Vol.2012-ARC-202 No.32
Vol.2012-HPC-137 No.32
2012/12/14

数分割問題に対するGPUを用いた並列化

佐藤 友哉[1]　藤本 典幸[1]

概要：NP 困難な組み合わせ最適化問題である数分割問題（number partitioning problem）は公開鍵暗号などに応用を持つ．この問題に対する探索木の幅優先探索とビーム探索に基づく Pedroso らのアルゴリズムを CUDA アーキテクチャに基づく GPU を用いて並列化する手法を提案する．GeForce GTX 580 GPU と 2.67GHz Intel Core 2 Duo CPU E7300 を用いて評価実験を行ったところ，提案手法に基づく CUDA C プログラムは，Pedroso らの逐次 Python プログラムに比べて最大約 323 倍高速であった．また Python プログラムを参考に作成した C 言語プログラムに比べて最大約 12.2 倍高速であった．

Parallelizing the Number Partitioning Problem for GPUs

Tomoya Sato[1]　Noriyuki Fujimoto[1]

Abstract: The number partitioning problem is an NP-hard problem that can be applied to public-key cryptography and so on. For the problem, Pedroso et al. presented an algorithm based on breadth first search and beam search of a search tree. In this paper, we propose a method to parallelize Pedroso et al.'s algorithm for GPUs based on the CUDA architecture. The experimental results on an NVIDIA GeForce GTX 580 GPU and a 2.67GHz Intel Core 2 Duo CPU E7300 show that the CUDA C program based on the proposed method is up to 323 times faster than the serial Python program published by Pedroso et al. Another experiment show that the proposed CUDA C program is up to 12.2 times faster than our serial C program ported from the Python program.

1. はじめに

数分割問題（number partitioning problem）とは与えられた正整数の集合に対して，一方の集合の要素の総和ともう一方の集合の要素の総和との差の絶対値を最小化する分割を見つける問題である．数分割問題は NP 困難な組み合わせ最適化問題であり [2]，公開鍵暗号などに応用される．この問題の近似解を求める発見的アルゴリズムで，探索木の幅優先探索とビーム探索に基づいたもの [9] が Pedroso らによって提案されている．このアルゴリズムは厳密解を求めるアルゴリズムに比べて短い時間で近似解を出力することができ，他の発見的アルゴリズムに比べてより良い解を出力することが多い．しかしながら，依然として計算時間は高速化を要する．この論文では Pedroso らのアルゴリズムにおける幅優先探索時の同レベルのノード間の並列性と，各ノードでの一部処理の並列性に注目して，GPU を用いた並列処理によって高速化することを考えた．これらのアイデアを CUDA で実装した．

以降の論文の構成は次の通りである．第 2 章で数分割問題の定義と Pedroso らのアルゴリズムについて，第 3 章で提案手法について，第 4 章で評価実験について説明する．そして第 5 章でまとめと今後の課題について述べる．スペースの制限のため，本論文では GPU のアーキテクチャおよびプログラミングの概要については述べない．これらに不慣れな読者は文献 [1], [3], [6], [7], [11] を参照されたい．

1　大阪府立大学 大学院理学系研究科
Graduate School of Science, Osaka Prefecture University

2. 準備

2.1 数分割問題

数分割問題は形式的には次のように定義できる．N 個の正整数の集合 A=$\{a_1, a_2, \cdots, a_N\}$ に対して，

$$E_A(P) = \left| \sum_{i \in P} a_i - \sum_{i \notin P} a_i \right|$$

を A の分割 $P \subseteq \{1, \cdots, N\}$ に対する相異（discrepancy）

© 2012 Information Processing Society of Japan

という．数分割問題は，A が与えられたとき $E_A(P)$ を最小化する P をひとつ求める問題である．相異 $E_A(P)$ が 0 または 1 となるような分割 P を A の完全分割（perfect partition）という．完全分割は最適な分割である．

2.2 Karmarkar and Karp（KK）差分法

数分割問題に対する，これまでに知られている最善の多項式時間ヒューリスティックは，次に示す Karmarkar と Karp の差分法（differencing method）[5] である．

