Vol.2013-HPC-141 No.2
2013/9/30

GPUを用いた分枝限定法における メモリ参照効率を高めるための配列パッキング手法

重岡 謙太朗[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，GPU（Graphics Processing Unit）における分枝限定法の高速化を目的として，メモリ参照効率を高めるための配列パッキング手法を提案する．提案手法は，CPU の替わりに GPU が配列を詰め込むことにより，CPU・GPU 間のデータ転送を最小限に抑える．詰め込みにより配列を密に維持でき，分枝限定操作におけるメモリ参照効率を高める．GPU 上の詰め込みは，よく知られた並列 prefix sums アルゴリズムに基づく．さらに，多くの対象問題において配列要素の並び順を詰め込みの前後で維持する必要がないことに着目し，in-place 型の詰め込みを実現する．これにより，入力配列と出力配列を区別する（非 in-place な）単純手法と比較して，およそ 2 倍の部分問題を GPU 上で並列処理できる．ナップザック問題を用いた実験では，CPU 上で配列を詰め込む既存手法と比較して 1.15 倍の速度向上を得た．また，in-place 版は非 in-place 版と比べて扱える部分問題の数を増やせたが，性能は 8.8%減少した．

キーワード：分枝限定法，配列パッキング，高速化，CUDA，GPU

An Array Packing Method for Increasing Memory Reference Efficiency of the Branch and Bound Method on a GPU

Kentarou Shigeoka[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: In this paper, we propose an array packing method for increasing memory reference efficiency on a graphics processing unit (GPU), aiming at accelerating the branch and bound method. Our method minimizes data transfer between the CPU and GPU by packing arrays on the GPU instead of the CPU. This array packing procedure generates dense matrices, so that it increases memory reference efficiency during processing branching and bounding operations. Our GPU-based packing method employs a well-known parallel prefix sums algorithm. Furthermore, we realize in-place array packing by allowing array elements to change their order after the packing procedure. This doubles the number of parallel subproblems that can be processed on a GPU, as compared to a not-in-place method, which distinguishes input arrays from output arrays. In experiments using a knapsack problem, our method achieves a speedup of 1.15 over a previous method that packs arrays on a CPU. As compared to a not-in-place version, our in-place version increases the number of parallel subproblems but decreases performance by 8.8%.

Keywords: Branch and bound method, array packing, acceleration, CUDA, GPU

1. はじめに

分枝限定法とは，最適解化問題の最適解を求める手法である．この手法は，分枝操作および限定操作を繰り返すことにより，探索空間の範囲を削減する．分枝操作では，1つの問題を複数の小さな部分問題に分割する．限定操作では，これまでの部分問題における最良の下限値および上限値を求め，これらの値を基に，最適解を導出しえない部分問題を削除（枝刈り）する．削除を免れた（活性）部分問題に対し，さらに分枝操作を適用していく．例えば，ナップザック問題では，ナップザックに詰める物品の組合せに

[1] 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

関して探索空間の範囲を狭めていく．一般に，分枝限定法が有用となる問題は，部分問題の数が多く計算時間が長いため，並列処理による高速化が求められている．

分枝限定法の高速化を目的として，Lalami ら [1] は GPU 上で分枝限定操作を高速化する手法を提案している．この既存手法は，部分問題の状態（活性・非活性）を管理するための配列を用意し，配列の各要素を 1 つの部分問題に対応させる．枝刈りに起因して配列が疎になることを防ぐために，探索木の深さを増やすたびに CPU 上で配列を密に詰め込む（配列パッキング）．具体的には，枝刈りされた要素を配列の後方に移動させることにより，分枝操作の対象となる要素を連続領域に格納する．GPU は連続領域への読み書きを得意としているため，詰め込みの結果，次の深さの分枝限定操作におけるメモリ参照効率を高めることができる．

しかし，既存手法は CPU が詰め込みの移動先計算を担当しているため，探索木の深さを増やすたびに CPU・GPU 間のデータ転送を必要とする．したがって，性能ボトルネックがデータ転送にある場合，加速を達成できない恐れがある．また，GPU が詰め込みを担当する場合，GPU の得意とするメモリ参照パターンのみを用いて，効率のよい詰め込み処理を実現できるか否かは定かでない．

