修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報システム 学研究科　情報ﾈｯﾄﾜｰｸｼｽﾃﾑ学専攻　博士前期課程		
氏　　名	早川 広記	学籍番号	1352023
論 文 題 目	IDA*アルゴリズム（Iterative Deepening A* algorithm）の計算リソース多種混在ハードウェア向けの最適な実装方法の検討		

要　旨

近年の汎用計算機は CPU 以外に GPU や専用アクセラレーションユニット等, 様々なハードウェアがプログラマブルデバイスとして搭載されるようになった．特に，GPU や Intel®Xeon Phi のようなメニーコアアーキテクチャの拡張デバイスが搭載される例が多く，スーパーコンピュータへの導入事例も多い．例えば，東京工業大学のスーパーコンピュータ，TSUBAME は多数の CPU 及び GPU で構成される．他には，TOP500 上位にランクインするスーパーコンピュータにおいても GPU や Xeon Phi などのメニーコアデバイスが採用されている例が多く見られる．

このような構成のマシンでは, CPU とアクセラレーションユニットが効率良く協調動作するプログラムを実装すると高い性能を発揮できる．アクセラレーションユニットを活用する技術は未知な部分が多く存在し，高い性能を実現するための理論的な研究及び実験的な研究が盛んに行われている．

しかしながら，これらの研究はアクセラレーションユニット上で高い性能を実現することを目的として完結している例が多く，計算機上のアクセラレーションユニットは十分に活用できているが，CPU が十分に活用できていない例が多く見られる．これはアクセラレーションユニットのみならず，CPU も適切に活用すれば更なる性能向上ができる可能性を示している．

本研究では，計算リソースが複数混在した計算機上（CPU 4 cores 8 threads + GPU 2 台）において，GPU アクセラレーションの成功実績のある最短経路探索アルゴリズムである IDA*アルゴリズム（Iterative Deepening A* algorithm）を GPU 実行のみならず，計算機上の計算リソース全て（CPU＋GPU）を効率良く動作させることを目的とし，単純な GPU アクセラレーションを行った場合よりも高速な実装手法の有無を検討した．研究成果として，IDA*アルゴリズムをルービックキューブの最短解探索に適用した場合において，CPU と GPU を同時に効率良く使用する実装手法を 3 つ提案することができ，従来手法に対し高いパフォーマンスを得ることに成功した．

平成２８年度修士論文

IDA*アルゴリズム（Iterative Deepening A* algorithm）の計算リソース多種混在ハードウェア向けの最適な実装方法の検討

大学院情報システム学研究科

情報ネットワークシステム学専攻

学 籍 番 号 ：１３５２０２３

氏　　名 ：早川 広記

主任指導教員：笠井 裕之

指 導 教 員 ：森田 啓義

指 導 教 員 ：加藤 聰彦

提出年月日　：平成２８年７月２２日

目次

1. 研究概要

近年の汎用計算機は CPU 以外に GPU や専用アクセラレーションユニット等，様々なハードウェアがプログラマブルデバイスとして搭載されるようになった．特に，GPU や Intel®Xeon Phi のようなメニーコアアーキテクチャの拡張デバイスが搭載される例が多く，スーパーコンピュータへの導入事例も多い．例えば，東京工業大学のスーパーコンピュータ，TSUBAME は多数の CPU 及び GPU で構成される．他には，TOP500*上位にランクインするスーパーコンピュータにおいても GPU や Xeon Phi などのメニーコアデバイスが採用されている例が多く見られる．

これらのメニーコアデバイスは非常に高い演算性能を持つが，アーキテクチャに沿ったプログラミングをしないと性能が発揮できない．例えば GPU は，内部に数百から数千の小規模プロセッサを持ち並列動作をする性質を持つため，動作させるプログラムも数百～数億スレッドの並列化実装を行わなければならない．ここで要求されるプログラミング技術は従来の技術とは異なり，高度な技術が要求される．この技術は未知な部分が多く存在し，高い性能を実現するための理論的な研究及び実験的な研究が盛んに行われている．

しかしながら，これらの研究はメニーコアデバイス上で高い性能を実現することを目的として完結している例が多く，計算機上のメニーコアデバイスは十分に活用できているが，CPU が十分に活用できていない例が多く見られる．これはメニーコアデバイスのみならず，CPU も適切に活用すれば更なる性能向上ができる可能性を示している．

本研究では，計算リソースが複数混在した計算機上（CPU 4 cores 8 threads + GPU 2 台）において，GPU アクセラレーションの成功実績 [1]のある最短経路探索アルゴリズムである IDA*アルゴリズム（Iterative Deepening A* algorithm） [2]を GPU 実行のみならず，計算機上の計算リソース全て（CPU＋GPU）を効率良く動作させることを目的とし，単純な GPU アクセラレーションを行った場合よりも高速な実装手法の有無を検討した．研究成果として，IDA*アルゴリズムをルービックキューブ†の最短解探索に適用した場合において，CPU と GPU を同時に効率良く使用する実装手法を 3 つ提案することができた．

3 つの提案手法はそれぞれ特徴があり，1 つ目は，CPU1 スレッドの性能が十分に高く，他の計算リソースの制御が十分に行える場合における実装，2 つ目の提案はルービックキューブ探索という条件に特化した実装，3 つ目の提案は，CPU 動作周波数が不十分な条件下においても計算リソースの制御が十分に行えるような実装となっている．

実験結果として，従来研究 [1]の実装に対し，CPU 動作周波数が高い場合と低い場合で比較実験を実施したところ，CPU 動作周波数が高い条件下では提案手法 3 つ共に，従来研究 [1]より高い性能を示し，CPU 動作周波数が低い条件下では 2 番目，3 番目の提案手法が従来研究 [1]より高い性能を示す結果が得られた．

---

*世界で最も高速なコンピュータシステムの上位 500 位までを定期的にランク付けし、評価するプロジェクト

† Erno Rubik が 1974 年に考案した立方体状のパズル

2. GPU コンピューティングと関連研究

ここでは, メニーコアハードウェアプログラミングの 1 つである GPU コンピューティングについて記す．本稿の研究目的としては Intel®Xeon Phi などの GPU 以外のメニーコアハードウェアも扱うべきだが, 本研究で使用したハードウェアは CPU＋GPU の環境であり, 他の拡張ハードウェアを使用しなかったため, 本セクションでは GPU コンピューティングについてのみ記す.

2.1. GPU とは

GPU（Graphics Processing Unit）とは，コンピュータ用の映像処理装置，および映像出力装置の総称である．1970 年代は単純な画像出力や 2D 描画などを行う装置であったが，集積回路のプロセスルールの進歩に伴い処理性能が向上し, 現在では 3D 映像のレンダリングや汎用的なプログラムの実行（GPGPU）などが可能となっている．現代の GPU には小規模のプロセッサが数百～数千個搭載されており，その演算能力は 8 TFlops（一般的な民生用 CPU の約 10 倍）を超える物も少なくない.

2.2. GPU コンピューティング（GPGPU）

GPGPU（General-Purpose computing on GPU）は GPU を汎用的なプログラムの実行デバイスとして使用する技術の事である.

GPU を用いて汎用プログラムを実行する手段はいくつか存在するが，GPU ベンダの提供する開発環境を用いるのが一般的である. 現在高性能な GPGPU 環境を提供している GPU ベンダは NVIDIA および AMD の 2 社存在し, NVIDIA からは CUDA, AMD からは ATiStream と呼ばれる開発環境が提供されている．しかし，両環境に互換性は無く，NVIDIA 製 GPU では AtiStream は使用出来ず，同様に AMD 製 GPU で CUDA は使用できない.

最近では OpenCL と呼ばれるフレームワークを用いる方法も普及しつつある．OpenCL は並列コンピューティング向けのフレームワークであり，GPU のみならず，マルチコア CPU での開発も行える. また, GPU や CPU の開発元に依存しない開発が行えるのが特徴である.

本研究では，NVIDIA 製 GPU および CUDA（Compute Unified Device Architecture）を用いて開発を行う．CUDA は NVIDIA が提供する統合開発環境であり，同社製 GPU を用いた開発が行える．C 言語の拡張によりプログラムを記述する事ができ，簡単に GPU 上で動作させる事ができるのが特徴である.

何れの環境においても GPU 上で動作するプログラムは SPMD（Single Program Multiple Data）と呼ばれるモデルで記述する．このモデルはプログラムを 1 つ作成すると，実行ハードウェア上で数十～数億スレッドにコピーされて実行される．コピーされたスレッドは自身のスレッド ID が自動で割り振られ，プログラム中で参照することが可能なため，処理すべきデータをスレッドごとに分けることができる. 他に, GPGPU 特有のアーキテクチャに沿った実装を行うと高いパフォーマンスを得ることが可能となるが，それらの詳細については参考文献 [3]などの，GPGPU 入門書を参照されたい.

2.3. 関連研究

2.3.1. GPU 上で完結しているタイプの関連研究

GPU を用いて最短経路問題を高速化する試みは多数存在し, 例えば, 奥山倫弘らの研究 [4]では全点対最短経路の高速化(ダイクストラ法に対して)に成功している. 他には, Rafia Inam らの研究 [5]では A*アルゴリズムを GPU 上で実装した場合に, 高速化に成功する例と失敗する例を示している．しかし，これらの研究は対象となるグラフの規模が小さく，アルゴリズムの実行が GPU 上で完結している．即ち，対象となるグラフが GPU に実装されているメモリに収まる場合の研究である．このようなケースでは，CPU と GPU を同時に使用することは現実的ではない．なぜならば，CPU と GPU はメモリ空間が異なるため，相互アクセスに多大なオーバーヘッドが発生し，メモリアクセスがボトルネックとなり失敗する可能性が非常に高いためである．これらの観点から，ここで紹介した 2 つの研究は本研究のターゲットとは異なるものとなる（CPU 実行，GPU 実行を完全に切り分けて考えるべき研究である）.

他の研究では，例えば Stefan Edelkamp らの研究 [6]では様々なパズルの求解を GPU で高速化する試みをまとめており，15 パズルやハノイの塔などの置換パズルで GPU による高速化に成功している．しかし，ルービックキューブの求解の高速化には失敗している． [6]の研究はパズルの求解問題に関する研究であるが，グラフの最短経路問題に包含される点で最短経路問題の高速化の研究と言える．しかし， [4]や [5]とは異なり，ノード数が膨大である特徴がある． [4]や [5]では探索対象となるグラフ全体を GPU 上のメモリに格納し，探索アルゴリズムを走らせるという手法に対し， [6]はグラフ全体をメモリ上に格納することは不可能なため，探索対象となるパズル毎に探索アルゴリズムを設計し，GPU 上で実行して高速化を試みているという点で性質が異なる．これは置換パズルの探索における重要な性質を利用している．その性質とは，ある探索ノード（グラフの頂点）に対し考え得る操作（グラフの辺）が即座にわかる性質である．この性質が，グラフの全頂点をメモリに格納することなくグラフの巡回を行うことを可能にしている．この性質は本研究においても重要な性質である．しかし， [6]も GPU 上でアルゴリズムが完結しているという観点から，本研究のターゲットとは異なるものである.

2.3.2. CPU と GPU が協調動作するタイプの関連研究

先述のように，GPU を用いた最短経路問題を高速化する試みは多数あるものの，本研究のターゲットに適した研究事例は少ない．しかし．本研究のターゲットとして適している研究が過去に 1 つ存在し, それが H. Hayakawa らによるルービックキューブの最短解探索の GPU アクセラレーションに関する研究 [1]である． [1]では，ルービックキューブの最短解探索で広く用いられている IDA*アルゴリズム [2]を CPU で行うタスクと GPU で行うタスクに分割することで高速化に成功している． [1]の最も重要な特徴は，CPU と GPU が協調動作している点である．これは前章で紹介した関連研究とは明らかに異なる性質である． [1]は，IDA*アルゴリズムの複雑な部分を CPU が行い，GPU に適したタスクを多量に生成し，そこで生成されたタスクを GPU が担うことで高速化を実現している．ここで重要な点は，探索中に互

いに独立した多量の小規模問題を生成できる点である．この小規模問題は，小規模問題を生成したCPUが担うことも可能であり,GPUに転送してGPUで処理する事も可能である．[1]の研究では，この小規模問題を全てGPUに行わせており，CPUは1スレッドで小規模問題の生成およびGPU制御を行っていた．しかし，現代のCPUはマルチコアマルチスレッドであることが多いため，CPU 性能を使い切れていない．そこで，本研究では [1]で提案された手法を改良し，計算リソース（CPU+GPU）を効率良く使い，IDA*アルゴリズムを高速に実行できる実装手法の提案を目標とする．

3. ルービックキューブと求解アルゴリズム

3.1. ルービックキューブとは

ルービックキューブはハンガリーの建築学者である Erno Rubik が 1974 年に考案した立方体状のパズルである．立方体の各面には独立した色が設定されている．更に，立方体の各面は 3×3 に分割されているおり，分割された層を回転させる事が出来る．この回転に伴い，表面の色が置換される．ルービックキューブの初期状態は各面の色が統一された状態となっている．初期状態のキューブに対し，上記の回転操作をランダムに加え，表面の色を混合する事を“シャッフル”と呼び，シャッフルされたキューブを適当な操作により初期状態に戻す事を“解く”と呼ぶ．図 3-1 にルービックキューブの外観を示す．

ルービックキューブはあらゆる状態から 20 手（手数の定義は 3.3 参照）以内で解けることが 2010 年に T. Rokicki らにより証明されている [7].

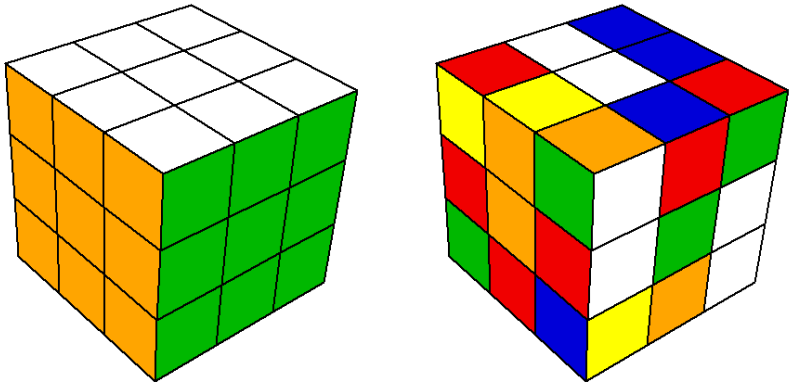

図 3-1　ルービックキューブ（左は初期状態，右は混合状態）

3.2. ルービックキューブの各部位の名称

この章では，ルービックキューブに関連する文書を記述する際，一般的に用いられている名称を記す.

