IDA*探索を用いた15パズルSolverのGPUに適した並列探索法について

萩野谷 一二　　古宮 嘉那子†

Iterative Deeping A* Search （IDA*探索）を用いた15パズルSolverをGraphic Processing Unit（GPU）に単純移植すると，手数の長い問題の探索において，性能が向上するどころか劣化するという深刻な問題が発生する場合がある．その原因は，IDA*探索の内部で行っている深さ優先探索でスレッド分散が多発しているためと考えられる．
本発表では，IDA*探索の内部探索処理に幅優先探索法の考えを導入してスレッド分散を解消すると共に，その際発生する作業域不足を共有メモリを用いたソフトキャッシュ機能により回避する方式を提案する．また，提案方式を実現したSolverの作成・評価を行った結果，NVIDIA GeForce GTX580とIntel Core i7 2600 3.4GHz CPUを使用した場合，15パズルの最長手数に近い問題（約80手）において，CPUのみの逐次探索の場合と比較して実行時間を30分の1以下に短縮することができた．

Parallel Search of GPU for 15-Puzzle Solver with IDA* Search

Kazuji Haginoya,　and　Kanako Komiya†

When a 15-puzzle solver with Iterative Deeping A* Search (IDA*Search) was simply introduced into Graphic Processing Unit (GPU), the serious problem sometimes happens: the performance decreases on searches of some maps whose optimal solution has many moves. This is because the thread divergence becomes a serious problem in depth-first search which is used in IDA* Search. This paper proposes the method which solves this problem by introducing breadth-first search instead of depth-first search into IDA* Search and avoids shortage of the work area by soft-cache using the common memory. The results of the experiment of the solver with this method show that the 15-puzzles whose solution is almost the longest (around 80 moves) were solved 30 times as fast as the serial search of GPU using NVIDIA GeForce GTX580 and Intel Core i7 2600 3.4GHz CPU.

† 茨城大学工学部
Department of Computer and Information Sciences, Ibaraki University

ⓒ 2015 Information Processing Society of Japan

1. はじめに

15 パズルは 4 x 4 の盤面に 1 から 15 までの数字を順序良く揃えるパズルであり，良く知られているパズルの１つである．このパズルの最短手数が最長となる問題（最長手数問題）が 80 手であることは Cenju というスパコンを使って既に求められている（文献１）が，手数の長い問題（付録 A.1 の例 1,2,3 参照）の最短手数解を求めることはそれ程簡単ではない．例えば，例 1, 2 を手元の DeskTop PC(3.4GHz CPU) で解いてみると，表 1 の結果となった．Takaken 版 Solver（文献１）は，通常の Solver （筆者の自作版）に較べて約 30 倍高速である．

本発表では，ごく普通の 15 パズル Solver (CPU 版 Solver) を GPU を用いた大規模並列探索によって高速化する手法を提案する．また，提案手法の実装を行い Takaken 版 Solver と同等以上の性能を実現できたことを報告する．

表 1 Solver の簡単な性能比較

	Takaken 版	通常の Solver
例 1	47 秒	1223 秒
例 2	641 秒	約 5 時間 7 分

2. 関連技術および関連研究

2.1 15 パズルの最短手数解を求めるアルゴリズム

パズルの最短手数解を求める探索アルゴリズムとして IDA*探索がよく使用される．IDA*探索では，よい Lower Bound Estimater（LBE）が作れることが前提となる．しかし，15 パズルにおいて IDA*探索を用いる場合，マンハッタン距離（MD）はあまり精度のよい LBE ではないため，最長手数（80 手）に近い問題では探索範囲を絞り切れず時間がかかりすぎて解けないという結果になる．

Takaken 版 Solver では，高橋謙一郎氏独自の考案となる InvertDistance（ID）と WalkingDistance （WD）を用いて精度のよい LBE を得ることに成功していることが高速解法の秘密である．（付表 A.1）しかし，ID,WD のような精度のよい LBE は，パズルに対する深い洞察力によって初めて得られるものであり，誰でも作れるというものではない．

一方，GPU を用いた高速化の試みは多方面で行われており，文献 2,4,5,6 では数十倍という成功事例も報告されている．以下では，15 パズル Solver に関連する 2 つの事例を簡単に紹介する．