そこで，本稿では GPU における分枝限定法の高速化を目的として，メモリ参照効率を高めるための配列パッキング手法を提案する．提案手法は，GPU が配列を詰め込むことにより，CPU・GPU 間のデータ転送を最小限に抑える．また，詰め込みにより配列を密に維持でき，分枝限定操作におけるメモリ参照効率を高める．GPU 上の詰め込みは，よく知られた並列 prefix sums アルゴリズムに基づいており，Thrust ライブラリ [2] の実装による．さらに，多くの対象問題において配列要素の並び順を詰め込みの前後で維持する必要がないことに着目し，in-place 型の詰め込みを実現する．これにより，入力配列と出力配列を区別する（非 in-place な）単純手法と比較して，およそ 2 倍の部分問題を GPU 上で並列処理できる．

ナップザック問題を対象とした実験では，提案手法は既存手法と比べて，速度向上が最大 1.15 倍であった．また，単純手法と比べて，ビデオメモリ容量が不足することに起因する実行失敗を 3/5 に削減できた．ただし，in-place 詰め込みのための前処理が 原因でメモリの読み書き量が最大 50%増大し，単純手法よりも 8.1%低速であった．このように，詰め込み処理の in-place 性と実行時間にトレードオフの関係があった．

以降では，まず 2 節で既存方式を含む予備知識についてまとめる．次に，3 節で提案手法を示す．4 節で評価実験の結果を示し，5 節で本稿をまとめる．

2. GPU を用いた分枝限定法

本節では，まず分枝限定法における分枝限定操作の概要を説明する．次に，既存の GPU による高速化手法を紹介し，最後に，GPU による高速化において解決すべき問題点を述べる．

2.1 分枝限定操作

0-1 ナップザック問題を例に分枝限定法を説明する．n 個の物品があり，この物品を容量 b のナップザックに詰める．i $(1 \le i \le n)$ 番目の物品の重量および価値をそれぞれ w_i および p_i とし，x_i は i 番目の物品をナップザックに入れる場合は 0，入れない場合は 1 を格納する変数とする．このとき，ナップザック問題は以下のように表せる．

$$\begin{cases} \sum_{i=1}^{n} p_i x_i & \rightarrow \quad \max \\ \sum_{i=1}^{n} w_i x_i & \le \quad b, \qquad x_i = 0, 1. \end{cases}$$

この n 変数問題を，ある変数 x_i を 0 もしくは 1 に固定することにより，2 個の $n-1$ 変数問題（部分問題）に分割する．さらに，生成された部分問題に分枝操作を繰り返し適用し，可能な限り分解を進める．この処理を分枝操作と呼ぶ．

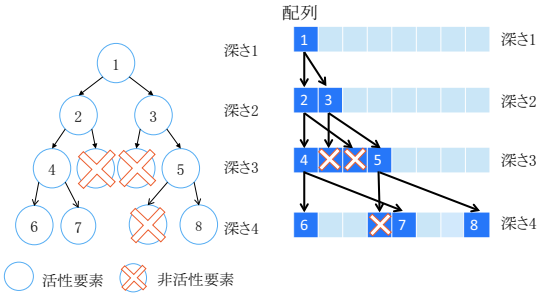
分枝操作のみである場合，単純な列挙法となるため，全ての部分問題の一部分のみを処理する工夫をする．この工夫を限定操作と呼ぶ．限定操作は，ある部分問題 P_i が元の問題 P_0 の最適解を得られないことが結論付けられれば，P_i を削除する．限定操作は，下限値によるものおよび優越関係を用いるものが代表的である [3]．

この限定操作を適用することにより，実際に生成される部分問題の個数を小さくできる．その様子を**図 1**(a) に示す．分枝限定処理は，元問題 P_0 を根とした探索木として表わせる．この際，最適解を得られる可能性がある部分問題を活性要素と呼ぶ．また，限定処理により削除された問題を非活性要素と呼ぶ．

GPU を用いて分枝限定法を高速化する場合，GPU は多数の活性要素を並列に分枝限定操作をする．これらの操作において，部分問題の状態を示す上限値および下限値などのデータは，種類ごとに GPU メモリ上に配列を用意し，活性要素を格納する．この場合，一つの活性要素 P_i のデータは，全ての配列について同一のインデックス i に格納する．

問題点は，分枝限定操作を繰り返すことにより，配列の活性要素が疎となることである（図 1(b)）．なぜなら，GPU は連続した 128 バイトのメモリデータを一度に参照するため，非連続なデータ参照によりメモリ参照効率が低下し，実行時間が増大するためである．