3.2.1. ルービックキューブを構成する機械的な部品の名称

ルービックキューブは機械的に見ると，キューブ頂点を構成する 8 つの部品，キューブの辺（キューブの頂点部品に挟まれた部分）を構成する 12 個の部品，キューブの各面の中央を構成する 6 つの部品から構成され，それぞれ，コーナーキューブ，エッジキューブ，センターキューブと呼称する． 次項にこれらのキューブの識別法を記す.

3.2.2. ルービックキューブの各面，各部位の識別方法

ルービックキューブの各面，部品は色により個々を識別できるが，その構成色は普遍的ではない．そこで，シングマスター記法 [8]と呼ばれる，色情報を使用しない記法を導入する．この記法では，ルービックキューブが目の前に存在すると仮定した時，正面を F（front），裏

面を B（back），上面を U（up），下面を D（down），右側を R（right），左側を L（left）と各面に対し識別文字を割り当てる（図 3-2）.

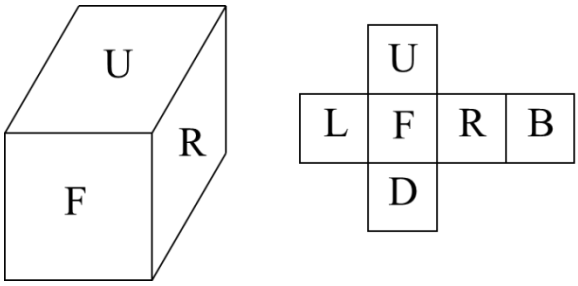

図 3-2　シングマスター記法に基づく各面の名称，右は展開図

ルービックキューブの各構成部品は，各部品にどの面が含まれるかで判断する．例えば，コーナーキューブは urf, urb，…のようにコーナーキューブを構成する 3 つの面を指定し，コーナーキューブの識別を行う．なお，urf は ruf, fru などと順序を入れ替えて記述しても同一部品を表す．エッジキューブは同様に構成する 2 面（例：rf），センターキューブは 1 面を指定する（例：f）事で，それぞれの部品の識別が可能となる（図 3-3）.

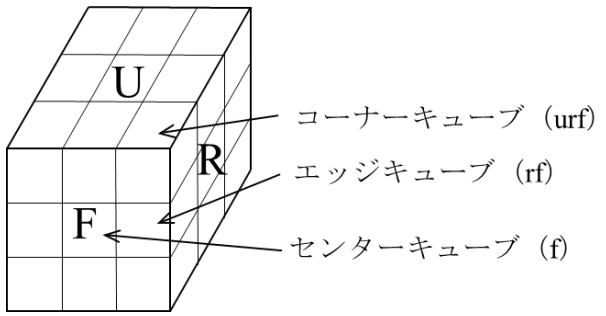

図 3-3　ルービックキューブの構成部品の名称と識別法

次に，スライスと呼ばれるルービックキューブの 1 回の操作で移動する部品群の命名法を記す．U 面の 9 個の部品を U スライス，D 面の 9 個の部品を D スライス，U 面，D 面の間に存在する 8 個の部品の層を UD スライスと呼ぶ（図 3-4）．他のスライスも同様の規則で F スライス，FB スライス，B スライス，L スライス，LR スライス，R スライスと名前が付けられる．ここで，UD スライスは DU スライスと記述しても同一部分を表す．FB スライス，LR スライスについても同様である．

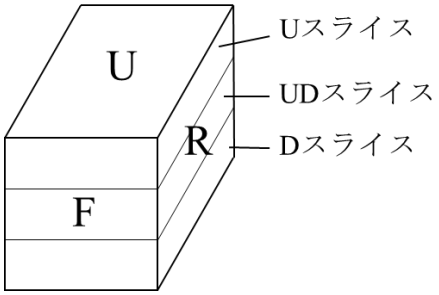

図 3-4　ルービックキューブを構成するスライスの名称

3.3. ルービックキューブの手順の表記法，手数の数え方

ルービックキューブの操作の記述はどのスライスを何度回転させたか，により定義する．F スライスを時計回りに 90°回転させる操作は F，180°回転させる操作は F2，270°回転させる操作は F’ あるいは F^{-1} と表す．UD スライスなど，中間層に対する回転操作は表層に対する回転操作 2 回に相当するため，表層に対する操作 2 回として記述し，専用記号は設けない．従って，操作は F，F2，F’，R，R2，R’，U，U2，U’，B，B2，B’，L，L2，L’ , D，D2，D’の 18 通りが定義される．手順の表記はここで定義した操作記号を，操作順序通りに並べて記述する．例えば，R の次に L の操作を加えた場合は R L と記述する．手数は各スライスに回転操作を 1 回加える事を 1 手と数える．F, F2, F’，はどれも 1 手であり，回転角度による区別は無い．この数え方を，FTM（face turn metric）と呼ぶ．回転角度を区別する QTM（quarter turn metric）と呼ばれる数え方も存在するが，本研究では扱わない．

3.4. ルービックキューブ理論の基本定理

ここでは，ルービックキューブを扱う上で重要な，キューブ理論の基本定理 [8]と呼ばれる定理を紹介する．これらの定理はデータ構造の決定時に重要な役割を果たす．

定理 1 はルービックキューブの機械構造から直ちに導かれる自明な定理である．5.3 で詳述するルービックキューブの機械構造に基づくデータ構造の設計を行う上で，この定理が重要となる．定理内に現れる“置換”，“向き”の概念は 5.3 にて詳述する．

定理 1　キューブ理論の第一基本定理

ルービックキューブの配置は次の 4 項目で決定する.

(1)エッジキューブの位置がどのように置換されたか（12!通り）

(2)コーナーキューブの位置がどのように置換されたか（8!通り）

(3)エッジキューブの向きがどの方向か（第二定理より 2^{11} 通り）

(4)コーナーキューブの向きがどの方向か（第二定理より 3^{7} 通り）

定理 2 はルービックキューブに対する操作で実現可能な“置換”に対する制約を表す．証明は参考文献 [8]参照．この定理により，ルービックキューブを表すデータ構造のデータ量が削減される．この定理を活用したデータ構造は 5.4 にて詳述する．

定理 2　キューブ理論の第二基本定理

(1)1 つのコーナーキューブの向きは残り 7 つのコーナーキューブの向きにより一意に定まる

(2)1 つのエッジキューブの向きは残り 11 個のエッジキューブの向きにより一意に定まる

(3)コーナーキューブの位置の置換の偶奇と，エッジキューブの位置の置換の偶奇は一致する

※(3)は求解不可能配置の検出に用いる．本研究では求解不可能配置は扱わないため (1)，(2)を利用する

3.5. ルービックキューブの求解アルゴリズム

ルービックキューブを解く処理は，頂点数 4.33×10^{19}，枝数 $4.33\times10^{19}\times18\div2$（1 状態に対し与えられる回転操作は 18 通り）の巨大なグラフ上での経路探索問題に等しい．最短解を求める手段はこのグラフを全探索する他なく，この全探索を効率良く行えるアルゴリズムが IDA*アルゴリズム [2]である．同アルゴリズムの解説は次章にて行う．

最短解以外の解を見つけるアルゴリズムは多数存在するが，ここでは扱わない．

4. IDA*アルゴリズムと GPU アクセラレーション

4.1. IDA*アルゴリズムの概要

Richard E. Korf は 1997 年にルービックキューブの最短解を求めるアルゴリズムとして IDA*アルゴリズム（Iterative Deepening A* algorithm）を考案した [2]．アルゴリズム自体はグラフの最短解を求めるという目的で汎用的であるが，特にルービックキューブの最短解探索において効率良く働く性質を持つ．探索方法は深さ 1 の全探索，深さ 2 の全探索，と探索する深さを 1 ずつ増やしながら，解が見つかるまで全探索を繰り返すという単純なものである．また，特徴的な点を 2 つ持つ．1 つは探索済み状態の保存をしない点である．即ち，このアルゴリズムは，探索済み状態の再探索（探索ノードの衝突）を防ぐ機構を備えていない．しかし，ルービックキューブの探索においては探索ノードの衝突がほとんど起こらないという特殊な性質を持つ．結果としてメモリアクセスが大幅に削減され（探索済み状態を保持する場合に対し），高速なアルゴリズムを実現している．もう 1 つの特徴は距離関数による枝狩りを行う点である．詳細については次項，及び， 6.1 の実装セクションにて述べる．

4.2. IDA*アルゴリズムの詳細

IDA*アルゴリズムは，最短経路を求めるために，深さ n（n は正整数）の全探索を複数回行う．探索は深さ 1 から始め，深さ 1 の全探索，深さ 2 の全探索，…と，探索する深さを 1 ずつ深くし，解の発見と共に探索を終了する．深さ n の探索で解が発見されたとき，事前に深さ 1～（n－1）の全ての全探索において解が無いことが確認されているため，解が見つかった時点での深さ n の経路が最短解であることが保証される．また，それぞれの深さ n の全探索は深さ優先探索にて行う．これは深さ優先探索が幅優先探索に比べて使用するメモリ量が少ないためである．また，それぞれの探索深度の探索結果は探索終了後に破棄する．即ち，深さ n の探索は毎回根ノード（探索対象の初期状態）から全てのノードを巡回する処理となる．多くの最短経路探索アルゴリズムは探索の途中状態を保持するが，IDA*アルゴリズムではそれを行わない．理由は原論文にて主に 2 つ述べられており，1 つは途中経過を保持するよりも捨てた方が CPU のキャッシュを効率良く使うことができ，実験的に高速であること．もう 1 つは途中経過のノード数は十数億以上に達するため，同アルゴリズムの考案時点（1997 年）のハードウェア‡ではメモリ不足により実装不可能であったためである．

また，探索途中に距離関数による枝狩りを行う．距離関数は，解への到達まで少なくとも必要な探索の深さを返す関数である．例えば，深さ 6 の全探索の最中に，深さ 4 のノードが距離関数をコールし，距離関数からの戻り値が 3 だとする．この場合，深さ 4 のノードは深さ 6 の全探索のためにはあと深さ 2 の探索をしなければならないが，距離関数から少なくともあと深さ 3 の探索をしないと解が無いという情報を返却されていることを意味する．この場合, 該当ノードから残りの深さ 2 の探索を行うことが無意味なため, 枝狩りをする（図 4-1）.

‡ 1997 年当時の高性能 PC は Intel®Pentium II 350MHz　RAM 64MB のような構成

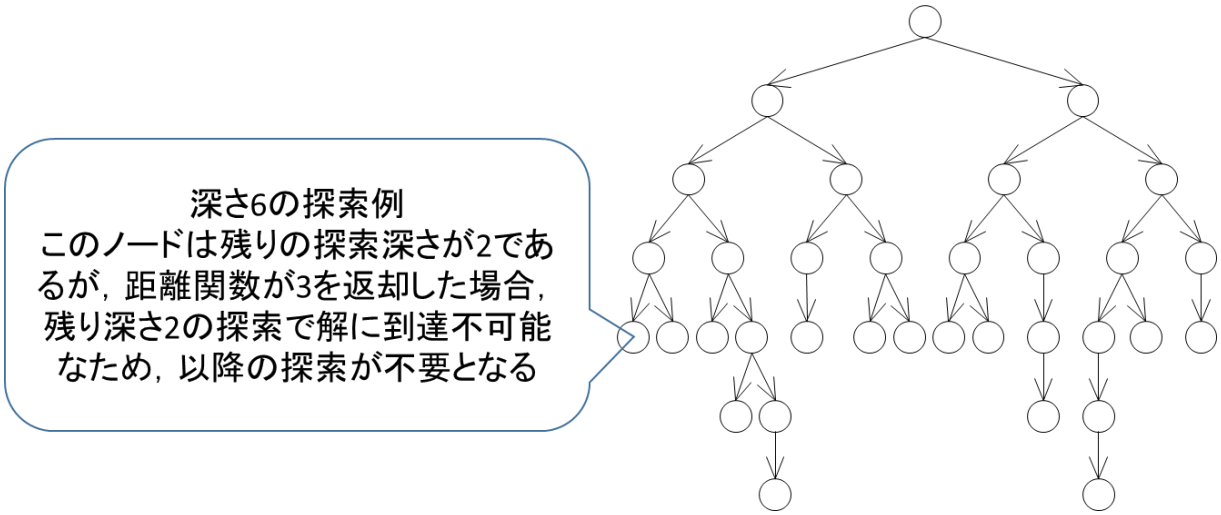

図 4-1　枝狩りの例

4.3. [1] による GPU アクセラレーション（Simple Separated GPU Acceleration）

IDA*アルゴリズムは H. Hayakawa [1]により GPU アクセラレーションに成功している．[1]の主張によれば，CPU 1 core に対し 21 倍，4core に対し 5 倍の性能向上に成功している．[1]では，CPU にて特定の深さまでの探索を行い，途中経過を保存する．この保存された途中経過を GPU に転送し，並列処理を行うことで高速化を実現している． [1]より引用した処理の簡略図を図 4-2 に示す．