KK(A)
(1) n 個の頂点 $v_1, \cdots, v_n$ を用意，任意の A の要素 a_i について $key(v_i) = a_i$
(2) $T = \{\}$ //枝集合
(3) while 頂点が 1 つになるまで
(4) 各頂点の中で最も key が大きい 2 頂点を取り出す，大きい方を u，小さい方を v とする
(5) T に枝 (u, v) を追加
(6) 頂点 v を取り除く
(7) $key(u) = key(u) - key(v)$
(8) return 残った頂点の key と T

最後に残った頂点の key が KK 差分法が導く近似解の相異で，T からその分割を構成できる．2 つの要素を差で置き換えることはそれらが異なる集合の要素に分割されることを意味する．

2.3 総当たり差分法（complete differencing method）

KK 差分法では 2 つの最も大きな数をそれらの差の絶対値によって置き換えていたが，和による置き換えも行うようにすると，2 つの要素を同じ集合に分割するという選択肢も考えることになり，全ての分割を探索することになる．そうすると**図 1** のような 2 分木が考えられる．図 1 は {8, 7, 6, 5, 4} に対して総当たり差分法を適用した例である．この 2 分木の各葉はある分割の相異に一対一で対応していて，その分割は根から各葉への路の枝から構成できる．

この探索木の一部は以下の性質により，枝刈りできる．
(1)KK 差分法は要素数 4 以下の入力に対して厳密な解を出力する．よってノードの要素数が 4 になった場合，それ以上分枝する必要がない．
(2) もし入力の集合の最も大きな数とそれ以外の数の和との差の絶対値が 0 か 1 なら完全分割がみつかったのでアルゴリズムを終了する．
(3) もし入力の集合の最も大きな数とそれ以外の数の和との差が 1 以上なら，最も大きな数を片方の分割に，もう片方の分割にそれ以外の数を配置するのが最善の解なので，この副木の分枝は止められる．

2.4 Pedroso らの手法

Pedroso らの手法は総当たり差分法で作られる探索木に対して，一つのレベルで探索するノードの数をパラメータ α で制限する幅優先探索を行う．幅優先探索は一つのレベルで探索するノード数を制限することでパラメータ化できる．これは Pinedo によって提案されたスケジューリング問題に対するビーム探索 [10] の変形である．ビーム探索とは，指定した優先度に基づいて各レベルで探索するノード数をビーム幅 α 個のみに制限して探索する方法である．Pedroso らの実装では，各レベルで KK 差分法からどれだけ遠ざかっているか，つまり根からそのノードまでに和で置き換える処理を行った回数でノードの集合をソートする．そして和で置き換える処理を行った回数が少ない α 個のノードを選び，残りは無視する．以下はこの手法のアルゴリズムである．

BFS(A, α, T)
(1) $E* = KK(A)$
(2) $B = \{A\}$ //要素はノード
(3) while $B \neq \{\}$ and CPU time used $< T$
(4) $C = \{\}$ //ここに入らないノードは枝刈りする
(5) for B の各要素 β について
(6) $b = \beta$ の最大の要素
(7) $r = \beta$ の b を除いた各要素の総和
(8) if $|b - r| = 0$ or $|b - r| = 1$
(9) return $|b - r|$ //$\{b\}, \beta - \{b\}$ と分ける分割が完全分割
(10) if $b \geq r$ //$\{b\}, \beta - \{b\}$ と分ける分割が β のとり得る分割で相異が最善の値
(11) if $b - r \leq E*$ //上界を更新する
(12) $E* = b - r$
(13) else
(14) $C = C \cup \beta$
(15) $B = \{\}$
(16) $C = \{C$ のノードの中で要素を和で置き換えた回数の少ないもの α 個 $\}$
(17) for C の各要素 β について
(18) $b_1 = \beta$ の最大の要素
(19) $b_2 = \beta$ の 2 番目に大きな要素
(20) $X = \{b_1 - b_2\} \cup \beta - \{b_1, b_2\}$
(21) $Y = \{b_1 + b_2\} \cup \beta - \{b_1, b_2\}$
(22) $B = B \cup \{X, Y\}$
(23) $E = KK(Y)$
(24) if $E < E*$ //上界を更新する
(25) $E* = E$
(26) return $E*$