まず，文献 2 は IDA*探索により Rubik キューブの最短経路を求める Solver を GPU を用いて高速化を行った例である．

その特徴は，

(1) 探索の半分を CPU 側で行い，残りの部分を GPU で探索するという役割分担
（GPU 側の探索の深さは高々10 手となる）

(2) LBE に相当する距離テーブルの容量は，2.2G と 88G の２つを使用

(3) CPU と GPU の並列化による探索時間の短縮

(4) 約 50 万スレッドという大規模並列探索

である．その結果，GTX570 (480core, 1.46GHz）を使用した評価実験では，CPU による逐次探索に較べて 21 倍高速となり，グラフの最短経路問題＋IDA*探索では，GPU が活用可能と評価している．

次に，文献 4 は最良優先探索（BFS）を用いた最短経路探索を GPU に向いた並列探索に変更した例である．
GPU において並列探索を行うため，(1) Priority Queue を分散化して各コアで分担可能とし，(2) Node Duplication Detection のために Hash 機構を採用している．(closed list（探索済のノードの一覧）の管理) 具体的な Hash 機構として， Parallel Cockoo Hashing と Parallel Hashing with Replacement（ハッシュ値の衝突時は古い Node を新 Node で置き換える方式）を用意し，使用可能メモリ容量 (closed list) からどちらかを選択する考えである．

一例として，GPU に Telsa K20c（2496core, 706MHz）を使用した実験では，約 60 手の 15 パズルの問題で約 30 倍強の改善効果が得られている．

3. GPU を用いた高速化への取り組み

3.1 DFS 版の実装

文献 2,5,6 の事例を参考に，CPU 版 Solver を GPU に単純移植した DFS 版を作成した．なお，CPU 版 Solver の概略は付録 A.2 を参照されたい．次に，DFS 版作成時の主な修正点について説明する．

(1) CPU 側と GPU 側の役割分担

GPU は単純・大量処理に向いているので，IDA*探索処理内部の深さ優先探索（DFS）処理をオフロードすることとした．GPU 側で行う探索の深さ（OLP:OffLoad Point）を設定して GPU 側の負荷の調整を行う．

また，OLP は CPU から GPU 側への引き継ぎ情報（Seed）のそれぞれの仕事量も調節する役割がある．OLP を大きくしすぎると Seed の総数が少なくなり，個々のスレッドの仕事量のばらつきが大きくなる．（負荷の平滑化の問題）これは，スレッドの終了処理のばらつきとして表面化する．具体的には 4.5 OLP の調整の節を参照されたい．

(2)再帰関数の変更

CPU 版では DFS 処理を再帰関数で実装しているが，GPU では再帰関数を使用できないので，配列を用いたループ処理に変更した．

(3) GPU 使用時の留意点の考慮

文献 5,6 で述べられている留意点を参考に，動的負荷バランス機能，スレッド分散対策（if 文の削減），共有メモリアクセス時のバンク衝突の回避策などを実施した．

(4) コンスタントメモリの活用

MD テーブルは，小容量かつ全スレッドから頻繁にアクセスされるためコンスタントメモリ(Cmem)に配置した．

3.2 DFS 版の評価

DFS 版において，Blocks=32, Threads=64 に固定して OLP を変化させて探索時間の変化（図 1 参照）をみると，OLP が増えるに従って（GPU への offload 量が増えると），探索時間が極端な増加となっていることがわかった．