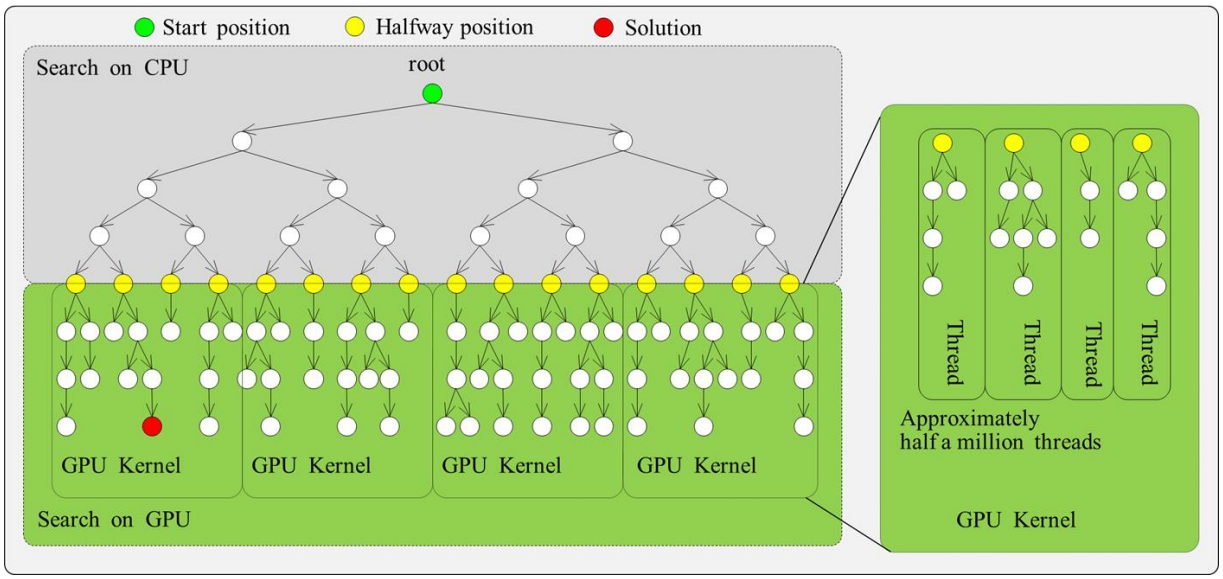

図 4-2　[1]による IDA*アルゴリズムの GPU アクセラレーション（ [1]より引用）

図中の灰色部分が CPU 処理であり一定の深さ（図中では深さ 4）の探索を行い，探索途中のノード（黄色ノード）を収集する．図では省略されているが，この探索途中のノードは順次キューに格納され，一定数（50 万個程度）集まり次第 GPU に転送し，GPU カーネルを実行する構造になっている．ここで，GPU カーネルとは GPU のプログラム実行単位のことであ

り，GPU プログラムの開始から終了するまでを指す．GPU プログラムはハードウェアの性質上長時間の連続実行ができない．例えば，Linux 上で CUDA を実行する場合は，1 回のカーネル実行が 10 秒以内で終了しなければならない制約を持つ．このため，[1]では 1 回のカーネル実行で処理する中間ノードを 50 万ノード（50 万スレッド）程度に制限し，1 回のカーネル実行が 1 秒以内で完了するよう調整されている．

[1]の手法は簡潔に CPU 処理と GPU 処理を分離して高速化を実現しているため，本稿では SSGA（Simple Separated GPU Acceleration）と呼称する．

4.4. SSGA（Simple Separated GPU Acceleration） [1]の問題点

ここで紹介した先行研究である SSGA には下記 2 つ問題点がある．この 2 つの問題点のうち，特に（2）の問題点の解消を本研究の課題とする．

（1）使用されていたハートウェアが古い

[1]で使用されたハードウェアは CPU が Core i5 3570K @3.4GHz, GPU が GeForce GTX 570 であり，メインメモリ 32GB，ビデオメモリ 1.2GB という構成である．この構成では特にビデオメモリの容量の少なさに問題がある．近年の GPU ではビデオメモリが 8GB 以上実装されている例も珍しくなく，また，IDA*アルゴリズムをルービックキューブに適用した場合，性能はメモリ容量に大きく依存する．そこで，本研究ではビデオメモリを 4GB 実装した GPU を用いて追試を行った．

（2）計算リソースを全て使い切れていない

[1]では GPU 負荷は十分であるが，CPU スレッドが 1 スレッドであり，CPU の他のスレッドが処理を行っていない．この CPU の残りスレッドにも処理をさせるようにプログラムを実装することができれば更なる性能向上が実現可能であると考えられる．本研究では，3 通りの実装手法を提案する．

5. ルービックキューブのデータ構造

5.1. 概要

ルービックキューブを扱うプログラムを作成するためにデータ構造の設計を行う．ここで定義するデータ構造はルービックキューブに関わる研究におけるプログラム中で広く用いられるものであり，[9]でも同様の構造が使用されている．しかし，web サイトでの紹介などに限られ，いつでも参照可能とは限らないため，本稿でもまとめておく．ここで紹介するデータ構造は Kociemba の web サイト [9]を参考にしたものである．5.2 でルービックキューブの最も基本的な表現，5.3 でルービックキューブの機械的な構造の表現，5.4 で探索アルゴリズム中のデータ構造を説明する．これらのデータ構造は，相互変換可能なデータ構造であるが，ルービックキューブの問題入力や GUI によるルービックキューブ操作ではデータ管理の簡潔さ，解探索アルゴリズムの中では計算量の少なさが重要になるため，ルービックキューブを管理するデータ構造を階層的に複数用意する必要がある．

5.2. シール単位レベルのデータ構造

5.2.1. 表現法

この表現法は，ルービックキューブの表面の情報をそのまま格納する．即ち，ルービックキューブの表面の 48 枚のシールの位置を 48 個の変数で管理する．表面中央のシールは移動しない（中間層に対する操作は行われない）ため管理する必要はない．このデータ構造を使用した場合の各シールの番号付けの例を図 5-1 に展開図にて示す．なお，図 5-1 では，全てのシールに異なる番号を付けているが，色毎に番号を分け，計 6 つの番号で管理しても良い．C 言語にて実装を行う場合は図 5-2 のような構造体宣言となる．

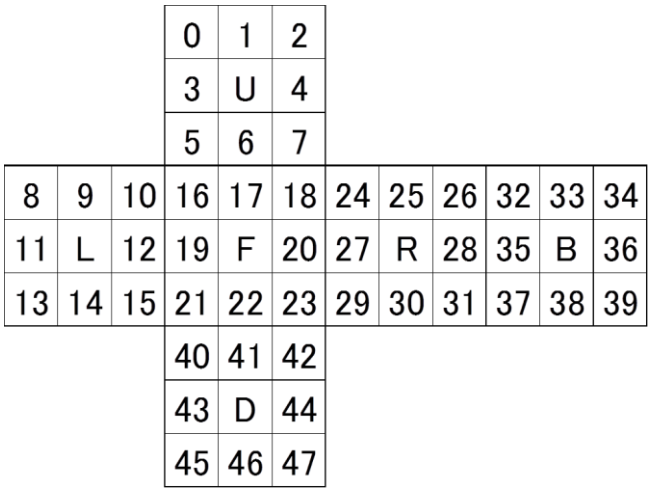

図 5-1　シール毎の番号付けの例

```
typedef struct _facelet_level_structure__{
  /* キューブの表面情報．face[1] == 0がtrueの場合1番の場所に0番の色があることを意味
     する */
  int8_t face[48];
} facelet_cube;
```

図 5-2　C 言語におけるシール単位レベルの構造体宣言例（48 byte 要する）

5.2.2. 回転操作の記述

このデータ構造に対する回転操作は各変数に対する置換操作となる．キューブの初期状態 S を

S=(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47)

と表すとする．数字の位置がキューブ上での位置であり，格納されている値がシールの番号である．初期状態 S に対し，操作 F を加えた結果を G とすると，

G=(0,1,2,3,4,15,12,10,8,9,40,11,41,13,14,42,21,19,16,22,17,23,20,18,5,25,26,6,28,7,30,31,32,33,34,35,36,37,38,39,29,27,24,43,44,45,46,47)

となり，与えた操作に応じて格納されている値が置換される．他の操作についても同様に置換により表現でき，1 操作につき 20 個の変数の置換が行われる．

5.3. 機械構造レベル

5.3.1. 表現法

この表現法では，キューブの状態はコーナーキューブとエッジキューブで独立に考える（キューブ理論第一基本定理（3.4）に基づきキューブの状態を表現する）．コーナーキューブ 1 つは 8 カ所の取り得る位置を持ち，1 カ所につき 3 通りの方向が考えられる．そこで，位置情報に関しては，各コーナーキューブ位置と各コーナーキューブに識別番号を付け，何番の位置に何番のキューブが存在しているか，によりキューブの位置を表す．これに加え，各コーナーキューブが基準からどの程度捻れているか，によりキューブの向きを表す．キューブの捻れ量を表現するには，各コーナーキューブの位置に角度 0 の印を付け，コーナーキューブ自身にも角度 0 の印を付ける．捻れ量は，該当箇所のコーナーキューブを反時計回りに 120° 回転させる操作を何回行えば，捻れ量を示す印が重なるか．によって表す．これを図 5-3 に示す．x 印が回転量 0 を表す印（絶対位置）であり，o 印がコーナーキューブに付けられた印である，エッジキューブについてもコーナーキューブと同様である．エッジキューブ 1 つは 12 カ所の取り得る位置を持ち，各位置にて 2 通りの向きを持つ．これをコーナーキューブと同じ要領で表現する．これを図 5-4 に示す．

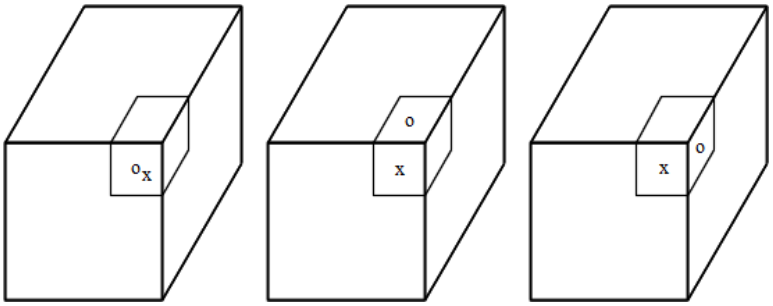

図 5-3　コーナーキューブのねじれ量の表現（左からねじれ量 0,1,2）

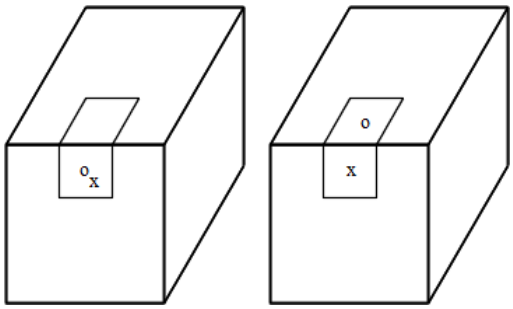

図 5-4　エッジキューブのねじれ量の表現（左からねじれ量 0,1）

5.3.2. データ構造

ここで示した表現法をデータとして扱うためには，次の 40 個の変数が必要となる．C 言語にて実装する場合は図 5-5 のような構造体宣言となる．

5.3.2.1. コーナーキューブの位置を格納する変数

コーナーキューブの位置は 8 カ所あるので，8 つ変数が必要となる．格納方法は，キューブの位置毎に変数を用意し，該当箇所にどのキューブが存在するか，で表す．

5.3.2.2. コーナーキューブの向きを格納する変数

コーナーキューブの向きを格納する変数は,位置を格納する変数と同様に 8 個必要である．これらの変数は，どの位置のキューブがどの程度捻れているか，という情報を格納する．

5.3.2.3. エッジキューブを格納する変数

エッジキューブについてはコーナーキューブと同様に，12 個の変数を位置格納用，向き格納用の 2 通り（計 24 個）用意する．

```
typedef struct __cubie_structure__{
  /* コーナーキューブの位置. corner_position[1] == 0がtrueの場合は1番の位置に0番のコーナーキューブが存在するという意味 */
  int8_t corner_position[8];
  /* コーナーキューブの捻じれ. corner_position[1] == 2がtrueの場合は1番のコーナーキューブが基準から2捻じれているという意味 */
  int8_t corner_orientation[8];
  /* エッジキューブの位置 */
  int8_t edge_position[12];
  /* エッジキューブのねじれ */
  int8_t edge_flip[12];
} cubie_cube;
```

図 5-5　C 言語における機械構造レベルの構造体宣言例（40 byte 要する）

5.3.3. 回転操作の記述

このデータ構造での回転操作は，5.2 の構造と同様に，変数の置換で表現できる．各操作は 4 カ所のコーナーキューブ，4 カ所のエッジキューブを置換するため，16 個の変数の置換として表現できる．

5.4. 数値レベル

5.4.1. 表現法

この表現法は，5.3 の構造を少ない数の数値に置き換え，データ量の圧縮と回転操作の処理量の削減を図ったものである．具体的には，コーナーキューブの位置情報，向き情報，エッジキューブの向き情報をそれぞれ 1 変数で表し，エッジキューブの位置情報を 3 つの変数で表し，計 6 つの変数にてルービックキューブの状態を表現する．以下にその詳細を記す．

本データ構造を C 言語にて実装する場合，図 5-6 のような構造体宣言となる．図より，ルービックキューブの 1 状態が 12 byte という非常にコンパクトな容量で表現できることがわかる．このデータ構造が，ルービックキューブ研究におけるプログラム実装でのスタンダードなデータ構造である．

```
typedef struct __coordinate_level_structure__{
        uint16_t corner_position; //コーナーキューブの位置
        uint16_t corner_orientation; //コーナーキューブのねじれ
        uint16_t edge_flip; //エッジキューブのねじれ
        uint16_t edge_position_ud; //エッジキューブ（UDスライス）の位置
        uint16_t edge_position_lr; //エッジキューブ（LRスライス）の位置
        uint16_t edge_position_fb; //エッジキューブ（FBスライス）の位置
} coord_cube;
```

図 5-6　C 言語における数値レベルの構造体宣言例（12 byte 要する）

5.4.1.1. コーナーキューブの位置

コーナーキューブは 8 個のキューブ其々が 8 カ所のどこかに配置されるので，8! 通りの組み合わせを持つ．つまり，5.3.2.1 での 8 変数の状態は 8!通りの組み合わせしか持たない．即ち，5.3.2.1 の 8 変数の状態と 0 ～ (8! － 1)の整数値との間で全単射を構成することができる．従ってコーナーキューブの位置を 1 つの変数で管理する事が可能となる．5.3.2.1 の 8 つの変数の状態を 1 つの数値に変換する手段は, 変数の状態と 0 ～ (8! － 1)の整数値との間の全単射が構成されればどのような方法でも問題無いが，階乗進数を用いた実装法を 5.4.3 に示す．

5.4.1.2. コーナーキューブの向き，エッジキューブの向き

コーナーキューブの向きは 3.4 のキューブ理論第二基本定理より, 3^7 通りの組み合わせが存在する．よって，位置情報と同様に，向きを表す 8 個の変数の状態と 0 ～ (3^7 － 1)の範囲の整数値との間で全単射を構成する方法を考える．これは 3 進数を用いれば容易である．キューブ理論第二基本定理より 8 個の変数のうち 1 つの値は無視する事ができるので, 7 つの変数を 7 桁の 3 進数とみなし，2 進変換したものをコーナーキューブの向き情報として扱う．