16 行目のノードの要素を和で置き換えた回数とは，そのノードが 21 行目を実行された回数のことである．また，和で置き換えた回数が同じものは KK 差分法の解が良い方が優先される．

© 2012 Information Processing Society of Japan

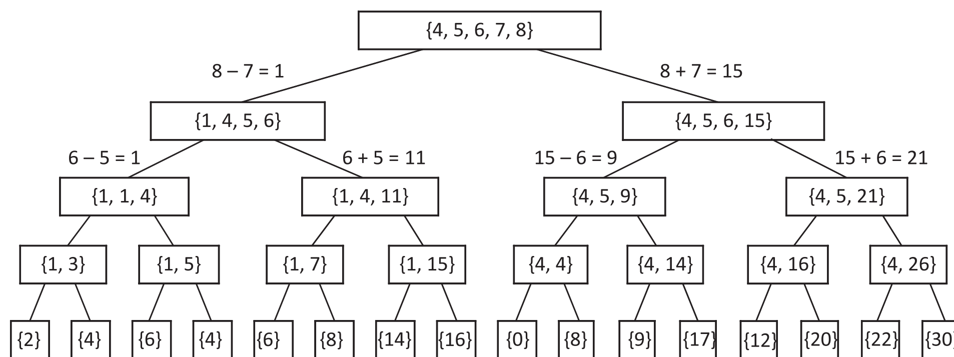

図 1 {8,7,6,5,4} に対する総当たり差分法

Fig. 1 The complete differencing method for {8,7,6,5,4}.

3. 提案手法

3.1 概要

同レベルのノード間で行う処理については，上界の更新を除いて互いの結果に依存しないので，並列に処理できる．上界の更新については，あるレベルでいずれかのノードがひとつ前のレベルまでの上界より良い解を見つけた場合，そのレベルのノードの処理がすべて終わった後に逐次的に上界を更新する．複数の上界を更新する候補が見つかったときは，それらの中で最も良いものを新たな上界とする．次に行う枝刈りの処理は逐次的に実行した．探索するノードを制限するためにノードをソートする必要があるが，ソートは CUDA 用の並列アルゴリズムライブラリの Thrust[4] を使用した．そのあとに行う分枝処理（2.4 節のアルゴリズムの 17-25 行目）も左への分枝と右への分枝をそれぞれ並列に実装した．さらに，GPU にはスレッドブロック間の並列性とスレッドブロック内のスレッド間の並列性の 2 段階の並列性があるので，探索のときに，各ノードで行う処理のなかで KK 差分法の処理の一部を並列に行った．

3.2 KK 差分法の並列化

各ノードで行う計算において最も計算時間を必要とするのは，KK 差分法による解の計算である．このヒューリスティックの実装はまず shared メモリ上に入力の集合の各データを昇順を保ったまま転送する．次に，一番後ろの 2 要素，つまり最も大きな 2 要素を取り出してそれらの差の絶対値を元の配列に昇順を保ったまま挿入する．以上の処理を要素数が 1 になるまで繰り返す．そして，最後に残った値が KK 差分法が出力する相異となる．

これらの処理のうち，最も大きな 2 要素の差の絶対値 d をもとの配列に昇順を保ったまま挿入する処理を並列化した．その手順は，次の 2 つに分けることができる．

手順 1．まず昇順を保つためにはどこに d を挿入すればよいかを見つける．その場所を q 番目とする．

手順 2．q 番目に値を挿入するためにその場所より後ろの要素を 1 つ後ろへずらす．そして，q 番目に d を挿入する．

手順 1 を並列処理で実現する手法を**図 2** に示す．計算する集合の要素が配列 $A[0], A[1], \cdots, A[n-1]$ に昇順に格納されているとすると，初めに各スレッドで $d = A[n-1]-A[n-2]$ を計算する．そして q は要素数 n で初期化しておく．次に元の配列のどこに挿入すれば昇順を保つことができるか調べる．線形探索を複数のスレッドで協力して各要素と d を比較する．$A[i] < d$ となる i が見つかると探索を終了，その i を q に記憶する．いくつかのスレッドが $A[i] < d$ を見つけた時は，アトミック関数 atomicMin を使うことで，それらの中で最小の i を q に格納する．アトミック関数とは，複数のスレッドが同じメモリに対して処理を行うとき，矛盾が生じないように排他制御を行う関数である．atomicMin は第 1 引数のアドレスの値と，第 2 引数の値を比較して，第 2 引数の方が小さければ，第 2 引数の値で第 1 引数のアドレスの値を更新する関数である．