ラベル配列およびインデックス配列を基にして，活性要



(a) 探索木　　(b) データ構造

図 1　分枝限定法

素を連続領域に格納することにより配列を密にする．分枝限定操作以外に，この配列パッキングの操作を加えることにより，GPU におけるメモリ参照効率を高め，分枝限定法を高速化できる．

ラベル配列は，各配列の要素における活性要素の有無を示す．活性要素の位置には 1，非活性要素の位置は 0 が格納された配列である．次に，インデックス配列はラベル配列内における活性要素の prefix sums を格納した配列である．活性要素の位置には，先頭からの活性要素のインデックスが示される．

2.2　既存の高速化手法

分枝限定法は部分問題間にデータ依存がないため，複数の部分問題を並列処理できる．計算グリッドおよびクラスタを用いて分枝限定法を高速化する研究がある．マスタ・ワーカ方式による分枝限定法は，ワーカにおいてある部分問題の分枝限定処理を実行する．そのため，マスタ・ワーカ方式による分枝限定法は，各ワーカの分枝限定処理を高速化する必要がある．その単一ノード内の計算資源として近年 GPU が注目を集めている．

例えば，Kurowski ら [4] は，分枝限定法により巡回セールス問題を高速化するために，GPU による分散計算フレームワークを実装している．下界値を用いた限定操作をマスタ・ワーカ方式により高速化している．

さらに，GPU を用いて組合せ最適化問題を分枝限定法により高速化する研究がある．Lalami ら [1], [5] は，GPU を用いてナップザック問題を高速化している．彼らの手法は，処理する要素の規模に従い，GPU および GPU 処理を切り替える．後に，彼らはこの研究を発展させ，配列パッキングに必要な CPU・GPU 間データ転送を，GPU によるデータ移動の手法により削減している．ただし，インデックス配列の計算は CPU により処理しているため，詰め込み処理における CPU・GPU 間データ転送を排除できない．

同様に，Chakroun らはフローショップ問題を高速化している．彼らは CPU 上で分枝操作を処理しており，限定操作のみを GPU により高速化する．また，Carneiro[6] らは，幅優先探索および深さ優先探索を組み合わせて探索することにより，対称巡回セールス問題を分枝限定法により高速化している．これらの研究は，Lalami らの研究と同様に，GPU により限定処理した活性要素を，CPU もしくは GPU により詰め込む必要がある．

また，Zhang[7], [8] らは，メモリデータの置換およびスレッドの動的なデータ割り当てにより，制御フローおよびメモリ参照の動的不規則性を排除している．この際，動的なデータ割り当てのオーバーヘッドは，CPU・GPU パイプライン処理により排除している．しかし，本研究では，分枝限定処理の反復において，ひとつ前の処理結果が次の処理に必要となるため，パイプライン処理によるオーバーヘッドを排除できない．

さらに，GeForce GTX680 などの GPU は 2GB のビデオメモリしか持たないため，メモリ不足により解けない問題が存在する可能性がある．

3.　提案手法

本研究では，GPU の潜在的な性能を最大限活用するため，多数の要素を並列に解ける幅優先探索を対象とする．提案手法は，前節にて述べた問題点に対して，以下の項目を実現する．

(1) GPU による配列パッキングにより，CPU・GPU 間データ転送を削減
(2) in-place 型の詰め込みにより，並列処理可能な要素数を増大

まず，3.1 節では (1) の解決手法を示し，3.2 節において (2) を実現する詰め込み手法について説明する．

3.1　GPU による配列パッキング

提案手法は，GPU が詰め込みの移動先を計算することにより，CPU・GPU 間データ転送を最小限にする．GPU を用いた詰め込みは，よく知られた並列 prefix sums を用いたコンパクション [9] に基づく．**図 2**(a) にアルゴリズムの概要を示す．ラベル配列の prefix sums を求め，結果をインデックス配列に格納する．作成したインデックス配列が示すインデックスを基に，先頭から活性要素を出力領域に格納する．この手法を単純手法と呼ぶ．

単純手法は入力および出力を別々に配列に格納する必要がある．なぜなら，単純手法はスレッドブロックの処理にデータ依存があるためである．**図 3** に入出力を統合して単純手法を処理する一例を示す．この例では，図に示す活性要素 p_4 は，スレッドブロック 2 が読み出し，スレッドブロック 3 が書き込む．しかし，CUDA は複数のスレッドブロックにデータ依存がある処理を許さない．したがって，単純手法は入力領域と出力領域を別々の配列に割り当てる必要があり，メモリ使用量が一つの配列に対して 2 倍必要となる．