エッジキューブについても同様であり，向きを表す 12 個の変数のうち，11 個を 11 桁の 2 進数として扱えば良い．

5.4.1.3. エッジキューブの位置

エッジキューブは位置が 12! 通りと取り得る状態数が非常に多いので，UD スライス，LR スライス，FB スライスの 3 組に分割し，それぞれのスライスに属するエッジキューブの位置情報をそれぞれ 1 つの数値に対応させる．各中層スライスのエッジキューブ位置の情報は 12×11×10×9 = 11880 通り存在する．これらの位置情報はコーナーキューブの位置情報と同様, 階乗進数を用いて表現する事が出来る．

5.4.2. 回転操作の記述

回転操作は各変数に対する表引き操作で実装される．たとえば，コーナーキューブの位置の場合，8! = 40320 通りの組み合わせが考えられる．この 40320 通りの任意の状態に対し，18 通りの回転操作の何れかを加えると，40320 通りのどれかの状態に遷移する．この遷移の表を予め用意し，回転操作の動作を定義する．他の 5 要素についても同様である．このデータ構造では，回転操作は 6 回のテーブル参照となる．

5.2, 5.3 の構造における回転操作と，本項での遷移表による回転操作の関係を図 5-7 に示す．図中の indexA, indexB は本データ構造における 6 つの数値を表し，TwistID は回転操作（R, F, …）を表し，0～17 の値をとる．本項の構造では，コーナーキューブの位置，コーナーキューブの向き，エッジキューブの向き，エッジキューブの位置（3 つ）の計 6 つの要素について，図中のインデックス変換（indexA→indexB）のテーブル（TwistTable）を用意する事で回転操作を実装する．

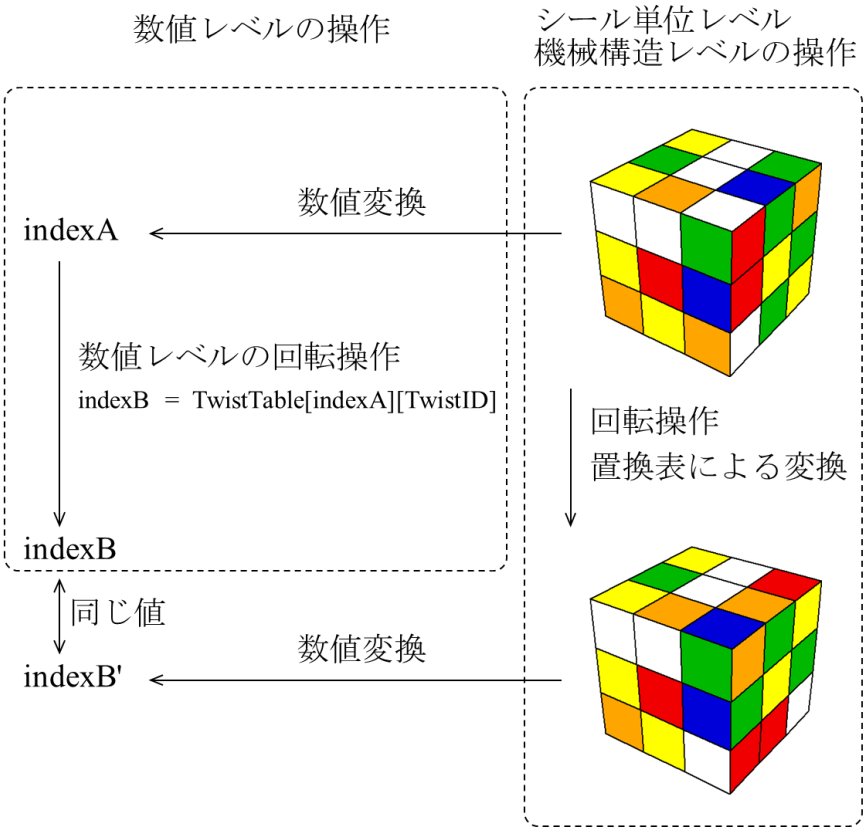

図 5-7　各データ構造の回転操作の相互関係

5.4.3. 順列を数値に変換するアルゴリズム

ここでは，順列を数値に変換する方法のうち，階乗進数を用いた方法を紹介する．階乗進数とは，図 5-8 のような数値の表記法であり，順列を決定付ける操作を一意に，かつ無駄無く記述出来る．順列は，候補となる集合から適当な元を選び，候補から除く，という操作を繰り返す事で定まる．たとえば，4 個の値(0, 1, 2, 3)の順列 P = (2, 1, 3, 0)を階乗進数で表現すると，順列 P の最左の 2 は 4 つの数字のうち左から 3 番目の値なので“2”，P の左から 2 つ

目の 1 は残された 3 つの値(0, 1, 3)の左から 2 番目なので“1”，P の 3 は残された 2 つの値(0, 3)のうち左から 2 番目なので“1”，P の 0 は残された 1 つの値(0)のうち，左から 1 番目なので“0”，引用符で括られた値を並べると，“2110”となり，この値が順列から導かれた階乗進数での値となる．この階乗進数の値を 10 進変換すると，2×3! + 1×2! + 1×1! + 0×0! = 15 となり，順列 P は 15 という 0 ～ (4! － 1)の範囲内の値に一意に変換される．この様に，階乗進数は下位から何桁目か，で順列決定時の集合の要素数が表現でき，その桁の値からどの要素を選び出したか，を表現可能なため，順列の状態に適当なインデックスを割り振る用途に適している．

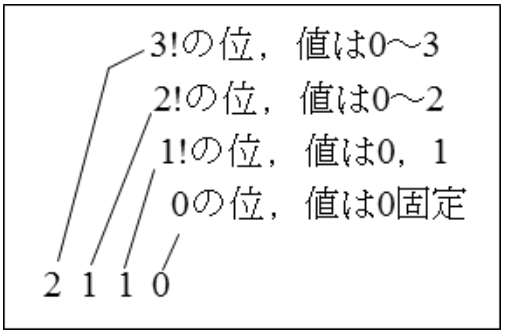

図 5-8　階乗進数の例

図 5-9 に 8 個の値の順列を数値に変換するアルゴリズム，図 5-10 に 12 個の値のうち 4 つを並べた順列を数値に変換するアルゴリズムを示す．図 5-9 中の src[8]には，(1, 0, 7, 5, 6, 4, 3, 2）のような 0～7 の数値を並べた順列を入力し，図 5-10 中の src[4]には，(11, 5, 7, 1）のような，0～11 から 4 つを選んだ順列を入力する．戻り値として順列に一意に対応する数値が得られる．これらのアルゴリズムは，候補となる集合から元を選ぶ操作をビット列と population count（図中では関数 pop()と表記）により表現している．population count は整数値に含まれる 1 のビットの数を数える手続きである．候補の集合を表すビット列は，各ビットが残された候補の値を表す．population count により候補から選んだ元が候補中の何番目かを調べる．選ばれた数字を示すビットより下位に存在する 1 のビットの合計数が，選ばれた数字が何番目か，に一致する．次に，図 5-9 のアルゴリズムの詳説に移る．まず，変数 flags は候補となる集合を表し，下位から n 番目のビットが 1 なら集合に（n）が含まれる事を意味する．L8，L9 の操作は候補の集合から選ばれた数字に対応するビットを 0 にする操作である．L10 にて選ばれた数字より下位に存在する 1 のビットを数える．ここで注意すべきなのは，flags と（tmp － 1）との論理積をとっている点である．tmp は L8 にて，選ばれた数字に対応するビット位置のみ 1 となる値が格納されている．この値から 1 を引く事により，元の 1 のビットより下位のビットが全て 1 となるマスク生成が行える．このマスクと flags との論理積を取り，その上で population count を行う事で，目的の値を得る事が出来る．また，population count の処理方法は多数存在するが，ソフトウェア実装においては最も高速と考えられる [10]に記されている方法を使用している（図 5-11）．このようにして，順列を一定範囲内の数値に置き換える事が可能となる．

```
1:    int permutation_corner(int src[8]){
2:      int ret, i;
3:      unsigned int flags = 0xFFFFFFFF;
4:      unsigned int tmp;
5:      ret = 0;
6:      for(i = 0; i < 8; i++){
7:        ret *= (8 - i);
8:        tmp = 1 << (src[i]);
9:        flags ^= tmp;
10:       ret += pop(flags & (tmp-1));
11:     }
12:     return ret;
13:   }
```

図 5-9　8 個の数字の順列を数値に変換する関数

```
1:    int permutation_4edge(int src[4]){
2:      int ret, i;
3:      unsigned int flags = 0xFFFFFFFF;
4:      unsigned int tmp;
5:      ret = 0;
6:      for(i = 0; i < 4; i++){
7:        ret *= (12 - i);
8:        tmp = 1 << (src[i]);
9:        flags ^= tmp;
10:       ret += pop(flags & (tmp-1));
11:     }
12:     return ret;
13:   }
```

図 5-10　12 個の数字のうち 4 つを並べた時の順列を数値に変換する関数

```
1:    unsigned int pop(unsigned int n){
2:      n = (n & 0x55555555) + (n >> 1 & 0x55555555);
3:      n = (n & 0x33333333) + (n >> 2 & 0x33333333);
4:      n = (n & 0x0f0f0f0f) + (n >> 4 & 0x0f0f0f0f);
5:      n = (n & 0x00ff00ff) + (n >> 8 & 0x00ff00ff);
6:      return (n & 0x0000ffff) + (n >>16 & 0x0000ffff);
7:    }
```

図 5-11　[10]による値 n に含まれる 1 のビットを数える関数

6. 新しいハードウェアによる先行研究（SSGA）の追試

ここでは，SSGA [1]のアルゴリズムを本稿で使用する実験環境に適した方法で実装し，GPU アクセラレーションの効果を追試，検証する．

6.1. ルービックキューブにおける IDA*アルゴリズムの実装

IDA*アルゴリズムを実装するためには距離関数の設計が必要である．加えて，ルービックキューブの探索をより効率良く行うために，探索ノードの衝突を回避する方法を実装する．ここでは，探索ノードの衝突回避方法と，距離関数の設計について詳述する．

6.1.1. 探索ノードの衝突回避方法

ここではルービックキューブにおける IDA*アルゴリズムを実装する上で，探索ノードの衝突数を減らす方法を述べる．なお，ここに記述する方法は [2]を始めとするルービックキューブ関係の研究では広く一般に用いられている手法である．

探索ノードの衝突を減らすために，探索時に以下の 2 つのルールを追加する．

（1）同一面を 2 回連続して操作しない
（2）平行な 2 面に関して順序制約を設ける

まず，(1) は探索時に必要不可欠なルールである．同一面を 2 回以上連続操作するのは手数の定義において 1 手と数えられるため，無意味な探索となる．従って，このルールを追加して予め無意味な探索の出現を予防する．(2) は探索ノードの衝突数削減のためのルールである．例えば，L R の順に操作した場合と R L の順に操作した場合は結果が同一である（探索ノードが衝突する）．従って，L R の順序は許可し，R L の順序は不可とすることによって，この単純な探索ノード衝突を防ぐ．U D 面，F B 面にも同様のルールを付加する．このルールを追加した場合に，距離関数を用いずに探索を行うと表 6-1 のようなノード数となる．表から探索深度 10 程度まではノードの衝突が 5%程度に抑えられていることがわかる．このように，探索に単純な順序のルールを付加するだけで探索ノードの衝突をほとんど防ぐことが可能である事実が，IDA*アルゴリズムでは探索経過を保持しない理由の 1 つとなっている．

表 6-1　探索ノード数と実際のルービックキューブの状態数

探索深度	探索ノード数	実際のルービックキューブの状態数	衝突割合(%)
1	18	18	0.00
2	261	261	0.00
3	3501	3501	0.00
4	46755	46740	0.03
5	624123	621648	0.40
6	8331111	8240086	1.10
7	111207591	109043122	1.98
8	1484451135	1441386410	2.99
9	19815150303	19037866205	4.08
10	264501924111	251285929521	5.26

6.1.2. 距離関数の設計

距離関数は解まで”少なくとも d 手”という情報を返す関数である．この関数を用いて，深さ n 手の探索において，探索中のあるノードに対し，“このノードを深さ n まで探索しても解が無い”，という情報を検出し，該当ノードの探索を打ち切る．

ルービックキューブ探索において，距離関数はテーブルであり，テーブルには，簡略化されたルービックキューブに対する全探索結果が格納されている．例えば，簡略化されたキューブとしてコーナーキューブを選ぶと仮定する．コーナーキューブは位置が 8! 通り，向きが 3^7 通りあり，合計 88179840 状態をとりうる．この全状態に対し，初期状態からの距離を全探索してテーブルに格納する．すると，このテーブルは任意の状態のキューブに対しコーナーキューブを揃えるために要する最短手数を表すが，この手数はキューブ全体を揃えるために最低限必要な手数を与えているとみなす事も出来る．即ち，距離関数の要求である“解まで少なくとも必要な手数 d を得る”という性質を満たす．

完全な距離関数を構成するためには，ルービックキューブ全状態に対して距離を計算する必要があるが，そのためにはデータの記憶領域として 4.33×10^{19}byte（1 状態 1byte として）が必要となり現実的ではない．よって，距離関数用のテーブルを構成するためには，メモリに収まる条件下で簡略化されたキューブを選択する必要がある．

図 6-1 に実験用に実装した簡略化されたルービックキューブを示す．このキューブは 53210234880 通りの状態数を持ち，解から最も遠い状態まで 13 手である．従って，1 状態 4bit で格納することができ，26.5GB 程度の領域があればテーブルに格納することができる．しかし，GPU のメモリは 4GB のため，このテーブルをそのまま格納することはできない．そこで [9]に示されている方法でテーブルサイズの圧縮を行った．詳しい圧縮方法は [9]に記されているのでここでは結果のみ記す．図 6-1 のキューブは UD スライスのエッジキューブの状態数が 12×11×10×9＝11880 通りであるが，キューブの対称性を考慮することで考えるべき状態数をおよそ 1/8 の 1560 通りに減らすことができる．これにより，テーブルに格納すべき状態数を 53210234880 から 6987202560 まで減らすことができ，1 状態 4bit で格納して 3.5GB 程度の容量で実装することができる．