手順 2 を並列処理で実現する手法を**図 3** に示す．この手法では，各スレッドが協力して $A[q]$ より後ろの要素を 1 つずつ後ろへずらすのだが，これを各スレッドが一斉に $A[j+1] = A[j] (q \leq j < n-2)$ を実行すると，異なるウォープに属するスレッドは命令レベルでは同期していないので，配列の値が不定になってしまう．よって $A[j]$ の値をまずレジスタに代入して，スレッド間の同期をとってからレジスタの内容を $A[j+1]$ に代入する．そして $A[q]$ に 1 つのスレッドが d を代入すれば昇順を保ったままの挿入が完了する．

この処理に割り当てるスレッドの数について検討する．**図 4** は 1 つのスレッドブロックを 1 つノードに割り当てて，スレッド数だけを変えて，問題を解いたときに KK 差分法を実行するのにかかった平均時間を，1 スレッドでの平均実行時間を 1 とした比率で表している．図 4 でスレッドをある程度以上多くしても高速化につながらないのは，KK 差分法の処理はだんだん調べる要素の数が減っていくので，スレッドが多すぎると，無駄な演算が増えてかえっ

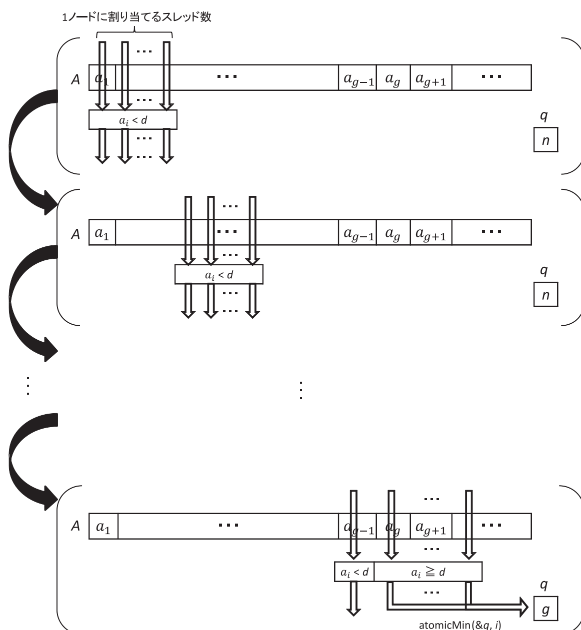

図 2 d を挿入する場所の並列探索
Fig. 2 Parallel search of the position at which d should be inserted.

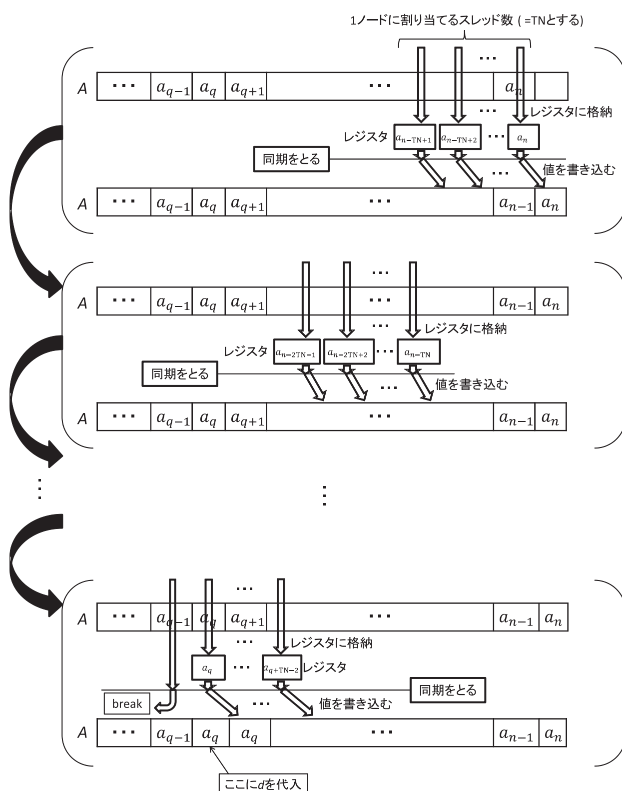

図 3 q 以降を 1 つずらして d の並列挿入
Fig. 3 Parallel insertion of d.