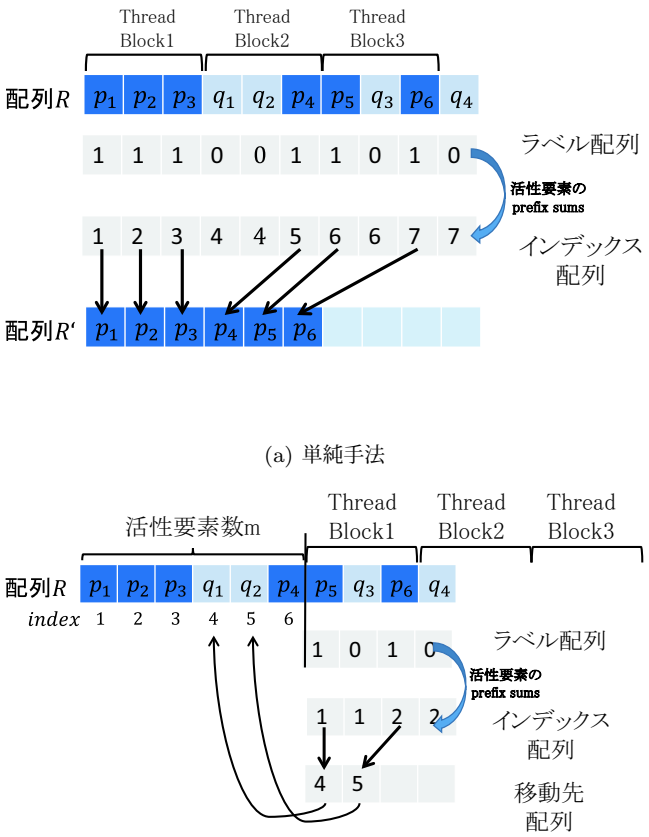


図 2 GPU によるインデックス計算

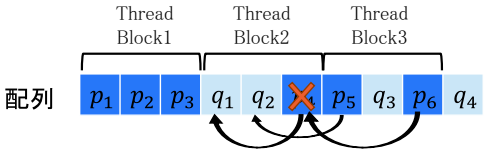


図 3 入力および出力を同一配列に格納した単純手法の処理

3.2 In-Place 型の詰め込み

前節の問題点を解決するために，多く分枝限定法の部分問題が要素の並び順を詰め込みの前後において維持する必要がないことに着目し，in-place 型の詰め込みを実現する．

提案手法は，図 2(b) に示すように，要素を含む配列を 2 つに分け，前半の配列に含まれる非活性要素の位置に，後半の配列内の活性要素を挿入する．この処理は活性要素の読み込みおよび非活性要素への書き込みのみにより構成されるため，入力および出力領域を区別する必要はない．

提案手法の詳細を述べる．アルゴリズム 1 に，提案手法の疑似コードを載せる．例として，大きさ n の配列 R を詰め込むことを考える．ここでは，配列 R の i 番目の要素を r_i $(1 \leq i \leq n)$ と呼ぶ．各 r_i は活性要素 p_j $(1 \leq j \leq m)$ もしくは非活性要素 q_k $(1 \leq k \leq n-m)$ のいずれかである．

まず，活性要素数 m 番目のインデックスにおいて配列を 2 つに分ける．ここでは，活性要素数 m は活性要素の prefix sums により求める．この配列の 2 つの領域を，領域 R_1 $(= r_i | 1 \leq i \leq m)$ および領域 R_2 $(= r_i | m+1 \leq i \leq n)$ と呼ぶ．この際，R_1 に含まれる非活性要素数 t は，R_2 に含まれる活性要素数と同一となる．

次に，R_1 内の非活性要素 q_k $(1 \leq k \leq t)$ に，R_2 内の活性要素 p_j $(m-t \leq j \leq m)$ を挿入する．つまり，$p_j \rightarrow q_{m-t+j}$ $(1 \leq j \leq m)$ の操作により詰め込みを実現する．この操作を図 2（b）に示す．