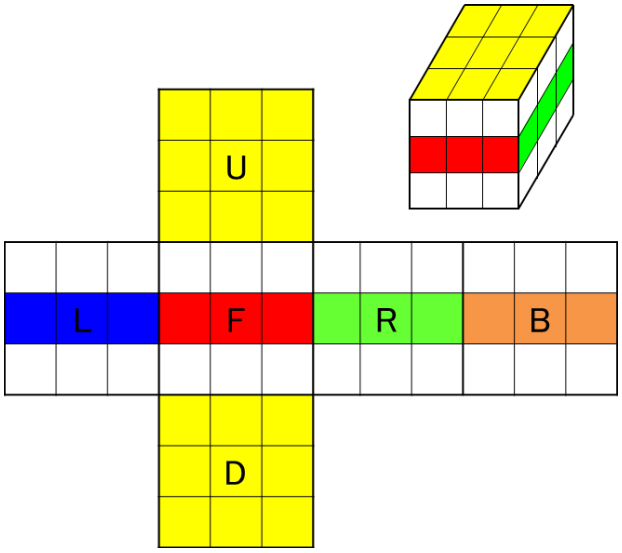

図 6-1　距離関数用の簡略化されたキューブ

表 6-2　図 6-1 のキューブの解までの手数の分布

解までの手数	分布
0	1
1	4
2	24
3	232
4	2755
5	34240
6	426329
7	5208707
8	60958527
9	617583980
10	3436278045
11	2843038764
12	23670924
13	28

表 6-2 に 6987202560 通りの状態の解までの手数分布を示す．この表の分布から，この距離関数は多くの場合（90%程度），10 か 11 を返すという事実がわかる．また，手数の最大値が 13 であることから，1 状態を 4bit で格納すればよいということもわかる．

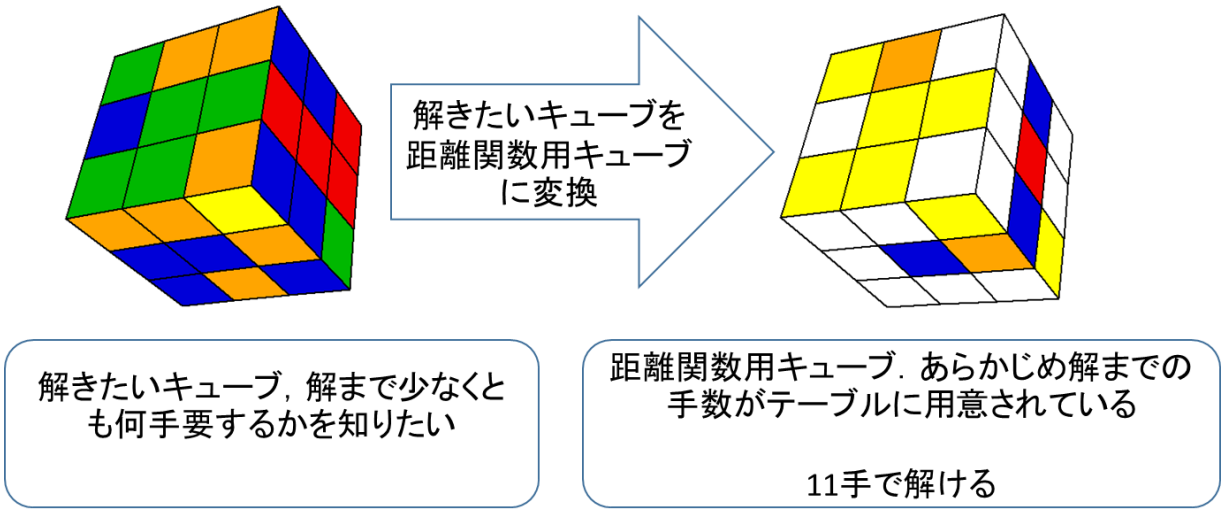

図 6-2　距離関数の使用方法

図 6-2 に距離関数の使用方法を示す．図の左側が解きたいキューブであり，解まで少なくとも何手要するかを知りたい．そこで，解きたいキューブを図 6-1 に示したキューブにマッピングし，距離関数のテーブルにアクセスして解まで少なくとも必要な手数を得る．

ここで，この距離関数を 3 方向から適用することを考える．図 6-1 で示した簡略化されたキューブは UD スライスのエッジキューブ，及びコーナーキューブの向きにより定義されているが，これは LR スライス，FB スライスで定義することもできる．そこで，図 6-2 に示した解きたいキューブから距離関数用キューブへのマッピングを，図 6-2 のように解きたいキューブの UD スライスをそのまま簡略化されたキューブの UD スライスにマッピングするだけでなく，解きたいキューブの FB スライスまたは LR スライスを簡略化されたキューブの UD スライスにマッピングすることもできる．これにより，距離関数からは 3 つの値を得られる．得られた 3 つの値から最も大きな値を採用することで，距離関数をより効率良く働かせることができる．この距離関数を 3 方向から適用する方法は [1]や [2]でも同様に使われている手法である．

更に，先行研究 [1]で示された工夫を加える．上記で示した距離関数は距離 0 を返却したときにそれが解きたいキューブで解であるとは限らない．上記の距離関数は，エッジキューブが全て正しい位置で正しい方向, コーナーキューブが全て正しい方向である場合に 0 を返す．すなわち，距離関数が 0 を返した場合にコーナーキューブの位置がずれている場合が考えられる．この問題を解消するため，さらにもう一つ距離関数用テーブルを用意する．もう一つの距離関数は図 6-3 のキューブを用いる．このキューブは 88179840 通りの状態数を持ち，

表 6-3 に示す分布を持つ．このキューブは状態数が少ないため，1 状態 4bit で格納し，44MB のテーブルとして実装する．

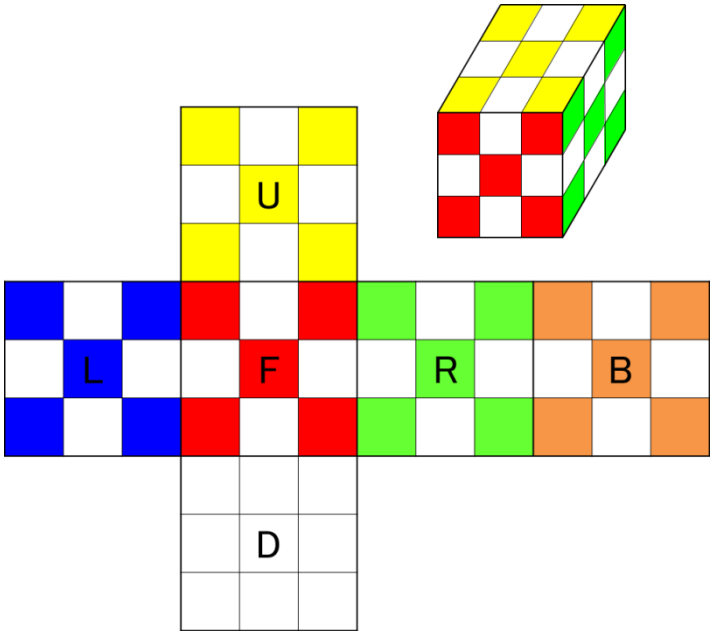

図 6-3　距離関数用の簡略化されたキューブ（2）

表 6-3　図 6-3 の距離関数用キューブの分布

解までの手数	分布
0	1
1	18
2	243
3	2874
4	28000
5	205416
6	1168516
7	5402628
8	20776176
9	45391616
10	15139616
11	64736

表 6-3 からわかるように，この距離関数は高々11 までの値しか返さないため，図 6-1 のキューブにより構成された距離関数と比較すると性能が悪い．しかしこの距離関数はコーナーキューブの位置，向きが共に揃った状態で 0 を返す．即ち， 図 6-1 により構成された距離関数と併用することにより，距離関数で 0 を返すことと，ルービックキューブが解けたことを同一の意味にすることができる．また，距離関数を複数併用することで，探索の高速化が実現できたことが [1]により示されているため，今回の追試においても同様の手法を採用することにした．

6.1.3. 探索アルゴリズムの実装

IDA*アルゴリズムのコアは深さ優先探索である．この深さ優先探索は再帰関数を用いて実装されるのが一般的である．しかし，GPU は再帰関数を実行することができない．先行研究[1]では言及されていないが，再帰関数を使わない深さ優先探索アルゴリズムを実装しなければならない．ここでは，実際に実装したソースコードを簡略化したものを載せる．探索ノードを図 6-4 のように宣言し，図 6-5 のようなソースコードにより深さ優先探索を実現した．図 6-4 の L2 に出現する構造はどのような構造でも良いが， 5.4 にて示した数値レベルの構造が適しており，本実装でも数値レベルの構造にて実装した．

図 6-5 のアルゴリズムは，問題となるキューブ（cube），探索深度（search_depth），結果格納用配列（result）を引数として与える．そして，深さが search_depth の木を巡回し，最深部のノードに解が存在すれば，その手順が得られる．また，このアルゴリズムは，深さ（search_depth）手に解が存在する場合に解を発見できるため，深さ n 手の探索をするためには，深さ(n-1)手の探索を終えて，深さ(n-1)以内に解が無い事を予め確認しておく必要がある．そのため，search_depth の値は 1 から順次増やしていかなければならない．

この探索アルゴリズムは非常にシンプルであり，L28 と L35 の continue を無いものと考えると，長さが（search_depth）に等しい手順を全て巡回するアルゴリズムとなる．しかし，手順を全て巡回する方法では限界があり，効率を高める方法が必要となる．その方法が L27 と L34 にある条件分岐である．L27 は 6.1.1 にて示した探索ノード衝突回避ルール，L34 が 6.1.2 にて示した距離関数による枝狩りである．なお，L16 にて解けているかテストするコードがあるが，6.1.2 にて示した距離関数を用いた場合は距離関数が 0 を返した時点で解けているのでこのコードは不要となる．しかし，一般に距離関数が 0 を返したことと，解けていることとは同値ではないので図中に記している．

```
1:  typedef struct __search_node__{
2:    cube_structure cube;//キューブの状態を表す
3:    int remain_depth;//残りの探索深度を表す
4:    int move;//加えた操作を表す
5:  }search_node;
```

図 6-4　探索ノードの構造体宣言例

```
1:    int search_tree(cube_structure cube,int search_depth,int result[20]){
2:      search_node *p_node;          /* 探索中ノードを指すポインタ */
3:      search_node node_array[21];/* 探索用配列 */
4:      int tw;                       /* 次の探索ノードに移る時に加える操作 */
5:
6:      node_array[0].cube = cube;    /* 配列の0番に問題の状態を格納 */
7:      node_array[0].move = INVALID;/* 0番目は何も操作が加わっていない */
8:      node_array[1].move = -1;      /* 未探索を表す */
9:
10:     p_node = node_array;        /* ポインタに配列の先頭を代入 */
11:
12:     p_node[0].remain_depth = search_depth;/* 残りの探索深度 */
13:
14:     while(p_node >= node_array){
15:       if(p_node[0].remain_depth == 0){          /* 木の最深部に到達 */
16:         if(solved(p_node[0].cube)){             /* 解けているかテスト */
17:           copy_move_histry(node_array,result);/* resultに操作履歴をコピー */
18:           return SOLVED;
19:         }
20:         p_node--;                               /* 直前のノードに戻り探索続行 */
21:       }else{
22:         for(tw = p_node[1].move + 1; tw < N_TWIST; tw ++){
23:           /* 次のノードは残りの探索深度が1減る */
24:           p_node[1].remain_depth = p_node[0].remain_depth - 1;
25:
26:           /* 直前の手と，加える手を見て，明らかに不要な手は探索しない */
27:           if(invalid_move(p_node[0].move,tw)){
28:             continue;
29:           }
30:           /* 現在のキューブに回転操作を加え，次のノードのキューブの状態を生成 */
31:           p_node[1].cube = twist(p_node[0].cube,tw);
32:
33:           /* 残りの探索深度と距離関数の値を比較，不要な枝なら次へ */
34:           if(p_node[1].remain_depth < distance(p_node[1])){
35:             continue;
36:           }
37:           /* 有効な枝だと判断されたので，適用した操作を保存 */
38:           p_node[1].move = tw;
39:           break;/* 次のノードへ */
40:         }
41:         /* 直前のループが最後まで回る → そのノードの探索終了 */
42:         if(tw == N_TWIST){
43:           p_node--;/* 直前のノードに戻る */
44:         }else{
45:           p_node++;/* 次のノードに進む */
46:           p_node[1].move = -1;/* 未探索を表す */
47:         }
48:       }
49:     }
50:     return NOT_SOLVED;
51:   }
```

図 6-5　再帰関数を使わない深さ優先探索の実装例

6.1.4. 先行研究と追試実装の差異

ここで，先行研究［1]での環境，実装と追試での環境，実装の差異を表 6-4 にまとめる．大きな違いは GPU のメモリ量と距離関数の実装方法である．先行研究は GPU のメモリが少なく，距離関数が小規模（図 6-6）な点である．対し，追試実装では GPU のメモリは 3 倍以上あり，距離関数もより大規模なものを実装した．また，追試実装側では GPU の台数が 2 台となっている．

表 6-4　先行研究と追試実装の差異

	先行研究［1]	追試実装
CPU	Core i5 3570K @3.4GHz	Core i7 4770K @ 3.8GHz
RAM	32GB	32GB
GPU	GeForce GTX 570	GeForce GTX 760 GeForce GTX 970
VRAM	1.2GB	4GB + 4GB
距離関数	図 6-6 と図 6-3 の併用 対称性の考慮なし，1.1GB	図 6-1 と図 6-3 の併用 対称性の考慮有り，3.5GB