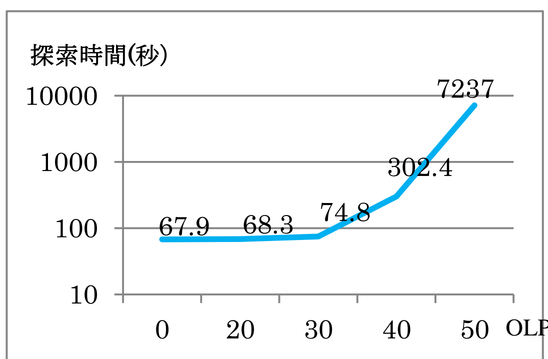


図 1　OLP と探索時間の関係

その原因を調べるため探索の深さ（Hand 数）ごとのノード数の分布をみると，図 2 のように中央にピークのあるグラフとなった．

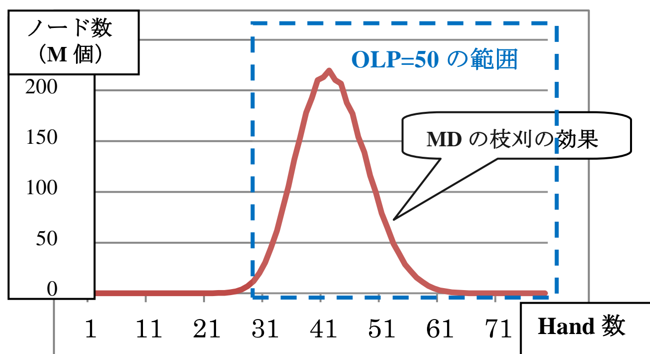


図 2　Hand 数とノード数の分布

このようにノード数の分布に大きな偏りがあることが極端なスレッド分散* を引き起こし，性能劣化の主因となっていると考え，各スレッドの動作を分析した．

付録 A.3 の例のように，両方のスレッドが同じ処理をする場合は同時に実行されるが，そうでない場合は片方しか実行されず，他方は待たされてしまう．そのため，A の nest が極端に深い場合，B の処理はかなり待たされてしまう．

以上より，この問題の本質は，スレッドと Seed の関係が固定されているためと判断した．また，文献 3 にはαβ木探索ではスレッド分散が深刻な問題となることが指摘されており，同様な問題と推測している．

3.3 対策案の検討

先の深さ優先探索によるスレッド分散の問題は，1 手進める毎に Seed の assign 処理を行う幅優先探索処理では回避できる（付録 A.3 を参照）と考え，以下の対策案 1,2 を検討した．

案 1 ：IDA*探索の内部探索を幅優先探索に変更したアルゴリズム

1) ブロック内の各スレッドに Seed を assign する
もし，assign する Seed が不足する場合は，グローバルメモリ（Gmem）の Seed で補てんする
Gmem の Seed がなくなれば，処理を終了する

2) 各スレッドは，1 手先の Seed を生成し，Seed バファに格納する

3) Seed バファが空きになるまで，1), 2) を繰返す

案 1 のアルゴリズムによる Seed バファ内の構成を図 3 に従って説明する．h0 は Gmem から取り出された Seed であり，生成元となる Seed を表している．h0 の Seed から生成された Seed が H1 (h1 を含む)である．以下，h1 → H2, … と Seed 生成を繰り返していく．　図 3 の水色の部分は生成元となった Seed を表し，黄色の部分はまだ生成元になっていない Seed を表している．

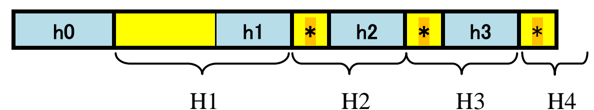


図 3　Seed バファの構成イメージ

案 1 の問題点は，h0 → H1, h1 → H2, …と Seed 生成を繰り返していくと，生成元になる Seed が少なくなり，＊の Seed を集める処理が必要になることである（水色の生成済の Seed があるため意外と面倒な処理）

案 2 ：案 1 の Seed assign 処理の改善

案 1 からの変更点は，「1）において，assign した Seed を Stack から削除し，その空き領域を生成した Seed を格納するために使用する」ことである．つまり，この案 2 では GPU で行った探索の履歴をすべて破棄するという大きなトレードオフを行っている．そのため，GPU からの応答情報は Goal に至った Seed の Gmem 上の id（中間 Node の

＊ スレッド分散とは，GPU において多数のスレッドが並行して if 文を処理する場合，一方のスレッドのみ実行され，他方のスレッドは休止状態となることである．スレッド分散が多発するとそれだけ効率は悪くなる．

