を評価と呼ぶ．具体的には問題となる計算式に個体の解を当てはめて計算し，その結果から適合値を得る．そのため，問題となる計算式が複雑である場合は評価に多くの時間がかかり，GA の全処理時間の大部分を評価が占めることになる．また，評価は計算を各々の個体に対して個別に行うため高いデータ並列性を持つ．そこで評価の処理を GPU にオフロードし，並列処理することで高速化を図る．ただし計算式は問題ごとに固有のものであるため，それぞれの計算式にあわせたチューニングが必要である．

今回の評価では GA のテストに用いられる，Rastrigin 関数と Rosenbrock 関数を用いた．それぞれの式を以下に示す．

$$F_{Rastrigin}(x) = 10n + \sum_{i=1}^{n}\left(x_i^2 - 10\cos(2\pi x_i)\right) \quad (-5.12 \leq x_i < 5.12)$$
$$min(F_{Rastrigin}(x)) = F(0,0,\ldots,0) = 0$$

© 2011 Information Processing Society of Japan

$$F_{Rosenbrock}(x) = \sum_{i=1}^{n-1}\left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right) \quad (-2.048 \leq x_i < 2.048)$$
$$min(F_{Rosenbrock}(x)) = F(1, 1, \ldots, 1) = 0$$

なお，式中の n は次元数を表す．これらの関数は総和計算内のそれぞれの計算も独立しており，データ並列性を持つと言える．そのため，これらの関数を実装するには 2 つの並列性を利用する．基本実装ではこれら 2 つの並列性を，ハードウェアの持つ 2 つの並列性に対応させる．具体的には 1 つの block に 1 つの個体を割り当て，1 つの thread に総和計算内の 1 次元分の計算を割り当て，計算を行う．

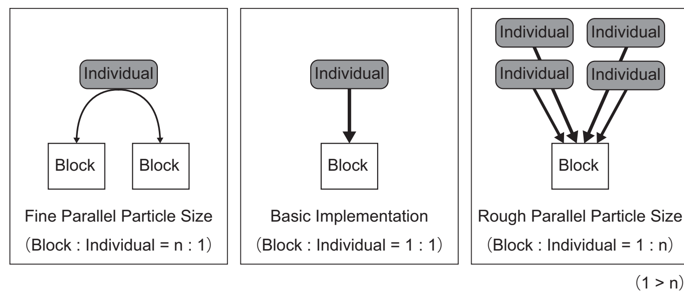

図 4 並列化粒度
Fig. 4 Parallel Particle Size

そして全ての thread 内の計算結果の総和を計算する．つまり個体数が block 数に，次元数が thread 数に対応するように実装する．ただしこれは基本実装であり，並列化粒度を変更すると個体の割り当て方が変わる．並列化粒度が異なる場合の個体の割り当て方を**図 4** に示す．図 4 のように，並列化粒度を細かくすると 2 つ以上の block で 1 つの個体の計算を行い，並列化粒度を粗くすると，1 つの block で 2 つ以上の個体の計算を行う．

5.2 CPU との処理時間の比較

GPU において基本実装での評価計算を行った場合と，CPU で評価計算を行った際の処理時間を比較した．Rastrigin 関数，Rosenbrock 関数それおぞれにおいて，次元数を 10，個体数を 400，2000，4000 とし，全個体に対する評価にかかる時間を 100 回測定し，その平均を算出した．またこれらの評価関数はテスト関数であり負荷が低いため，1 度の kernel 関数実行で 1000 回繰り返して計算し計算負荷を増加させている．

実験に用いた環境を**表 3**，**表 4** に示す．なおマシン 2 は 2 台の GTX460 を搭載しており，GPU を 1 台用いる場合と 2 台用いる場合について実験を行った．CPU のみを用いるプログラム，CPU と GPU を用いるプログラムのどちらも，評価以外の処理のコードは同一である．また CPU のみを用いるプログラムは，マシン 2 で実行した結果である．結果を**図 5**，**図 6** に示す．なお棒グラフは実行時間を示し，折れ線グラフは 1 秒当たりに評価する個体の数を示す．

比較の結果，それぞれの関数において GPU による高速化が確認できた．また基本的に個体数，つまり block 数が増加するにつれて GPU による評価計算の速度が上昇することが確認できた．kernel 関数実行に際して発生するオーバーヘッドは，個体数に関係無く変化しないと考えられる．それに対し kernel 関数の実行時間は，個体数と比例して線形に増える．そのため個体数÷実行時間として 1 秒当たりに評価する個体の数を算出すると，個体数が多い方がオーバーヘッドの影響が小さくなり，スループットが増加したと考えられる．