この際，p_j および q_{m-t+j} の要素を対応付ける必要がある．そのため，詰め込みの前処理（**図 4**）として，R_1 の非活性要素の prefix sums，R_2 の活性要素のインデックスを prefix sums により計算する．また，スレッドが各 p_i の移動を処理する際，移動先の非活性要素 q_{m-t+j} のオフセットを求める必要がある．そのため，R_1 の非活性要素の位置を保持した移動先配列を作成する．各スレッドは移動先配列の要素を参照し，活性要素の移動先を決定する．この配列のインデックスは活性要素のインデックスと一致しており，それぞれ活性要素の移動先である非活性要素のインデックスを示す．

提案手法は，ラベル配列およびインデックス配列に格納するデータは単純手法と同一である．しかし，提案手法では配列を 2 つに分け，R_1 における非活性要素の prefix sums，および R_2 における活性要素の prefix sums をインデックス配列から導出し，その結果を用いる．

提案手法では，(1)R の活性要素数 m の計算，(2)R_1 の非活性要素のインデックス計算，および (3)R_2 の活性要素のインデックス計算の計 3 回の prefix sums を計算する．この前処理のオーバーヘッドを削減するために，(2)，(3) の prefix sums 計算を (1) の結果から導出する．ここで，(1) の prefix sums 結果を R' $(= r'_i | 1 \leq i \leq n)$ とする．このとき，(2) の prefix sums の各要素は，$i - r'_i$ により計算できる．次に，(3) は，R_1 の最後の要素 $R[m]$ の結果を，$r_m - r_i$ により計算できる．以上の操作により，前処理は 1 度の配列 R に対する prefix sums 計算，および移動先配列の作成となる．

4. 評価実験

提案手法による速度向上および処理可能な要素数を実験により評価する．実験では，Lalami らの既存手法，提案手法および単純手法を 0-1 ナップザック問題に適用し，問題を解くための全体の実行時間を計測した．アプリケーションは Lalami らの実装を基に作成している．

実験では 2 種類のデータセットを用いる．物品の重量および価値に相関があるデータセットおよび相関がないデータセットの 2 つである．適用するデータセットの詳細を，表 1 に示す．データセットは，Martello ら [10], [11] の問

Algorithm 1 配列パッキングの擬似コード

Weights, *Profits*: 重量・価値の暫定解を格納する配列
Uppers, *Lowers*: 解の上限値，下限値を格納する配列
Labels: 要素の活性・非活性の状態を示す配列（ラベル配列）
Indexes: Labels の prefix sums 結果を格納する配列（インデックス配列）
Goals: 活性要素の移動先を格納する配列（移動先配列）
bestL: 既知の部分問題における最良下限値

for $i = 0$ to 要素数 n **do**
  branchOperation(*Weights*, *Profits*)
  boundOperation(*Weights*, *Profits*, *Uppers*, *Lowers*)
  bestL = findMaxBestLower(*Lowers*)
  Labels = createLabels(*Uppers*, *bestL*)
  Indexes, *Goals* = createIndexsAndGoals(*Labels*)
  packArrays(*Weights*, *Profits*, *Labels*, *Indexes*, *Goals*)
end for

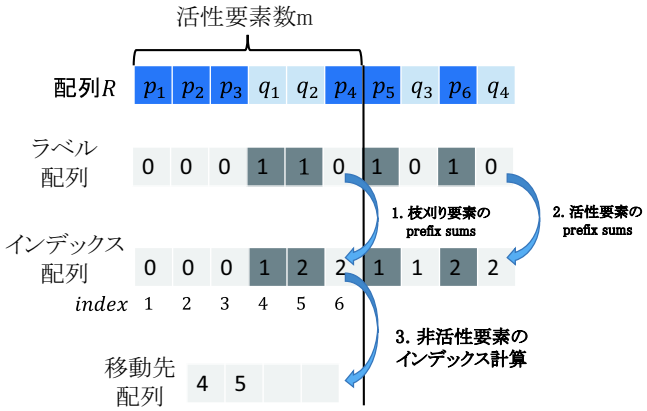


図 4 インデックス配列および移動先配列作成

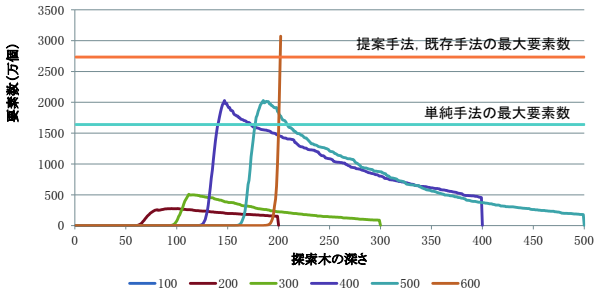
題生成器を用いた．

実験では，2GB のオフチップメモリを持つ NVIDIA Geforce GTX680，および Core i7 K3770 を搭載する Windows7 Professional 64bit の PC を用いた，ディスプレイドライバは 320.57 である．また，コンパイルには CUDA 5.5 および Visual Studio 2010 を用い，最適化オプションとして O2 を使用した．