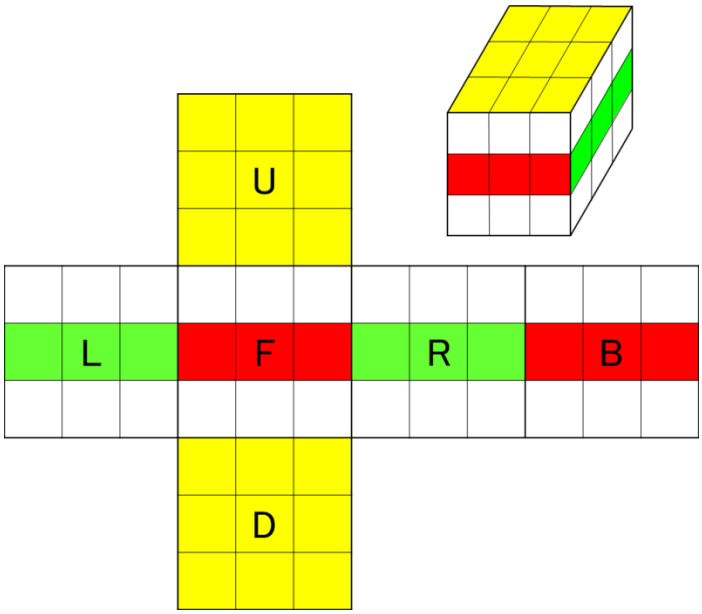

図 6-6　先行研究で使われた距離関数用キューブ

6.2. GPU 実行のパラメータチューニング

先行研究と異なる条件で SSGA の実装を行ったため，先行研究で行われたのと同様の手順でパラメータチューニングをしなければならない．特にパフォーマンスに影響を与えるのは GPU で探索する深さの決定である．4.3 に記した先行研究の紹介のために載せた図を図 6-7 に再掲する．先行研究での最短解探索では，GPU で探索する探索深度を 11 と決定している．

この 11 という数字は事前に実験を行い決定された値であるため，実装条件が変化している本追試実装においても同様の実験を行い，最適値を決定する必要がある．

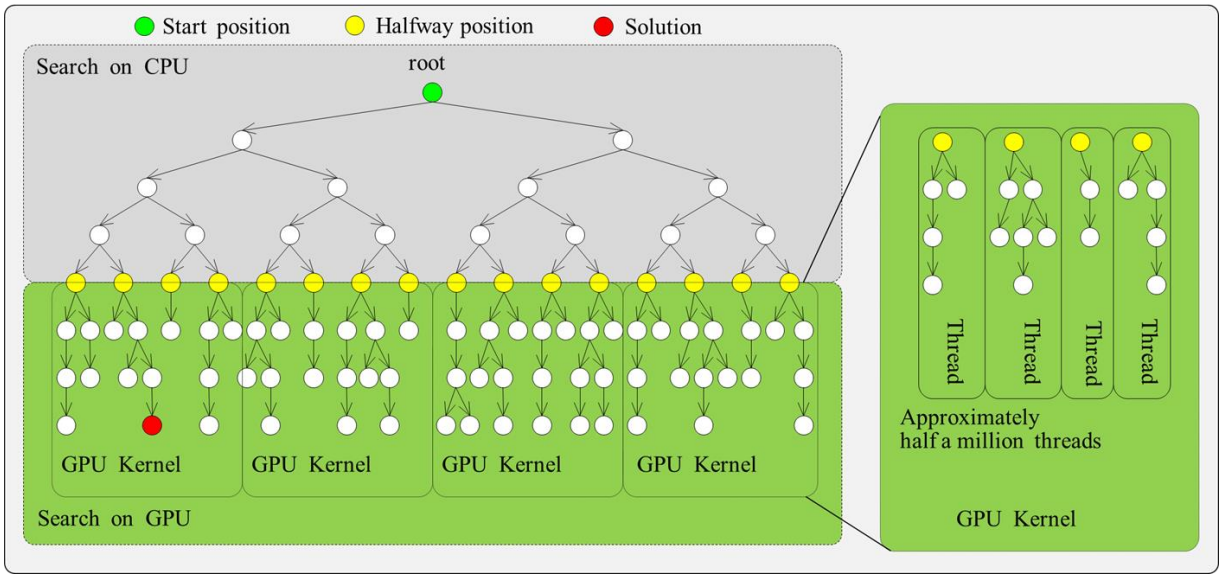

図 6-7　図 4-2 の再掲

実験では，解が 19 手の問題，生成手順（U　R　U2 R　F2 L　U2 R　F' B' R2 D　B2 U2 F2 L　R' F　R2）を GPU アクセラレーション有りで解き，GPU に与える探索深度ごとの実行時間を調べることで行った．結果を図 6-8 に示す．結果より，GPU で行う探索は先行研究と同様，深さ 11 が最善である結果が得られた．

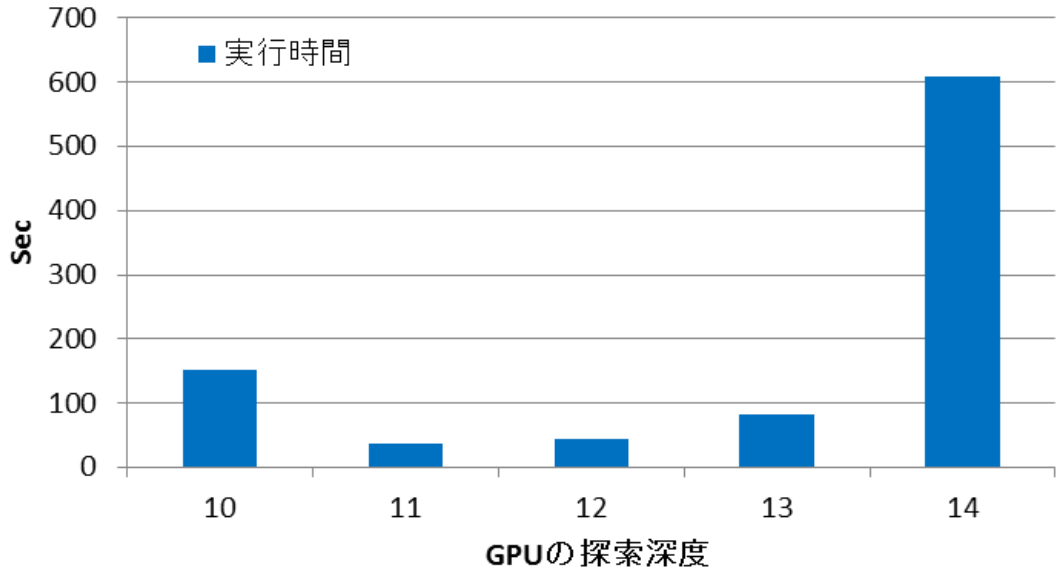

図 6-8　GPU での探索深度による求解時間の変化

6.3. 先行研究（SSGA [1]）の追試実験

ここでは，追試のために実装したプログラムの評価を行う．実験として，SUPER FLIP[§]（20手で解ける問題）と呼ばれる状態を解くのに要する時間を，CPU 実行した場合と GPU アクセラレーション有りで実行した場合を比較する．図 6-9 に追試の結果を示す．追試の結果，CPU（4cores 8threads）で実行した場合は 1431 秒で解が得られたのに対し，GPU アクセラレーション有り（CPU 1thread + GPU + GPU）の場合は 442 秒で解が得られた．結果は 4 コア CPU1 台に対し 3.23 倍の性能が得られた．先行研究の主張では，GPU1 台で 4 コア CPU の 5 倍の性能を実現したとしているが，本追試では先行研究のような大幅な性能向上とはならなかった．原因は未知であるが，使用メモリ量の増加や，距離関数の実装方法を複雑にしたことが要因であると推察される．しかし，先行研究の主張には及ばなかったが，IDA*アルゴリズムの GPU アクセラレーションには成功した．

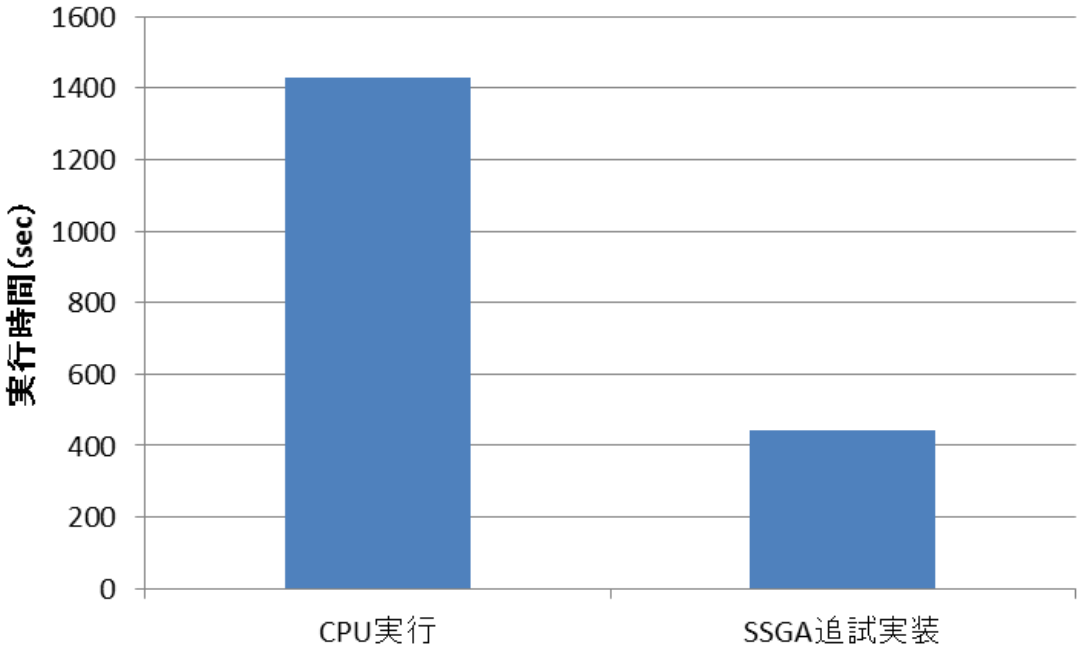

図 6-9　追試の結果

また，ルービックキューブソルバーとしての性能は，先行研究では 18 手の問題の求解時間の 100 個平均は 42.4 秒（GPU 実行）であったが，本追試では 9.3 秒（GPU 実行）となり，絶対性能は追試実装により大幅な向上がみられた．

§ U R U2 R F2 L U2 R F' B' R2 D B2 U2 F2 L R' F R2 D の手順で生成される

7. 実装手法 HWGA（Hybrid Worker GPU Acceleration）の提案

ここでは，SSGA [1]では触れられていなかった，GPU リソースだけでなく，CPU リソースも全て使い切る実装手法を提案する．SSGA では，CPU1 スレッドが GPU に供給するタスクを生成し，GPU 用のタスクを全て GPU で実行していたが，ここでは GPU 用に生成したタスクを CPU の残りスレッドにも割り当てる手法を提案する．ここで提案する実装手法は CPU1 スレッドがマスタースレッドとなり，残りの CPU スレッドと GPU がワーカーとなり動作するので，HWGA（Hybrid Worker GPU Acceleration）と命名する．

7.1. 実装の方針決定

ここでは，計算リソースを全て使い切った場合に，どの計算リソースにどの処理を割り当てるかを決定する．ここで実装した計算リソース割り当てを図にしたものを図 7-1 に示す．

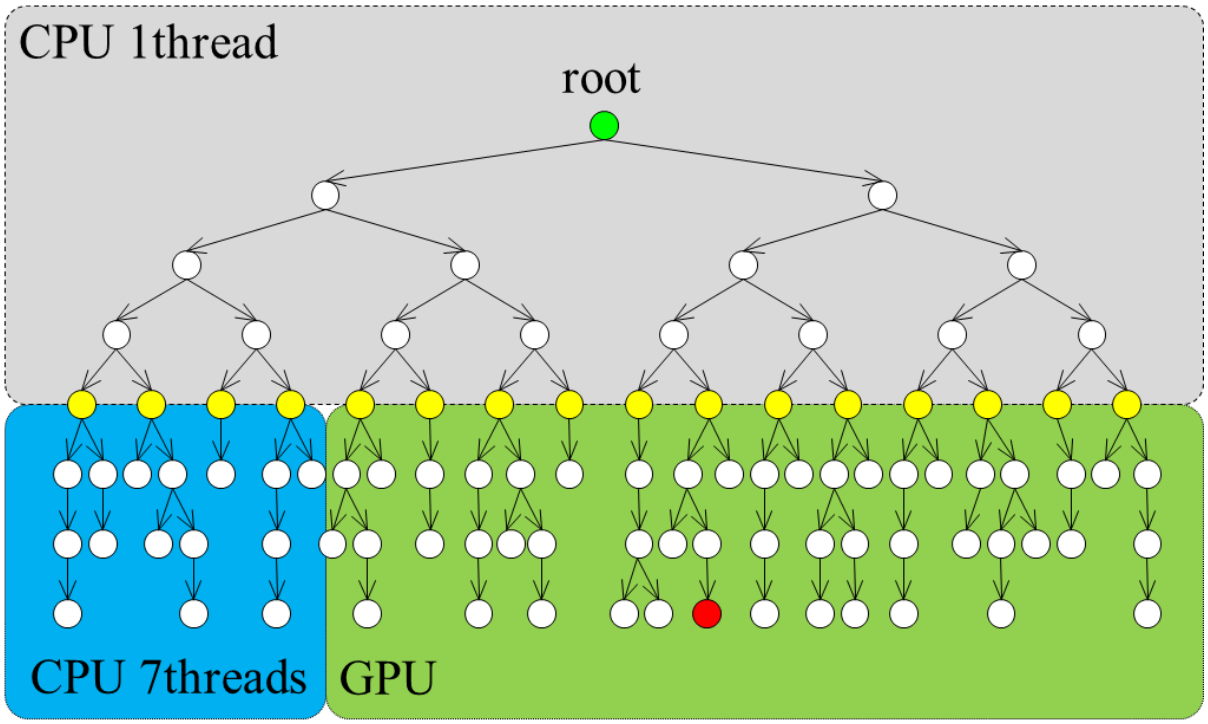

図 7-1　HWGA の計算リソース割り当ての方針

手法は基本的に SSGA と同様だが，GPU に割り当てていた処理の一部を CPU の残りスレッドに割り当てる．本研究での実装環境が 8 スレッド CPU のため，図中では CPU 7 スレッドと記したが, マスターとなる CPU1 スレッドを除いた残り全ての CPU スレッドをここに割り当てる．

各スレッド，GPU の制御は，図 7-1 中の灰色部を担当するスレッドをマスタースレッドとし，残りの CPU スレッドをワーカースレッド，GPU をワーカーとして，マスタースレッドから制御する方式とした．この方式を採用した根拠は，GPU は常駐プロセスを走らせること

ができないデバイスのため，CPU からの制御命令が無いと動作しない．そのため，GPU はワーカーとしてマススレッドから制御しなければならない．CPU スレッドについては常駐スレッドとして起動しておき，能動的に処理を受け取るような構造にすることも可能だが，GPU の処理と同一視できる設計の方がシンプルなため，マスター以外の CPU スレッドおよび GPU をワーカーとしてマススレッドから制御する方式を採った．