5.3 並列化粒度を変化させた際の処理時間の比較

CPU との処理時間の比較時と同一条件において，並列化粒度を変化させ，処理時間を 100 回計測し，その平均値の比較を行った．**図 7**，**図 8**，**図 9** にその結果を示す．

表 3 実験環境
Table 3 Systems used in experimentation

	マシン 1	マシン 2
CPU	Xeon W3530 2.8GHz	Core i5 2400 3.1GHz
Host Memory	6GB	8GB
GPU code Compiler	CUDA toolkit 3.2	
CPU code Compiler	gcc 4.4.5-15ubuntu1	

表 4 GPU のスペック
Table 4 Spec of the GPUs

	Tesla C2050	GeForce GTX 460
Memory	3GB	1GB
Memory Band Width	144GBs	115.2GBs
The number of SPs	448	336
The number of MPs	14	7

© 2011 Information Processing Society of Japan

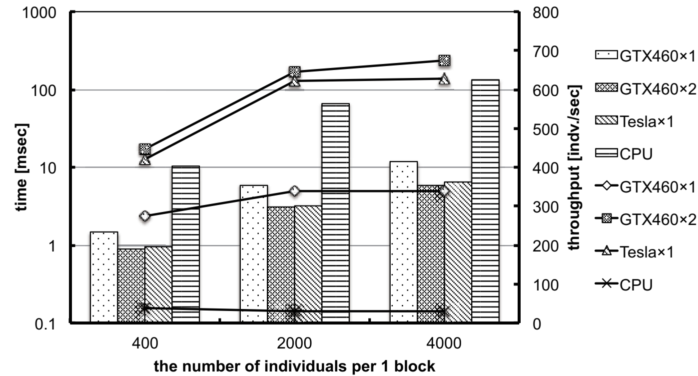

図 5 CPU と GPU の処理速度比較（Rastrigin）
Fig. 5 Execution speed comparison between CPU and GPU.(Rastrigin)

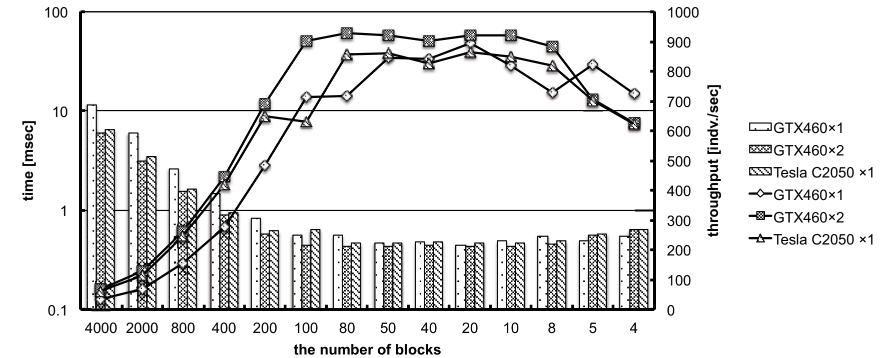

図 7 並列化粒度の変化と処理速度の関係（400 個体）
Fig. 7 Relationship between execution speed and changing parallel particle size.(400individuals)

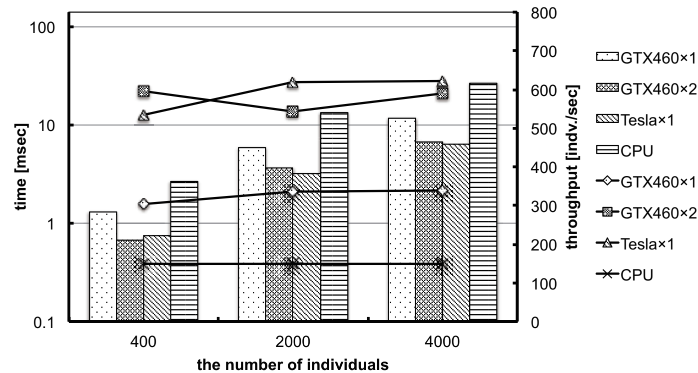

図 6 CPU と GPU の処理速度比較（Rosenbrock）
Fig. 6 Execution speed comparison between CPU and GPU.(Rosenbrock)