位置）のみとなる．CPU 側では，この中間 Node から Goal までのパスを再探索しなければならない．

案 2 において Seed 生成を繰り返した場合の Seed Stack 構成を以下の図 4 に従って説明する．まず，(1) Gmem から Seed（h0）を取り出し，Seed Stack に push する．次に，(2) h0 の seed から生成された新しい Seed（H1）は，もとの Seed（h0）を上書きして格納する．更に，(3) h1 の seed から生成された新しい Seed（H2）は，もとの Seed（h1）を上書きして格納する．以降，これを繰り返す．Seed Stack は，(5)までの処理を行った場合の状態を表す．図中の黄色の部分は，まだ処理されていない Seed である．案 1 と異なり，連続した領域となっていることに留意されたい．

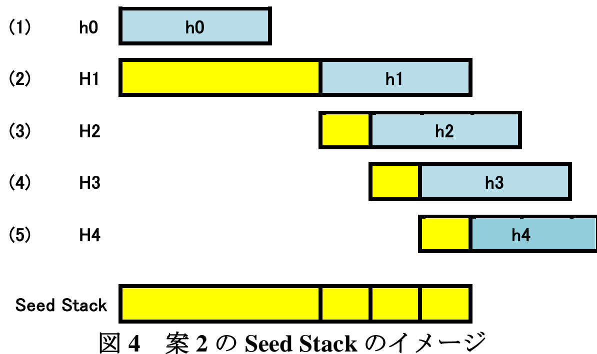


図 4　案 2 の Seed Stack のイメージ

【案 1,案 2 の選択】

15 パズル Solver においては，案 2 のトレードオフ項目は対処法があるため大きな問題にはならないと考え，Seed の取り出し論理が単純となりしかも作業域が少なくて済む案 2 を採用することとした．

4.　提案方式の実装

4.1　概要

図 5 は提案方式（案 2）全体の処理概要を示したものである．手順 1～4 を繰り返す単純な構造となっている．

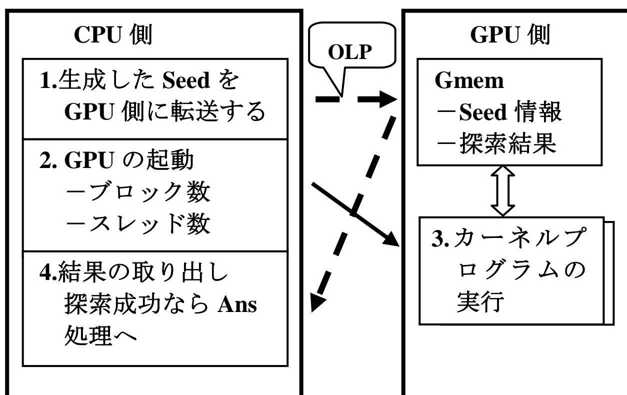
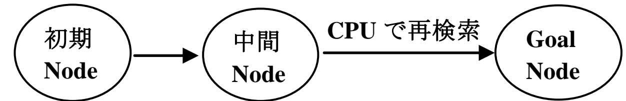


図 5　提案方式の全体構成

GPU 側の探索の深さは OLP で調整する．また，GPU の多重度（ブロック数，スレッド数）は，共有メモリ資源の制約により上限がきまる．

【Ans 処理の補足】（案 2 のトレードオフ対策）

CPU 側では，GPU から通知された中間 Node 情報をもとに，中間 Node → Goal Node の検索を行い，初期 Node から Goal Node までの手順を完成させる．