て遅くなったためと考えられる．

3.3 幅優先探索の並列化

各レベルのノード間で並列に処理できる部分は枝刈りの判定（2.4 節のアルゴリズム 6-12 行目:上界の更新は除く），分枝（18-22 行目），KK 差分法（23 行目），の 3 つである．枝刈りの判定は各ノードでの処理は 1 スレッドで十分である（集合の各要素の総和は実装では最初に求めておいて，分枝するときに変化した分だけ更新する）．このため、1 ノードに 1 スレッドを割り当てて並列化を行う．

分枝で並列に処理できるところは KK 差分法のアルゴリズムの 4-7 行目の処理とほぼ同じなので，KK 差分法と同じ割り当て方が良い．

3.2 節の結果より KK 差分法を計算するときは 1 つのノードに割り当てるスレッド数は 8 スレッドか 16 スレッドが良いと考えられる．しかし GeForce GTX 580 では，1 つのマルチプロセッサで同時に起動できるスレッドブロックの数は 8 個で，1 つのスレッドブロックで同時に起動できるスレッド数は 1536 スレッドなので，このまま 1 つのスレッドブロックに 1 つのノードを割り当てると，1 つのマルチプロセッサで同時に起動するスレッド数は

8 ブロック× 16（または 8）スレッド
=128（または 64）スレッド

となり，GPU の処理能力を持て余すことになってしまう．したがって，1 つのスレッドブロックに複数のノードを割り当てることを考える．そうすると，1 つのスレッドブロックに 1 つのノードを割り当てるときは同時に 8 つのノードを処理していたのが，1 つのスレッドブロックに割り当てたノード数倍のノードを同時に処理できるようになる．このとき注意すべきことは同時に扱える shared メモリのサイズにも制限があるということである．GeForce GTX 580 では同時に扱える shared メモリは 48KB で，これを超えないように同時に起動するスレッドブロックの数が決定される．したがって，8 つのスレッドブロックが同時に起動できるように，1 つのスレッドブロックに割り当てるノード数はスレッド数の制限と shared メモリサイズの制限を満たした中で最大のノード数を割り当てるようにする．**図 5** は横軸を実行時間として，1 つのノードに 16 スレッド割り当てた場合の KK 差分法の実行時間に対する 8 スレッドを割り当てた場合の実行時間の比率である．図 5 より 8

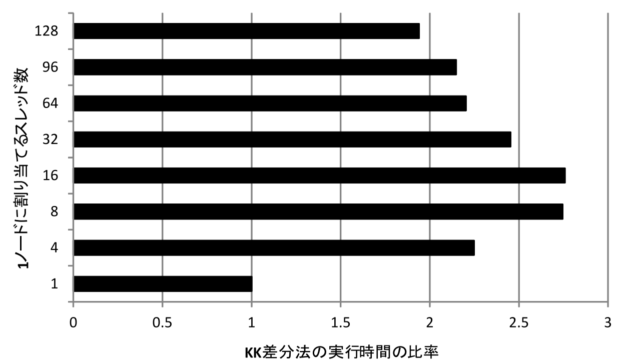

図 4 1 ノードに割り当てるスレッド数と KK 差分法の総実行時間
Fig. 4 The number of threads per node and the total execution time of KK differencing method.

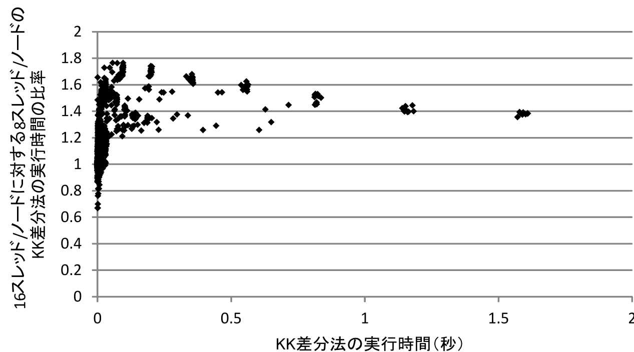

図 5 KK 差分法の実行時間に対する 16 スレッド/ノードと 8 スレッド/ノードの KK 差分法の実行時間の比率

Fig. 5 Speedup ratio of 16 threads per node to 8 threads per node for KK differencing method.