7.2. タスクキューの設計

図 7-1 の処理を実現するためには，先行研究と同様に，中間ノード（黄色ノード）を一時的に格納しておくキューが必要である．このキューをタスクキューと呼ぶことにする．本実装では，このタスクキューを 1 つ用意し，マススレッドが随時エンキューし，キューに十分なタスクが溜まり次第，マススレッドがデキューしデキューしたタスクをワーカーに転送する方式とした．

7.3. ワーカーの設計，制御

ここでは GPU ワーカー，CPU ワーカーの設計方法を記す．GPU ワーカーは GPU 制御の性質上，

・マススレッドからタスクの転送
・マススレッドから処理実行命令の発行
・マススレッドからの処理完了確認

の 3 つのセクションに分かれる．それぞれの処理は非同期処理であり，タスクの転送は転送命令発行直後に，転送が完了していなくてもマススレッドに制御が返る（データ転送の処理自体は OS により行われる）．マススレッドはタスク転送命令発行直後に処理実行命令を発行する．この処理も非同期であり，発行直後にマススレッドに制御が返る．このとき GPU デバイス上では，GPU 上の命令キューにデータ転送命令と処理実行命令がエンキューされた状態となり，データ転送は OS により行われ，転送が完了すると処理が開始される．GPU の命令キューに投入されたタスクが完了しても，GPU から OS にシグナルなどを送る機構が無いため，タスクの完了はマススレッドから GPU に問い合わせなければならない．CUDA では cudaStreamQuery という関数が提供されており，マススレッドから同関数を実行することで，GPU が busy 状態か idle 状態かを知ることができる．これを利用し，マススレッドは定期的に同関数を実行し，GPU 上の処理が完了したかを確認する．

次に，CPU ワーカーの設計方法を記す．CPU ワーカーも GPU ワーカーと同様に扱えるとマススレッドの処理が簡潔になるため，GPU の制御構造に似せた実装をした．CPU ワーカーは P thread にて常駐スレッドとして起動し，タスクを受け取るメモリ領域と，busy 状態か idle 状態かを示すフラグを設ける．ワーカースレッドはこのタスクを受け取るメモリ領域を監視し，タスクが転送されたらフラグを busy 状態にしタスクを実行し，完了したらフラグを idle 状態に戻す処理をする．このような設計にすると，マススレッドからは

・マススレッドからのタスクの転送
・マススレッドからの処理完了確認

の 2 つのセクションの制御になる．処理完了確認はワーカーのフラグを定期的に読み取り，busy 状態ならば実行中，idle 状態ならば処理完了とみなす．

以上の設計により，CPU ワーカーと GPU ワーカーをほぼ同一に使用できるようになり，マスタースレッドの処理が簡潔になる．

7.4. 解の転送制御

7.3 にて，ワーカーの制御構造を示したが，解の転送手段が未解決である．図 7-1 からわかるように，解はワーカーの何れかで発見される．ここでは，ワーカーでの解の処理について記述する．まず GPU 側の処理を記す．GPU は GPU プログラムからメインメモリに書き込むことができないため，GPU 上で解が発見された場合は GPU の特定のメモリ領域に解を書き込み，マスタースレッドが GPU 上の該当メモリ領域を読み取ることにより解の判定を行う．この解の読み取り処理は GPU が busy 状態から idle 状態に切り替わったタイミングで行い，ここで正しい解が読み取れれば解有りと判定し探索を終了し，正しい解が読み取れなかった場合は解無しと判定し，探索を続行する．

CPU ワーカーも GPU ワーカーと同一視できるような実装を行う．CPU ワーカーで解が発見された場合は，CPU ワーカー上のメモリ領域に解を書き込み，読み取りはマスタースレッドから行うことで GPU と同等に扱うことができる．読み取るタイミングも GPU と同様，CPU ワーカーが busy 状態から idle 状態に切り替わったタイミングで該当領域を読み，正しい解が格納されていれば解有りと判定する．

7.5. アルゴリズム

本セクションのまとめとして，マスタースレッド，CPU ワーカー，GPU ワーカーのアルゴリズムをフローチャートにて記す．

7.5.1. マスタースレッドの処理

マスタースレッドの処理をフローチャートとして表すと図 7-2 のようになる．フローチャートは複雑だが，基本的な流れは，タスクキューに中間ノードを溜める作業をしつつ，idle 状態のワーカーがあるかを常時確認し，idle 状態のワーカーがあればタスクを転送するという流れになっている．

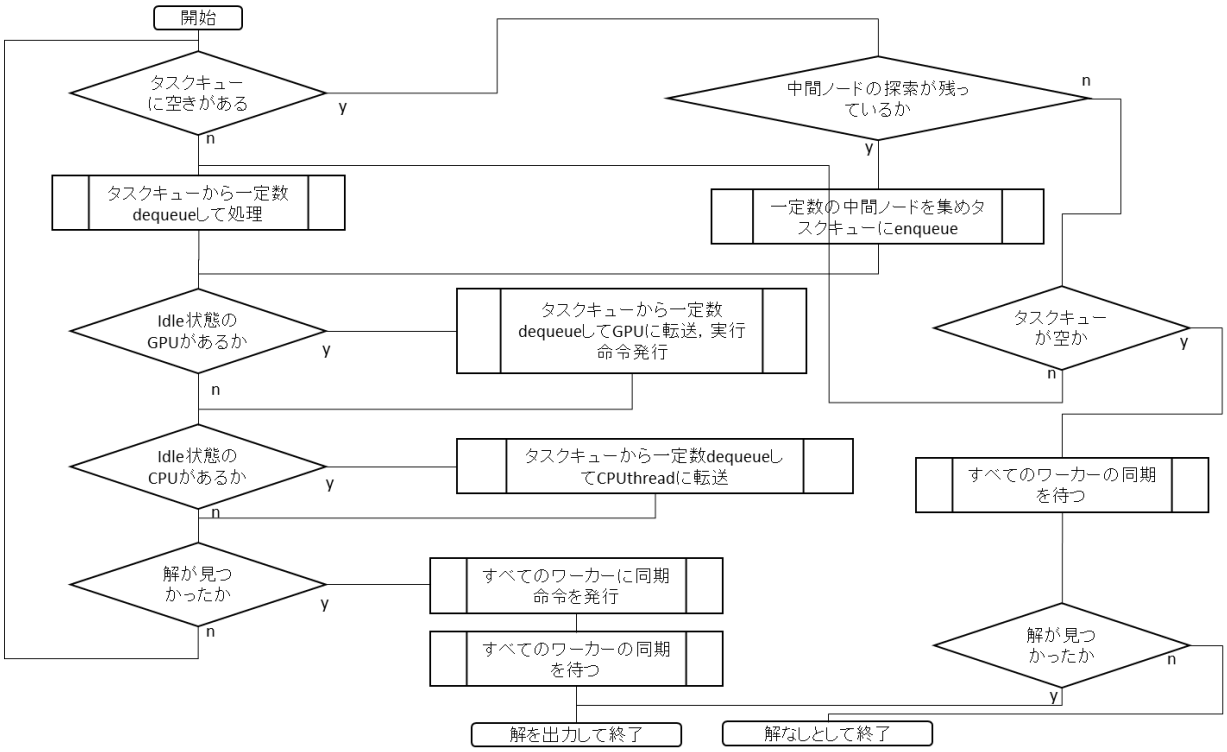

図 7-2　マスタースレッドの処理

7.5.2. CPU ワーカースレッドの処理

CPU ワーカースレッドは図 7-3 のようなシンプルなものとなる．基本的にタスクの受け取り待ち状態でループし，タスクが転送され次第処理，解けた場合に解を特定の領域に書き込むという流れになる．

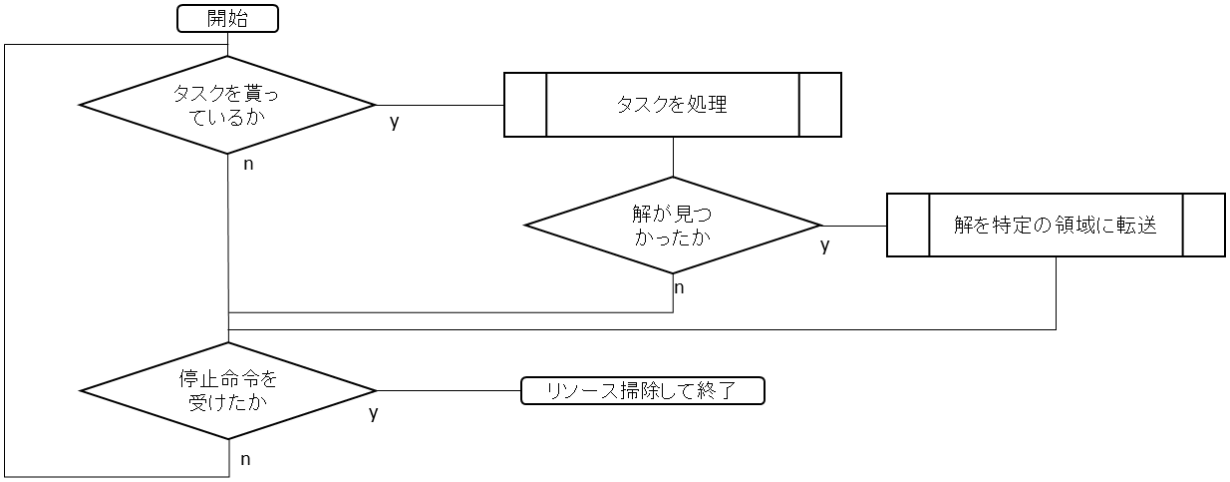

図 7-3　CPU ワーカースレッドの処理

7.5.3. GPU ワーカーの処理

GPU ワーカーの処理は図 7-4 のようなシンプルなものになる．図ではタスク処理後終了となっているが，GPU での処理はプログラムが終了しても特定の GPU メモリ領域の状態は保

持されるため，解が見つかった場合はこの領域に書き込み，プログラム終了と同時に破棄されないように制御する．

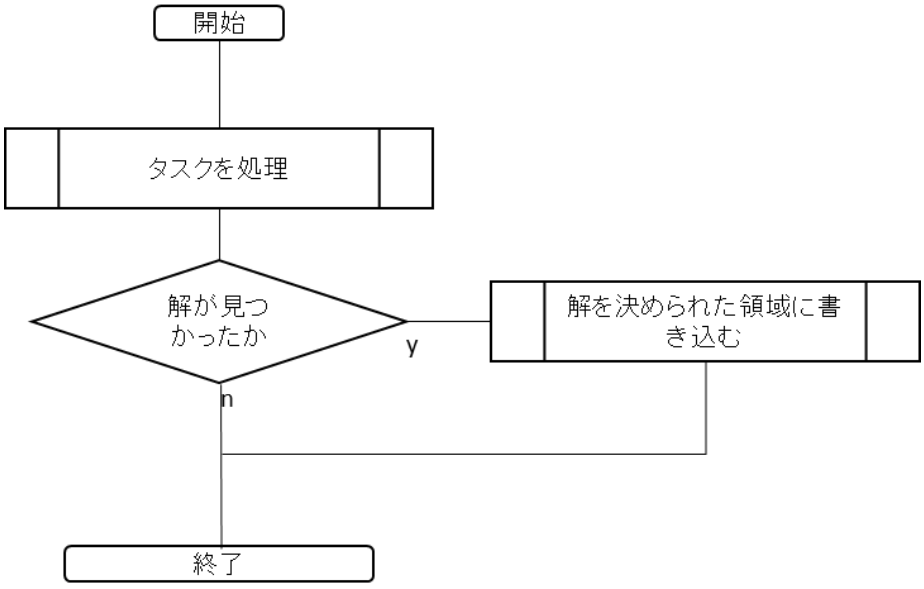

図 7-4　GPU ワーカーの処理

7.6. 性能評価と新たな課題

7.6.1. 性能評価

ここで，提案した HWGA の性能評価と，新たな課題の調査を行う．実験は SUPER FLIP を解く時間で評価する．結果は図 7-5 のようになった．CPU 4cores 8threads で実行した場合が 1431 秒，SSGA [1]が 442 秒，提案手法である HWGA が 347 秒となった．結果として，SSGA [1]に対し 22%の性能向上を実現した．

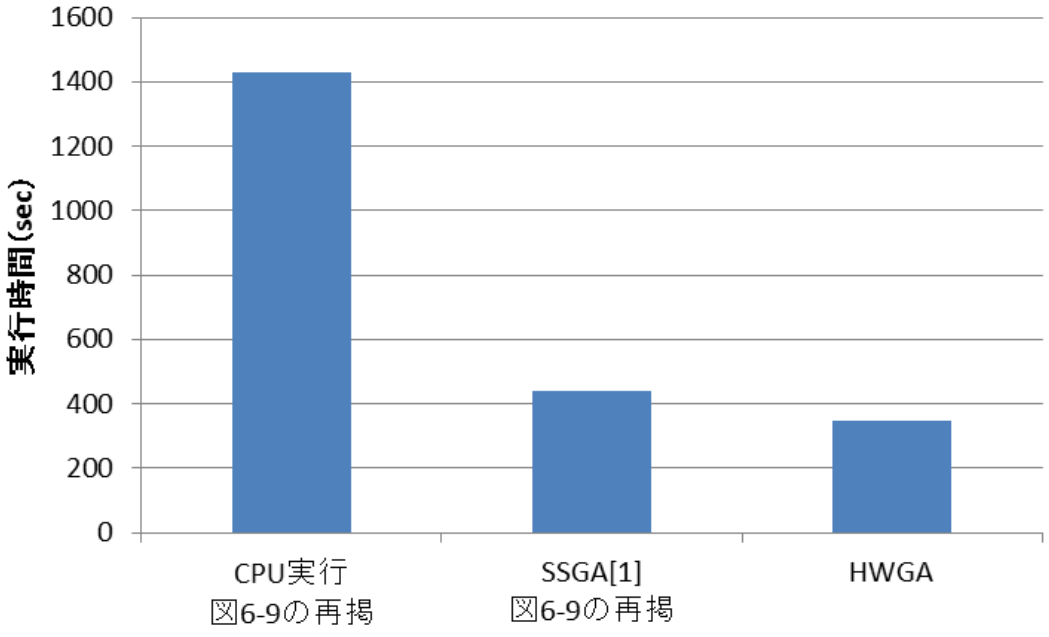

図 7-5　SUPER FLIP 求解による HWGA の性能評価

7.6.2. HWGA の課題

7.6.1 の評価を見ると，提案手法である HWGA は有効な方法と評価できる．しかし，探索時の処理時間を詳細に分析すると，新たな課題を見いだすことができる．HWGA の略図を図 7-6 に示す．