4.2　WIda（仮称）アルゴリズム

図 6 は案 2 の GPU 側の具体的な手順である WIda アルゴリズムの概念を示している．

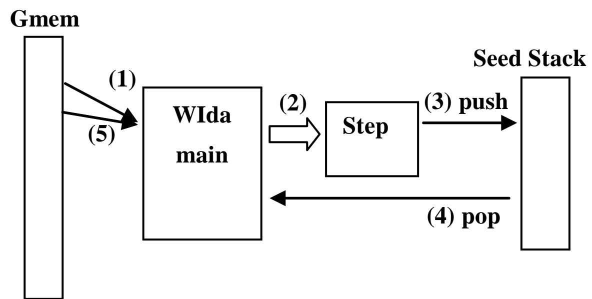


図 6　WIda アルゴリズムの概念

以下では WIda アルゴリズムの詳細を図 6 の(1)～(5)の順に従って説明する．まず，(1) Gmem から最大 T 個（T は 1 ブロックあたりのスレッド数）の Seed を取り出す．次に，(2) 取り出した Seed を Step 関数に渡す．(3) Step 関数は，1 手先の Seed を生成し，Seed　Stack に格納する．（Goal を検出した場合は元の Seed id（Gmem 上の位置）を検索結果として Gmem に格納し処理を終了する）　通常，Step 関数の処理が終わると，(4) Stack に T 個以上の Seed がある場合，Stack の先頭から T 個の Seed を取り出し，(2)の処理へ戻る．（このとき，Stack 内にその Seed は残さない）(5)　Stack に T 個未満の Seed しかない場合，Seed 補てん処理を行い，Stack の先頭から最大 T 個の Seed を取り出し，(2)の処理へ戻る．（Stack の Seed が T 個になる様に Gmem から Seed を補てんする）もし，Gmem の Seed がなくなり，かつ，Stack が空になった場合は終了となる．

4.3　Stack Cache 機構

Seed Stack を共有メモリ（Smem）に単純 mapping すると，Smem メモリ不足（スレッドなどの多重度の上限となること）が懸念される．この制約を解消するため Stack Cache 機構を装備する．

Stack Cache 機構は，WIda の Seed Stack を Smem を使ったソフト Cache 機能とデータ格納用の Gsave(Gmem)によっ

て実現する．本手法では，Seed Stack と同じ大きさの Gsave を利用する．また，Cache には Seed Stack の Top データを保持し，アクセス時間の短縮となるように制御する．

Stack Cache の制御方式は，図 7 の(1)~(4)の順で行われる．まず，(1)SeedStack への push 操作で Cache への write を行い，(2) 同時に Gmem への Write を行う．(write through 方式) 次に，(3) prepare 操作で Gsave から Cache に Load する．(pop 操作で必要とする Seed の数は高々T 個なので Step 処理の完了後に preload する) 最後に(4) pop 操作により，Cache から Load する．

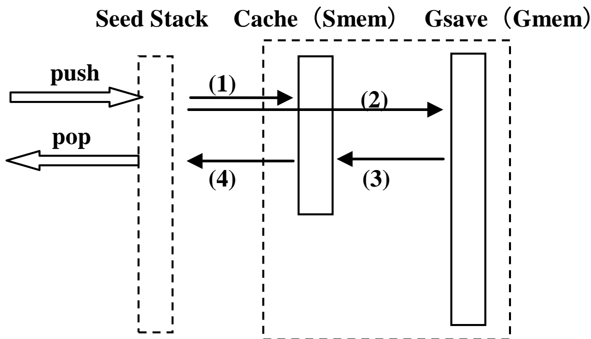


図 7 Cache 制御による Seed Stack の実装イメージ

4.4 WIda 版, Cache 機能付き WIda 版の評価

上記の提案方式を検証するため，WIda 版（Seed Stack を共有メモリに配置したもの)，および，Cache 機能付き WIda 版（WIda 版に Stack Cache 機能を追加したもの）を作成した．

WIda 版により WIda アルゴリズムを検証する．Cache 機能付き WIda 版は，(1) Stack Cache 機構を装備することで性能が大幅に劣化しないか，(2) Cache 制御が有効に働くかという 2 点を検証する．

図 8 に，OLP=50, Threads＝４に固定した場合の WIda 版，Cache 機能付き WIda 版の測定結果を示す．

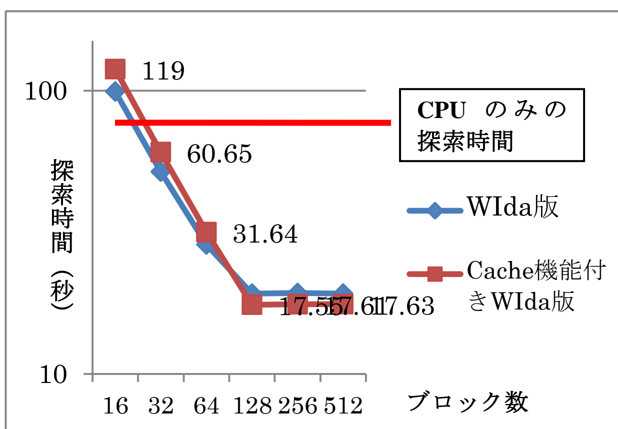


図 8 WIda 版,Cache 機能付き WIda 版の探索性能

(1) WIda アルゴリズムの検証

DFS 版で **7237** 秒かかった探索が WIda 版では約 **19** 秒まで短縮されており，スレッド分散は抑制できた．しかし，WIda 版は共有メモリの容量制限からスレッド数を大きくできず，スレッド数 ＝4 が最適値となった．(この状態では，コア数は 4 / 32 = 1 / 8 しか使用していない)