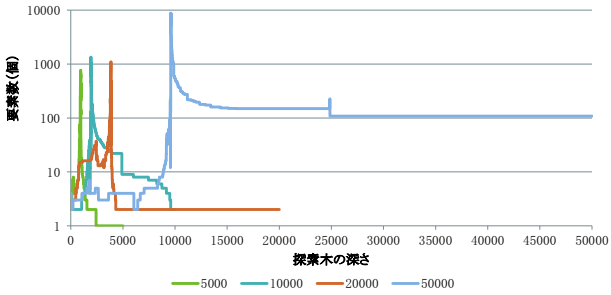
4.1 スケーラビリティの評価実験

まず，提案手法による処理可能な要素数の増大について評価する．そのため，物品数のパラメータを変更し，探索木の各深さにおける活性要素数を求めた．強い相関のデータセットについて要素数の遷移を**図 5**(a) に示す．同様に，図 5(b) には弱い相関の実験結果を示す．

図 5(a) におけるオレンジ色の横線は，既存手法により処理できる最大要素数が約 2730 万を示す．この要素数以上となるデータセットを用いる場合，既存手法の実行時にメモリ不足によりエラーが発生する．同様に青色の横線は，既存手法および単純手法の最大要素数が約 1640 万である



(a) 強い相関のデータセット



(b) 弱い相関のデータセット

図 5 スケーラビリティの計測

ことを示す．

強い相関のデータセットを用いた場合，提案手法および単純は要素数 500 までのデータセットの実行に成功している．しかし，既存手法は要素数 300 までのデータセットしか実行できず，提案手法と比べ 60%のデータセットしか実行できない．これは，提案手法および単純手法は In-Place な詰め込みをしているためである．

一方，弱い相関の実験結果では，一時的に要素数が増大するが最大でも約 8800 要素であり，一つの配列が約 34KB 程度に収まる．要素数が膨大とならずメモリ不足がないため，全ての手法において 50000 までのデータセットを実行できる．

要素数が膨大にならない理由は，弱い相関のデータセットは価値と重量に相関がなく，価値/重量の偏りが大きくなるためである．限定操作において，価値／重量の小さい物品を選択する要素は枝刈りされやすく，要素数は極度に増大しない．

以上から，提案手法は要素数が膨大となるデータセットに対して有用であると考えられる．

4.2 実行時間の評価実験

最後に，提案手法による分枝限定法の速度向上を実験により評価する．強い相関のデータセットおよび弱い相関のデータセットによる実行時間をそれぞれ**図 6** および**図 7** に示す．また，メモリ参照回数の詳細を表 3 に示す．

強い相関のデータセットは，既存手法が提案手法と比較して要素数 200 において，最大 1.15 倍の速度向上を得た．これは，全体の実行時間の 11%を占める CPU・GPU 間

表 1　データセットのパラメータ

	強い相関	弱い相関
物品数 n	100,200,300,400,500,600	5000,10000,20000,50000
価値 p_i	random$(w_i + 1000 - 20, w_i + 1000 + 20))$	random$(w_i - 1000, w_i + 1000)$
重量 w_i	random$(1, 10000)$	
容量 b	$\sum_{i=1}^{n} w_i * 100/1001$	

データ転送を排除できたためである．一方で，詰め込み処理の実行時間は 96 ミリ秒から 124 ミリ秒となり，25%速度低下している．これは，カーネル実行のオーバーヘッドが原因であり，GPU より CPU 処理が高速となっている．

また，提案手法は単純手法と比較すると，要素数 300 において実行時間は 8.8%の速度低下となっている．これは，移動先配列を作成するための前処理がオーバーヘッドとなり，インデックス計算の実行時間が 256 ミリ秒から 320 ミリ秒に 25%増大しているためである．また，提案手法ではインデックス計算のメモリ読み込み回数が単純手法と比較して，2.32×10^8 回から 3.08×10^8 回となり 32%増加している．