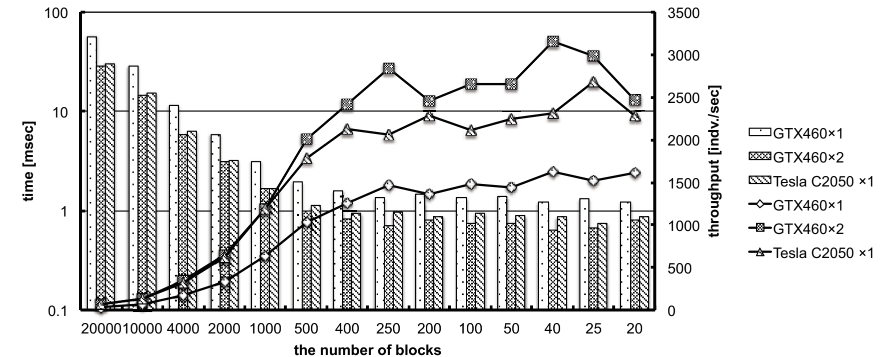

図 8 並列化粒度の変化と処理速度の関係（2000 個体）
Fig. 8 Relationship between execution speed and changing parallel particle size.(2000individuals)

ここで並列化粒度と総 thread 数，総 block 数の関係について述べる．各 thread は 1 つの個体の 1 次元分の計算を行うため，総 thread 数は個体数×次元数で一意に決まり，並列化粒度によって変化しない．一方で並列化粒度は，ここでは 1block に割り当てる個体数と考えることができる．つまり 1block 当たりの個体数が増えるほど並列化粒度は粗くなる．また総 block 数と 1block 当たりの個体数は反比例の関係にあるため，総 block 数が減るほど並列化粒度は粗くなる．なお基本実装である 1block 当たり 1 つの個体を割り当てるという方法では，1block に次元数分の thread を割り当てることになる．ここでは次元数は 10 であるため，この基本実装を基準として 1block 当たりの thread 数が 10 より多いと並列化粒度が粗く，10 より少ないと並列化粒度が細かいとする．

結果から，いずれの場合においても並列化粒度を細かくすると処理速度が低下し，粗くす

© 2011 Information Processing Society of Japan

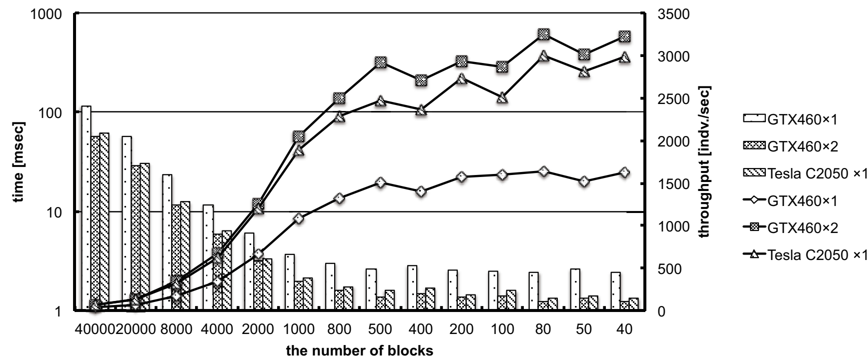

図 9 並列化粒度の変化と処理速度の関係（4000 個体）
Fig. 9 Relationship between execution speed and changing parallel particle size.(4000individuals)

ると処理速度が上昇することが確認できる．ただし，並列化粒度を粗くするほど処理速度が上昇することはなく，またそれぞれの GPU やその数によって性能がピークに達する block 数は異なる．

並列化粒度が粗くなるにつれて処理速度が上がった理由の 1 つとして，1block 当たりの thread 数が増加したことが挙げられる．基本実装においては 1block 当たりの thread 数は 10 であり，1warp 当たりの thread 数である 32 よりも少ない．そのため，10thread の計算を実行するために 32thread 分の計算を行なっており，効率が悪い．

また別の理由として，block 数と処理速度の関係が挙げられる．block は MP に割り当てられて計算を行うが，MP 数は表 4 にあるように Tesla C2050 で 14，GTX460 で 7 である．そのため block 数が大きいと何度も block の切り替えが発生し，コンテキストスイッチによる遅延が発生する．しかし block 数が小さいとメモリアクセスにかかるレイテンシを隠蔽することができなくなり，速度が低下する．そのため MP 数の数倍〜数十倍程度の block 数のとき，処理速度が上昇したと考えられる．

図 7，図 8 において block 数が 20 以下になると速度が低下しているのは，前述の block 数不足が原因と考えられる．特に個体数 2000 の場合は個体数 400 の場合よりも 1 つの block 当たりの計算量が大きいため，block 数 20 の時に GTX460 × 2，TeslaC2050 の処理速度が著しく低下していることがわかる．それに対して GTX460 × 1 の処理速度が低下していない理由は，MP 数が 14 の GTX460 × 2，TeslaC2050 では block 数が MP 数の約 1.4 倍であり不足しているのに対し，GTX460 × 1 では block 数が MP 数の約 3 倍であり，block 数が不足しなかったためと考えられる．