(2) Stack Cache 制御の検証

Cache 機能付き WIda 版は Blocks=16 のところで性能差がでている．これは Cache 制御の重さと考えられるが大幅な劣化ではない．また，ブロック数が増えると解消しているので，Seed Stack を Gmem に配置したが Cache 制御により Gmem の大きな Latency の隠ぺいに成功したと判断してよいであろう．

4.5 OLP の調整，その他

4.5.1 OLP の調整

OLP (OffLoad Point) は，GPU 側で行う探索の深さを指定するものである．OLP が大きいほど GPU 側の処理は重くなる．(その分 PC 側が軽くなる) 図 9 は，Cache 機能付き WIda 版を以下の測定条件（Blocks=112,Threads=64）に固定し，OLP を変化させて測定したものである．

この図より，(1) OLP=55 あたりが最適値となっていることがわかる．(2) OLP の上昇と共に，全体の探索時間が短縮されているのは当然の結果である．(減少しているのは PC 側の探索時間であるから) (3) OLP=60 で劣化しているのは，Seeds が少なく負荷分散が難しくなり始めたものと判断する．(PC 側の手順の探索を Nop にしても，7.92 秒かかっていることから明らか)

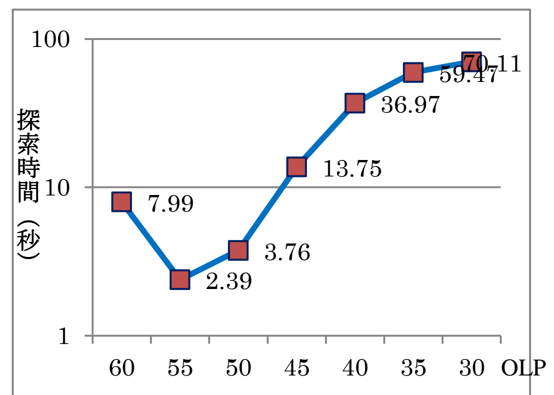


図 9 OLP と探索時間の関係

4.5.2 その他の調整

GPU の能力を十分に引き出すためには，その動作環境を最適に設定することが重要である．OLP の調整以外にも，HotSpot 対策（共有メモリへの Seed 格納処理時のアクセス負荷の集中対策)，Cache 容量と多重度（Blocks,Threads）の調整などがあるが，紙面の都合で詳細は割愛する．

ⓒ 2015 Information Processing Society of Japan

5. 評価実験

各 Solver の性能を評価するため, 3 つの Solver(CPU 版, GPU 版, Takaken 版）について付録 A.1 の 3 つの問題の解法を行った．その測定結果を表 2 に示す．なお，測定に使用したハードウェアは，NVIDIA GeForce GTX580 と Intel Core i7 2600 3.4GHz CPU である．

表 2　各 Solver の探索時間

	CPU 版	GPU 版	Takaken 版
例 1	1223 秒	38 秒	47 秒
例 2	約 5 時間 7 分	521 秒	641 秒
例 3	約 12 時間 37 分	1646 秒 *	2625 秒

*：最適値(OLP=48)での測定値は 1221 秒
CPU 版 ：MD を使用した通常の Solver（自作版）
GPU 版：最新の Cache 機能付き WIda 版を使用
動作条件：OLP=55, Blocks=144, Threads=96
Takaken 版：高橋謙一郎氏 HP（文献１）より入手

この表より，(1) CPU による逐次探索 Solver の約 30 倍強の性能向上を達成し，(2) Takaken 版 Solver と比較してもそん色ないことがわかる．

6. まとめ

GPU を用いて 15 パズル Solver を高速化するという取り組みは，IDA*探索処理を単純移植するだけではスレッド分散が多発してうまくいかないという問題があったため，本稿ではそれを回避する新たな制御方式を提案した．