したがって，強い相関のデータセットからは，提案手法は単純手法と比べて，メモリ使用量および実行時間においてトレードオフの関係があることが分かった．

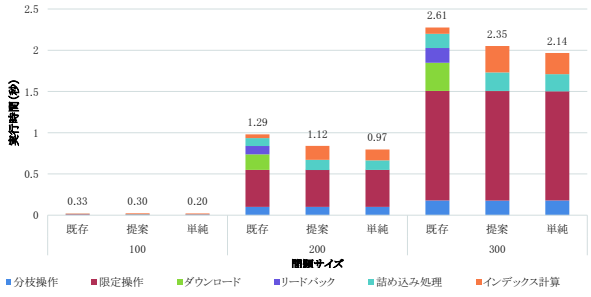
一方，弱い相関のデータセットにおいて，既存手法，提案手法および単純手法において実行時間にほぼ差はなかった．これは，Lalami らの実装は要素数が 192 以下である場合は， 全ての操作を CPU 処理するためである．弱い相関のデータセットによる実験では，要素数が 192 を超える探索木の深さの数は，全ての深さの数に対して最大 5.6%となり，GPU により処理する割合が小さかった．

5. まとめ

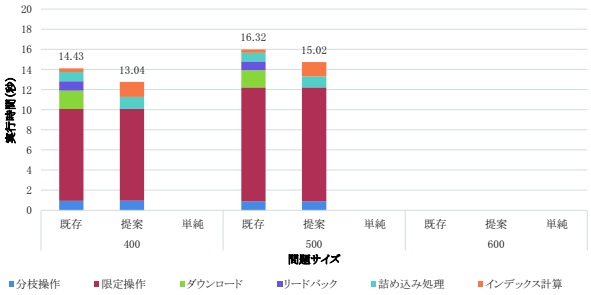
本稿では，GPU における分枝限定法の高速化を目的として，メモリ参照効率を高めるための配列パッキング手法を提案した．提案手法は，CPU の替わりに GPU が配列を詰め込むことにより，CPU・GPU 間のデータ転送を最小限に抑える．詰め込みにより配列を密に維持でき，分枝限定操作におけるメモリ参照効率を高められる．さらに，多くの対象問題において配列要素の並び順を詰め込みの前後で維持する必要がないことに着目し，in-place 型の詰め込みを実現する．これにより，非 in-place な単純手法と比較して，およそ 2 倍の部分問題を GPU 上で並列処理できる．

ナップザック問題を用いた実験では，CPU 上で配列を詰め込む既存手法と比較して 1.15 倍の速度向上を得た．また，in-place 版は非 in-place 版と比べて扱える部分問題の数を増やせたが，性能は 8.8%減少した．

今後の課題は，メモリ使用量を調節できる新たな探索法を作成することにより，幅優先探索では解けなかった問題サイズの大きい問題を GPU により解くことである．



(a) 要素数 100-300



(b) 要素数 400-600

図 6　実行時間：強い相関のデータセット

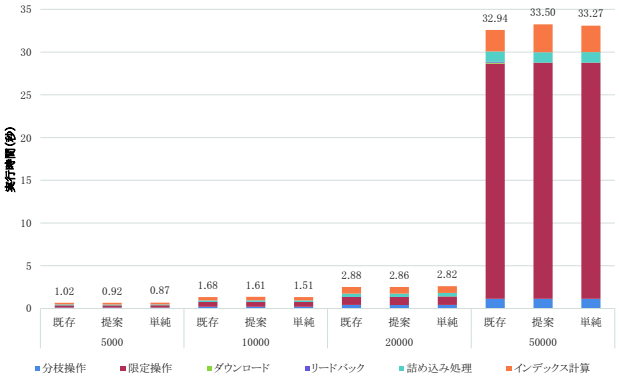


図 7　実行時間：弱い相関のデータセット

謝辞　本研究の一部は，JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」，科研費 23300007，および 23700057 の補助による．

参考文献

[1] Lalami, M. E. and El-Baz, D.: GPU Implementation of the Branch and Bound Method for Knapsack Problems, *Proc. 26th IEEE Int'l Parallel and Distributed Process-*