スレッドの方が高速に KK 差分法を処理している．これは 1 つのノードに割り当てるスレッド数が 8 スレッドの方が 1 つのノードの処理は若干遅いが，それ以上に，1 つのスレッドブロックが同時に処理するノードの数が 16 スレッドに比べて増えることによる処理全体の高速化の方が効果的であったためと考えられる．

4. 評価実験

CPU は 2.67GHz Intel Core2 Duo E7300，GPU は NVIDIA GeForce GTX 580，OS は Windows 7 Professional 64bit，開発環境は CUDA4.2 および Visual Studio 2008 Professional を用いた．CPU のプログラムは Pedroso 自身が Python 言語で実装しているものと，それを参考に作成した C 言語で実装したものを使用した．評価実験に使ったインスタンスと Python 言語のプログラムは Pedroso 自身が Web サイト [8] に公開している．Python のバージョンは 2.7 を使用した．評価実験に使ったインスタンスは D instance と呼ばれるもので，要素数 15〜105 で，要素の桁数が 10 進数で 10，12，14 桁である．これらのインスタンスには多くの完全分割を持つ見込みがあるもの（easy instance）と，完全分割がある見込みが低いインスタンス（hard instance）が混在している．1 つのレベルで探索するノード数を制限するパラメータ α は 10, 100, 1000, 10000, 100000 で試した．

図 6 は Python プログラムの実行時間に対する提案 GPU プログラムの速度向上率である．CPU での実行時間が 60 秒以上の問題に対しては，どれも 113 倍以上の速度向上率を示していて，最大約 323 倍の速度向上が見られた．**図 7** は C プログラムでの実行時間に対する提案 GPU プログラムの速度向上率で，最大約 12.2 倍の速度向上が見られた．ごく短い時間で終了する問題に対してあまり良い結果が出ていない原因として次の 3 つが考えられる．1 つ目は GPU で問題を処理するためには CPU と GPU 間でデータ転送の必要があり，その転送時間がネックになってしまっていることが挙げられる．**図 8** は GPU での実行時間に対するデータ転送時間の占める割合で，実行時間がごく短いものは，その実行時間のほとんどがデータ転送に使われていることがわかる．原因の 2 つ目は実行時間が短い問題にはパラメータ α の値が小さいものが多く，同レベルのノード間の並列性が低いことである．原因の 3 つ目は早い段階で完全分割が見つかり，ノードの数が増える前に終了した場合である．この場合も同レベルのノード間の並列性が低い．

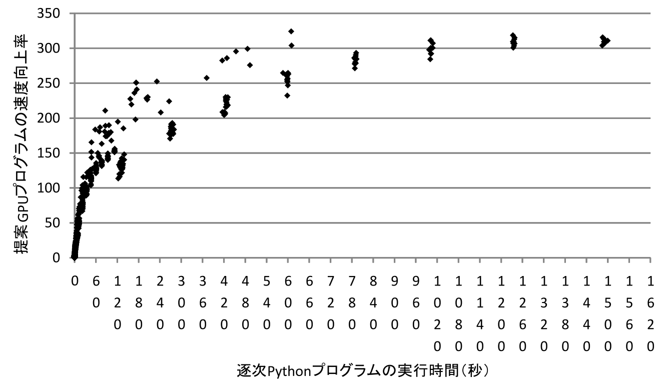

図 6 Python プログラムに対する提案 GPU プログラムの速度向上率

Fig. 6 Speedup ratio of our GPU program to the corresponding Python program.