本提案方式は，(1) WIda アルゴリズム（IDA*探索の内部でよく使われる深さ優先探索を幅優先探索の変形版に変更したもの)によりスレッド分散を大幅に抑制するとともに，(2) Stack Cache 制御（共有メモリをキャッシュとして使用することによりスレッドの作業域を Gmem に配置可能とした）によって共有メモリ容量の上限による探索の深さの制約を解消するものである．

そして，実際に実装を行い上記の効果を検証した結果，NVIDIA GeForce GTX580 と Intel Core i7 2600 3.4GHz CPU を使用した場合，15 パズルの最長手数に近い問題（約 80 手）において，CPU のみの場合と比較して実行時間を 30 分の 1 以下に短縮することができた．

本提案方式は，他のパズル問題の探索にも適用できるのではないかと考えている．例えば，Takaken 版 Solver，24 パズル Solver への適用などが期待される．

また，今回評価に使用した GPU は 2 世代前のアーキテクチャであるため，最新の GPU アーキテクチャを活用すれば，スレッド間の直接通信機構が可能であり，実装が容易となるとともに性能面でも更なる向上が期待できる．

参考文献

1) 高橋謙一郎氏のＨＰ(コンピュータ＆パズル)　公開プログラム研究開発ノート / 15 パズル自動解答プログラムの作り方 http://www.ic-net.or.jp/home/takaken/nt/slide/solve15.html
2) 早川広紀，村尾雄一, Rubik キューブの最小手数解の探索の GPU を用いた高速化，2013 年 3 月 Risa/Asia Conference 2013 http://cc11.math.kobe-u.ac.jp/lib/exe/fetch.php?media=cm:murao-risacon13-rubik.pdf
3) Kamil Marek Rocki, GPU における大規模モンテカルロ木探索 http://olab.is.s.u-tokyo.ac.jp/~kamil.rocki/phd_thesis.pdf
4) Yichao Zhou, Jianyang Zeng, Massively Parallel A* Search on a GPU，http://iiis.tsinghua.edu.cn/~compbio/papers/aaai_2015.pdf
5) 田中慶悟,藤本典幸, GPU を用いた N-Queens 問題の求解 情報処理学会シンポジウムシリーズ Vol.2011,No.6 pp.76-83,2011
6) 萩野谷一二，田中慶悟,藤本典幸，対称解の特性を用いた N-Queens 問題の求解と GPU による高速化，情報処理学会研究報告 Vol.2012-GI-27 No.7,2012
7) NVIDIA Corp., NVIDIA CUDA C Programming Guide 3.2
8) NVIDIA Corp., Tuning CUDA Applications for Fermi
9) NVIDIA Corp., PTX:Parallel Thread Execution　ISA Version 2.2

付録

付録 A.1　手数の長い問題の例

例 1：最短手数 ＝78 手

15	11	14	13
12	＊	9	10
8	3	6	5
4	7	2	1

例 2：最短手数 ＝80 手

＊	12	9	13
15	11	10	14
7	8	6	2
4	3	5	1

例 3：最短手数＝72 手

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	＊

注：＊は空きマス
例 3 は解法に時間のかかる問題として高橋謙一郎氏の HP（文献 1）に紹介されているもの

付表 A.1　LBE の比較

	例 1	例 2	例 3
MD	60	58	40
WD	64	66	40 *

注：WD は Takaken 版 Solver の LBE
＊：パターン DB を使用した LBE では 54 となる
WD－MD が探索の深さの差となる

付録 A.2　CPU 版 Solver（IDA*探索）の疑似コードの例

IDA*のメインループの概要

```
for (MaxHand=MD0;  ; MaxHand++) {   // MD0 は最初に与えられた局面の MD
    MaxHand を上限とする探索（深さ優先探索：DFS）
    もし，探索で解が見つかれば，終了する
    （そうでなければ，MaxHand を＋１して上記の探索を繰り返す）
}
```