表 2 読み込み回数：強い相関のデータセット

	インデックス計算		詰め込み処理	
	提案手法	単純手法	提案手法	単純手法
100	2.51×10^5	1.99×10^5	2.02×10^4	2.08×10^5
200	1.73×10^8	1.30×10^8	1.71×10^8	1.75×10^8
300	3.08×10^8	2.32×10^8	3.04×10^8	3.11×10^8
400	1.71×10^9	—	1.71×10^9	—
500	1.59×10^9	—	1.59×10^9	—

表 3 書き込み回数：強い相関のデータセット

	インデックス計算		詰め込み処理	
	提案手法	単純手法	提案手法	単純手法
100	1.51×10^5	1.26×10^5	1.25×10^5	1.30×10^5
200	1.08×10^8	8.62×10^7	1.06×10^8	1.09×10^8
300	1.91×10^8	1.54×10^8	1.90×10^8	1.94×10^8
400	1.06×10^9	—	1.07×10^9	—
500	9.84×10^8	—	9.89×10^8	—

表 4 読み込み回数：弱い相関のデータセット

	インデックス計算		詰め込み処理	
	提案手法	単純手法	提案手法	単純手法
5000	1.02×10^5	9.13×10^4	2.02×10^4	6.68×10^4
10000	1.53×10^5	1.26×10^5	1.85×10^4	9.80×10^4
20000	1.06×10^5	1.86×10^5	6.05×10^4	1.47×10^5
50000	5.27×10^6	4.15×10^6	4.41×10^6	9.09×10^5

表 5 書き込み回数：弱い相関のデータセット

	インデックス計算		詰め込み処理	
	提案手法	単純手法	提案手法	単純手法
5000	5.18×10^4	5.09×10^4	7.28×10^3	4.70×10^4
10000	8.33×10^4	7.02×10^4	3.63×10^3	6.03×10^4
20000	1.10×10^5	1.06×10^5	2.52×10^4	1.00×10^5
50000	2.95×10^6	2.39×10^6	2.75×10^6	2.49×10^5

ing Symp. Workshops & PhD Forum (IPDPSW'12), pp. 1769–1777l (2012).

[2] Hoberock, J. and Bell, N.: Thrust (2013). `http://thrust.github.io/`.

[3] Lawler, E. and Wood, D. E.: Branch-And-Bound Methods: A Survey, *Operations Research*, Vol. 14, No. 4, pp. 699–719 (1966).

[4] Kurowski, K. and Mackowiak, M.: Parallel branch and bound method for solving traveling salesman problem using hybrid CPU and GPGPU computing and the XMPP based communication protocol (2011). `http://apps.man.poznan.pl/trac/xmpp-mpi/export/383/publications/AINA\%202011/AINA\%202011\%20article\%20v2/AINA\%202011\%20article\%20v2.pdf`.

[5] Boukedjar, A., Lalami, M. E. and El-Baz, D.: Parallel Branch and Bound on a CPU-GPU System, *Proc. 20TH Euromicro Int'l Conf. Parallel, Distributed and Network-based Processing (PDP'12)*, pp. 1769–1777 (2012).

[6] Carneiro, T., Muritiba, A., Negreiros, M. and de Campos, G. L.: A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU, *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd Int'l Symp.*, pp. 41–47 (2011).

[7] Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K. and Shen, X.: On-the-Fly Elimination of Dynamic Irregularities for GPU Computing, *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, pp. 369–380 (2011).

[8] Zhang, E. Z., Jiang, Y., Guo, Z. and Shen, X.: Streamlining GPU Applications On the Fly, *Proc. 24th ACM Int'l Conf. Supercomputing (ICS'10)*, pp. 115–125 (2010).

[9] Harris, M., Sengupta, S. and Owens, J. D.: Parallel Prefix Sum (Scan) with CUDA, *GPU Gems 3* (Nguyen, H., ed.), Addison Wesley, chapter 39, pp. 851–876 (2007).

[10] Martello, S., Pisinger, D. and Toth, P.: New trends in exact algorithms for the 0-1 knapsack problem, *European J. Operational Research*, Vol. 123, No. 2, pp. 325–332 (2000).

[11] Martello, S., Pisinger, D. and Toth, P.: *Dynamic Programming and Tight Bounds for the 0-1 Knapsack Problem*, Datalogisk Institut København: DIKU-Rapport, Datalogisk Institut, Københavns Universitet (1997).