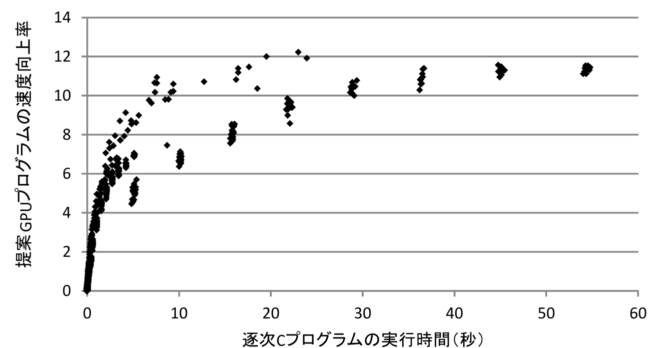

図 7 C プログラムに対する提案 GPU プログラムの速度向上率

Fig. 7 Speedup ratio of our GPU program to the corresponding C program.

5. まとめ

本論文では，数分割問題を計算する Pedroso らのアルゴリズムを GPU で並列に処理する手法を提案した．提案手法は Python プログラムに対して最大約 323 倍，C プログラムに対して最大約 12.2 倍高速で，実行時間のごく短い問題を除いて安定して高速であった．

今回の評価実験で使った数分割問題の入力の集合の各要素は 64bit で表現できる整数だった．数分割問題に対しては，要素の最大値が a_{max} ならば $O(Na_{max})$ 時間で解く CPU アルゴリズムが知られている．つまり a_{max} が N の多項式で抑えられるなら N の多項式時間で解ける．現在 1024bit までの多倍長整数を要素に持つ入力を計算する

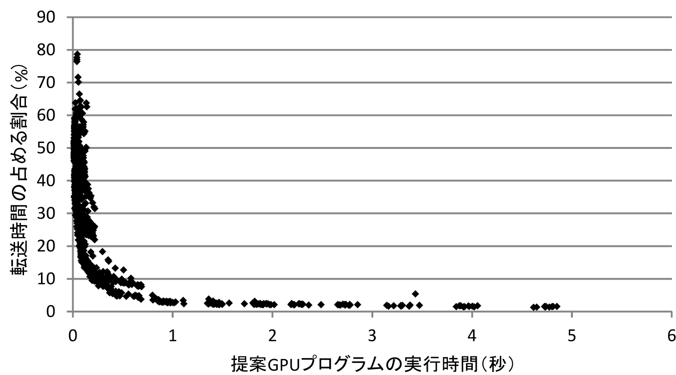

図 8 提案 GPU プログラムの実行時間に対する CPU と GPU 間のデータ転送時間の占める割合

Fig. 8 Ratio of data transfer time between CPU and GPU to the whole execution time of our GPU program.

GPU プログラムの実装はできているが，Python プログラムに対する速度向上率は高々 35 倍程度で本論文の結果とギャップがある．このギャップを埋めることが今後の課題となる．

参考文献

[1] 青木尊之, 額田彰: はじめての CUDA プログラミング, 工学社 (2009).

[2] M. R. Garey and D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W H Freeman (1979).

[3] M. Garland and D. B. Kirk: *Understanding Throughput-Oriented Architectures*, Comm. ACM, 53(11), 58–66 (2010).

[4] J. Hoberock and N. Bell: *Thrust*(online), 入手先 ⟨http://code.google.com/p/thrust/⟩ (2012.01.11).

[5] N. Karmarkar and R. M. Karp: *The differencing method of set partitioning*, Technical report UCB/CSD 82/113, University of California. Berkele, Computer Science Division (1982).

[6] D. B. Kirk and W. W. Hwu: *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann (2010).

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym: *NVIDIA Tesla: A Unified Graphics and Computing Architecture*, IEEE Micro, 28(2), 39–55 (2008).

[8] J. P. Pedroso: *Tree Search for Number Partitioning: An Implementation in the Python language*, Internet repository, version 1.0(online), 入手先 ⟨http://www.dcc.fc.up.pt/∼jpp/partition⟩ (2011.07.05).

[9] J. P. Pedroso and M. Kubo: *Heuristics and exact methods for number partitioning*, European Journal of Operational Research, 202, 73–81(2010).

[10] M. Pinedo: *Scheduling: Theory, Algorithms, and Systems*, Prentice-Hall,Englewood Cliffs(1995).

[11] J. Sanders and E. Kandrot: *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional (2010).

© 2012 Information Processing Society of Japan