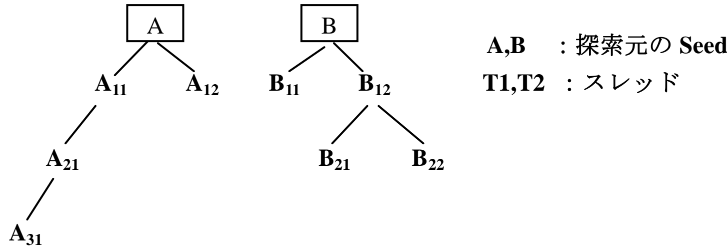
深さ優先探索（DFS）の概要

以下の Hand, M を入力として，1 手進めた局面の MD（m）を計算する．もし，Hand+m+1 > MaxHand であれば，その局面は探索範囲外なので廃棄する．(MD による枝刈)

Hand：既に探索済の手数
M　：現在の局面の MD
m　：次の局面の MD　（ M と移動した数字とその位置より以下の式で求めることができる）

m = md － md(c,p) + md(c,q)　//数字 c が位置 p から q へ移動する場合

付録 A.3　スレッド分散による待ちの発生の説明



【例１】 深さ優先探索時のスレッドの動作例

Step	0	1	2	3	4	5	6	7	8	9	10	11	12
T1	A	A11	A21	A31	a21	a11	a	A12	*	a12	A	*	*
T2	B	B11	*	*	b	*	*	B12	B21	b12	B22	b12	b

凡例
Axx, Byy は各スレッドが Node Axx, Byy へ進む処
axx, byy は各スレッドが Node Axx, Byy に戻る処理
＊は，該当スレッド待機状態（スレッド分散）

step0：スレッド T1,T2 は Gmem からそれぞれ A,B を取り出す.
step1：スレッド T1 は，A から A11 を生成し，スレッド T2 は，B から B11 を生成する.
step2：スレッド T1 は，A11 から A21 を生成するが，スレッド T2 は，B11 から生成できる Seed がなく，
　　　スレッド T2 は休止状態となる.
step3：スレッド T1 は，A21 から A31 を生成するが，スレッド T2 は，休止状態のまま.
step4：スレッド T1 は，A31 から A21 にもどり，スレッド T2 は，B に戻る.
step5：スレッド T1 は，A21 から A11 にもどり，スレッド T2 は，休止状態となる.
step6：スレッド T1 は，A11 から A にもどり，スレッド T2 は，休止状態となる.
step7：スレッド T1 は，A から A12 を生成し，スレッド T2 は，B から B12 を生成する.
　以降，同様の手順を繰り返す.

【例 2】 幅優先探索時のスレッドの動作例

Step	0	1	2	3	4	5	6	7	8
T1	A	A11	A12	*	*	b21	A21	@	A31
T2	B	B11	B12	B21	B22	b22	b11	*	x

凡例
Axx, Byy は各スレッドが生成した Node
axx, byy は Axx,Byy の Node から生成 Node がない状態
＊は，該当スレッドは待機状態（スレッド分散）
@は，次の Seed の補てん処理が行われる

step0：スレッド T1,T2 は Gmem からそれぞれ A, B を取り出す.
step1：スレッド T1 は A から A11 を生成し，スレッド T2 は B から B11 を生成する.
step2：スレッド T1 は A から A12 を生成し，スレッド T2 は B から B12 を生成する.
step3：スレッド T1 は A12 から生成する Seed がなく休止状態，スレッド T2 は B12 から B21 を生成する.
step4：スレッド T1 は A12 から生成する Seed がなく休止状態，スレッド T2 は B12 から B22 を生成する.
step5：スレッド T1,T2 はそれぞれ B21,B22 を Stack から取り出すが，ともに生成する Seed がない.
step6：スレッド T1,T2 はそれぞれ A11,B11 を Stack から取り出す．T1 は A21 を生成するが，
　　　T2 は生成する Seed がない.
step7：Stack に Seed は A21 しかないので，Seed 補てん処理を行う.（T2 は休止状態となる）
step8：スレッド T1,T2 はそれぞれ A21,@を Stack から取り出す．T1 は A31 を生成するが，
　　　T2 は@から新たな Seed x を生成する.

幅優先探索の場合，深さ優先探索に較べてスレッド分散は大幅に抑制される.

ⓒ 2015 Information Processing Society of Japan