また全条件中で処理速度が最高になるのは，図 9 の個体数 4000 で block 数が 80 のときである．このときの thread 数は 500 であり，個体数 400 で block 数が 8 のときも thread 数が 500 である．しかし，block 数が 8 では block 数不足のために処理速度が低いことがわかる．

図 8 の個体数 2000 で block 数が 40 のときも thread 数が 500 である．block 数が十分大きいため，GTX460 × 1，× 2 使用時には個体数 2000 の条件下での最高速度が得られている．しかし CPU との処理速度比較で得られた結果から，同じ thread 数ならば block 数が多い方が処理速度が高くなると考えられるため，block 数 40 よりも 80 の時の方が処理速度が高くなると考えられる．

複数の GPU を用いた効果は，特に個体数増加により負荷が増加した時に得られた．また隣接する block 間の計算結果が必要な処理においては，複数の GPU を用いる際に GPU 間でデータを通信する必要があり，これが高速化の妨げとなる．しかし評価計算の処理は，データ並列性が高く GPU 間でのデータ通信が必要ないため，複数 GPU により得られる効果が大きかったと考えられる．

6. 議　　論

処理速度を測定した結果，フレームワークを利用して GPU を用いることで，CPU のみを用いる場合に比べて高速化されることが確認できた．さらに，この実装では性能のチューニングを行なっていないため，高速化の余地があると考えられる．ただしチューニングには前述の通り高度な技術を要する．そのためこのフレームワークを応用し，専門家が kernel 関数をチューニングする形を取ることで高速性を得ることができると考えられる．つまりフレームワークの利用者と kernel 関数の開発者はそれぞれの専門分野に集中することができ，開発効率が上昇すると考えられる．

現状のように利用者が kernel 関数を開発する形ではそのような利点を得られないが，ユーザが kernel 関数を記述することで GPU によるプログラミングを一般化することができると考えられる．

7. ま と め

本論文では，CUDA による並列計算を容易に利用できるようなフレームワークを提案し

© 2011 Information Processing Society of Japan

た．提案したフレームワークを用いれば，kernel 関数の記述と 2 つの関数呼び出しを行うことで GPU による並列処理を容易に利用可能である．

そしてこのフレームワークを利用し，GPU を用いて一部の計算を行う GA を実装し，CPU と処理時間比較を行った結果，高速化を確認できた．並列化粒度と処理時間についての評価も行い，thread 数，block 数が処理速度に密接に関係することが確認できた．

また専門家によるチューニングや GPU プログラミングの一般化など，提案したフレームワークについて議論した．

今後の展開として，処理速度向上のための非同期通信やホストメモリの種類を選択可能にするなどの機能追加が考えられる．

参考文献

1) 湯川英宣，平野敏行，西村康幸，佐藤文俊：GPU によるタンパク質高精度静電ポテンシャル計算の高速化，生産研究， Vol.61, No.2, pp.103–110 (2009).
2) 成瀬　彰，住元真司，久門耕一：GPGPU 上での流体アプリケーションの高速化手法 : 1GPU で姫野ベンチマーク 60GFLOPS 超 (高性能計算とアクセラレータ)，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]， Vol.2008, No.99, pp.49–54（オンライン），入手先⟨http://ci.nii.ac.jp/naid/110007082201/⟩ (2008-10-08).
3) 東　竜一，藤本典幸，萩原兼一：GPU の汎用計算環境 CUDA による主記憶上の大規模なテキストに対する高速な全文検索の検討 (アプリケーション高速化,「ハイパフォーマンスコンピューティングとアーキテクチャの評価」に関する北海道ワークショップ (HOKKE-2008))，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]，Vol. 2008, No. 19, pp. 139–144（オンライン），入手先⟨http://ci.nii.ac.jp/naid/110006828688/⟩ (2008-03-05).
4) : Top500 Supercomputing Sites, http://www.top500.org/.
5) Belleman, R.G., Geldof, P.M. and Zwart, S. F.P.: High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA, *New Astronomy*, Vol.13, pp.103–112 (2008).
6) graphics powerfor MDsimulations, H.: http://www-old.amolf.nl/vanmeel/mdgpu/about.html.
7) 田中裕也，吉見真聡，三木光範：GPU 用自動並列化コンパイラを用いた Fortran プログラムの高速化手法の提案，第 10 回情報科学技術フォーラム (FIT2011) 講演論文集，Vol.1, pp.341–342 (2011).
8) XcalableMP: http://www.xcalablemp.org/.
9) OpenMP: http://openmp.org/wp/.
10) T.Hiroyasu, R.Yamanaka, M.Yoshimi and M.Miki: A Framework for Genetic Algorithms in Parallel Environments, PDPTA'11（The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications），pp.751–756 (2011).

© 2011 Information Processing Society of Japan