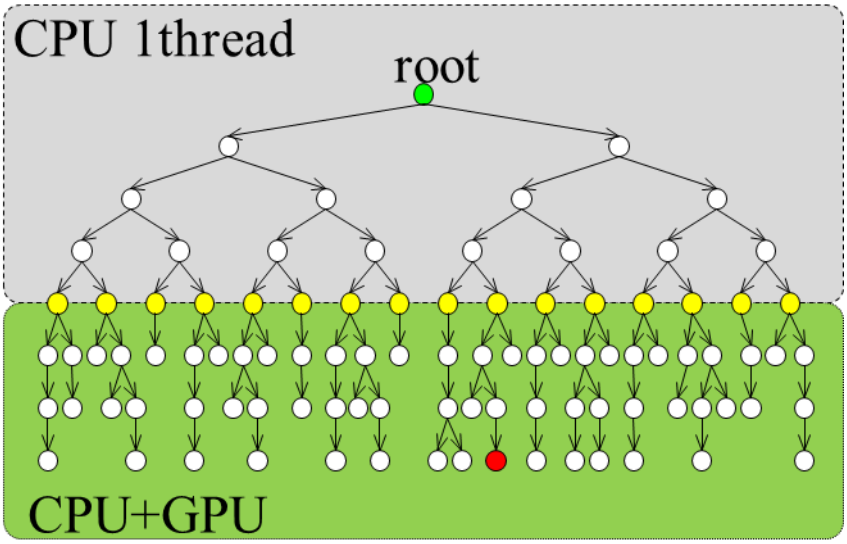

図 7-6　HWGA の略図

図中の灰色部の処理時間と緑色部の処理時間を比較すると，7.6.1 の評価実験における，深さ 19 の探索時における灰色部の処理時間は 302 秒であり，緑色部の処理時間は 318 秒であった．この結果から，マスタースレッドの処理速度とワーカーの処理速度がほぼ同じであることがわかる．これは GPU を増設した場合や，より高性能な GPU に換装した場合など，ワーカーの処理性能が更に向上した場合にマスタースレッドの処理時間がボトルネックとなる可能性を示している．これを解消する手段は 2 通り考えられ，1 つは灰色部の処理性能を向上させる手段であり，もう 1 つは緑色部の探索深度を増やす方法である．後者の方法では，6.2 で示したパラメータチューニングにより得られた GPU での最適な探索深度 11 を変更することになるため，GPU における実効性能を低下させることになる．従って，本研究では前者の灰色部の処理性能を向上させる手段を考えることにした．

8. CPU 動作周波数が不十分な場合における改良型 HWGA

7 にて，計算機上の計算リソースを全て効率良く使用するアルゴリズム HWGA を提案したが，7.6.2 にて新たな課題が浮上した．ワーカーの性能は GPU を増設するなどの手段で容易に向上させることが可能であるため，例えば GPU を 4 枚搭載した計算機では 7.6.2 で示したボトルネックが表面化することが予測される．この章では，7.6.2 で示したボトルネックが表面化した場合に，計算リソースを効率良く使用する実装手法を提案する．

8.1. 事前準備（HWGA の持つ課題の表面化）

新たな改良型 HWGA の提案の前に，7.6.2 で示したボトルネックが表面化した状態を再現しなければならない．ここでは，ワーカーの性能を向上させるのではなく，CPU の動作周波数を落とし，ボトルネックを再現することにした．

CPU の動作周波数を 800MHz に設定し，7.6.2 と同様の評価を行ったところ，図 7-6 の灰色部の処理時間は 532 秒，緑色部の処理時間は 534 秒となった．これとは別に，灰色部の処理が全て完了した後に緑色部の処理を実行した場合を計測したところ，緑色部の処理は 356 秒で完了するという結果が得られた．即ち，ワーカーの処理速度がマスタースレッドの処理速度を超えた状態を再現できた．

8.1.1. 計算機使用率に基づくボトルネックの可視化

ここでは，HWGA の CPU ボトルネックが表面化した場合の CPU 使用率・GPU 使用率を計測してボトルネックの可視化を試みた．図 8-1 に SUPER FLIP 求解時の CPU 使用率･GPU 使用率の時間的な遷移を示す．図のように，CPU 使用率・GPU 使用率共に負荷が 70%程度に留まっており，計算機を効率よく使用できていないことが確認できる．

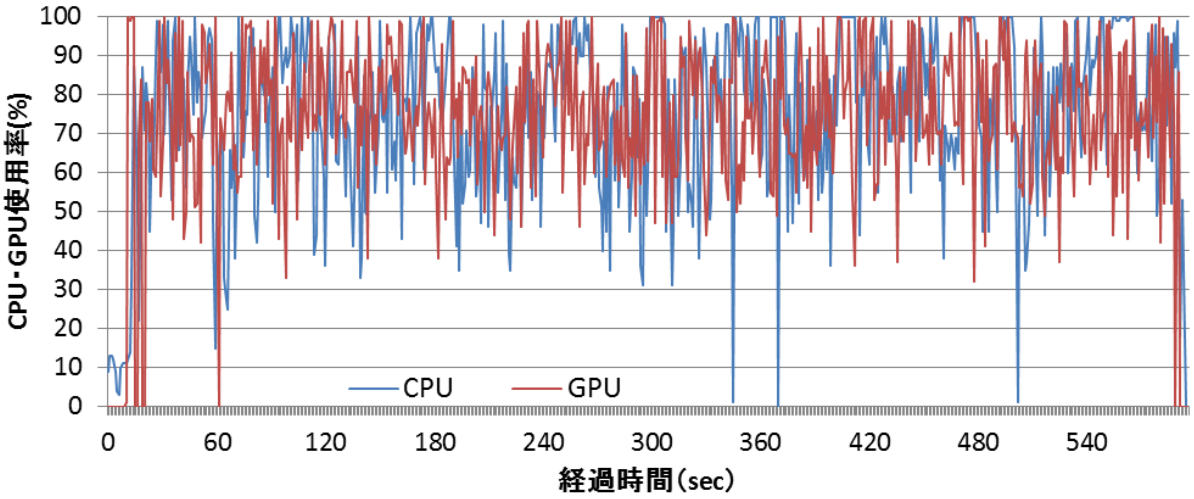

図 8-1　CPU 動作周波数が不十分な状態での HWGA の GPU・GPU 使用率

8.2. 改良型 HWGA（1）FM-HWGA（Fast Master Thread HWGA）の提案

ここでは，探索で形成されるグラフの性質を利用した高速化手段を示す．ここに示す手段はグラフの形に着目した手段であり，特にルービックキューブの探索に特化した手段であることから，汎用性は不十分である可能性があるが，ルービックキューブ探索においては良い効果が得られたため，ここに実装手法と結果を残す．

ここで示す手法は，HWGA にある工夫をすることで，マスタースレッドの性能を倍以上にすることができることから，FM-HWGA（Fast Master Thread HWGA）と命名する．

8.2.1. ルービックキューブ最短解探索における探索グラフの特徴

図 8-2 に，SUPER　FLIP の最短解探索における深さ 19 の探索時に形成された探索グラフの，探索深度ごとのノード数を示す．図から，深さ 8 までの探索では距離関数による枝狩りが効かず，指数関数的にノード数が増加していることがわかり，深さ 9 以降の探索では距離関数が有効に働きノード数が減少していることが確認できる．このとき，深さ 8 のノードが探索の中間ノードである（図 7-6 中の黄色ノード）ことに注目する．深さ 8 までの探索では，距離関数の効果が弱いため，指数関数的にノード数が増える．ここで，深さ 8 までの探索で距離関数を使わない場合を考える．距離関数を使わずに探索した場合に出現するノード数は表 6-1 から 1484451135 であることがわかる．このノード数と，距離関数を利用した場合のノード数である 1367223216 を比較すると，差が 8.6%であることがわかる．即ち，マスタースレッドで行う探索で距離関数を使用せず，ワーカーのみ距離関数を使用する場合を考えても，形成されるグラフは距離関数を常時使用した場合とほぼ同一であり，処理量の増加量が 8.6%程度であるといえる．

探索深さ	ノード数	ノード数の比
0	1	-
1	18	18.0
2	243	13.5
3	3240	13.3
4	43254	13.4
5	577368	13.3
6	7706988	13.3
7	102876384	13.3
8	1367223216	13.3
9	5331044616	3.9
10	1207692864	0.2
11	166686816	0.1
12	72314976	0.4
13	45023376	0.6
14	42533376	0.9
15	34793376	0.8
16	21214896	0.6
17	5676480	0.3
18	96000	0.0
19	0	0.0

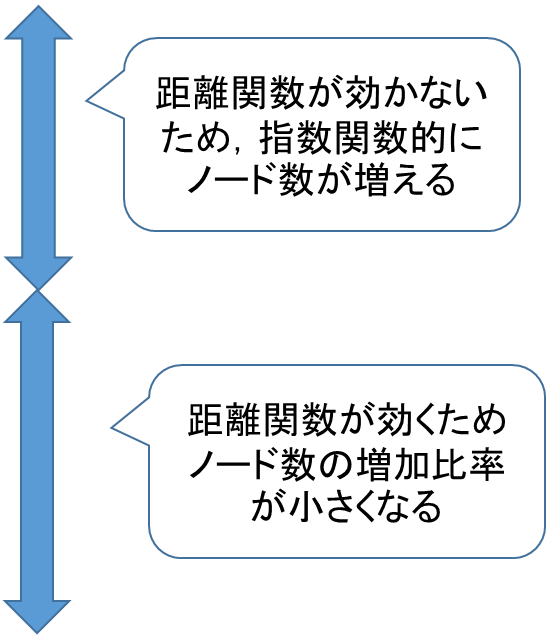

図 8-2　SUPER FLIP を探索したときの深さ 19 の探索のノード数

8.2.2. マスタースレッドの高速化

前章にて，マスタースレッドで行う探索で距離関数を使用せず，ワーカーのみ距離関数を使用する場合を考えても，形成されるグラフは距離関数を常時使用した場合とほぼ同一であることを示した．

ここで，実際にマスタースレッドで行う探索で距離関数を使用しない場合にパフォーマンス変化があるかを検討する．距離関数は CPU のキャッシュメモリに載らない大きなテーブルにて実装されていることは先述した．従って，図 6-5 に示した探索アルゴリズムは距離関数へのアクセスが大きな負荷を占めていると予測できる．即ち，マスタースレッド（図 7-6 の灰色部）の探索中で距離関数を使わない探索を行った場合，ワーカーが処理する処理量は微増（前章より 8.6%程度の増加）するが，マスタースレッドの性能が大幅に向上し，ボトルネックが解消し総合的なパフォーマンスの向上が期待できると考えられる．

8.2.3. マスタースレッドの性能評価

ここでは，マスタースレッドにて距離関数を使用した場合と使用しなかった場合とでどの程度マスタースレッドの実行速度に変化があるのかを実験する．実験では，SUPER FLIP を探索した場合の深さ 19 の探索時におけるマスタースレッド（図 7-6 の灰色部）の実行時間を計測することで行った．実験結果を図 8-3 に示す．ボトルネックがある状態の提案手法 HWGA ではマスタースレッドの処理に 532 秒要してたのに対し，ここで提案した FM-HWGA では 215 秒となり，マスタースレッドの実行速度を約 2.5 倍にすることができた．なお，ワーカーの処理には約 356 秒要することが事前実験によりわかっているため，目的であるボトルネックの解消が実現できたと予測できる．

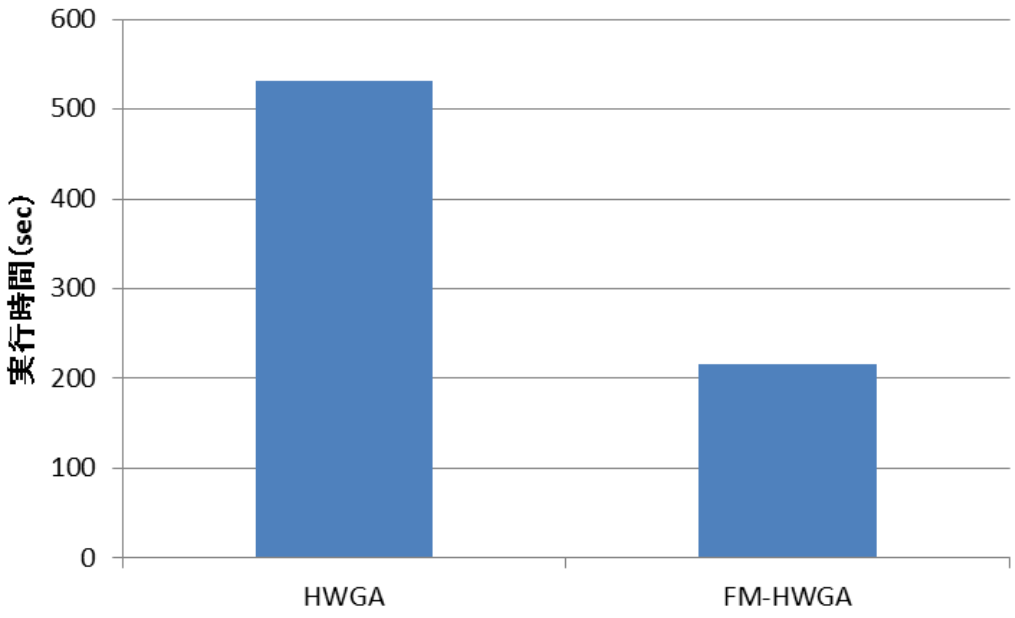

図 8-3　HWGA と FM-HWGA のマスタースレッドの処理時間比較

8.2.4. FM-HWGA の性能評価

ここでは，SUPER FLIP の求解に要する時間を測定して，FM-HWGA の性能を検証する．図 8-4 に実験結果を示す．ボトルネックがある状態の HWGA は 581 秒に対し，FM-HWGA では 399 秒となり，31%の性能向上を実現した．

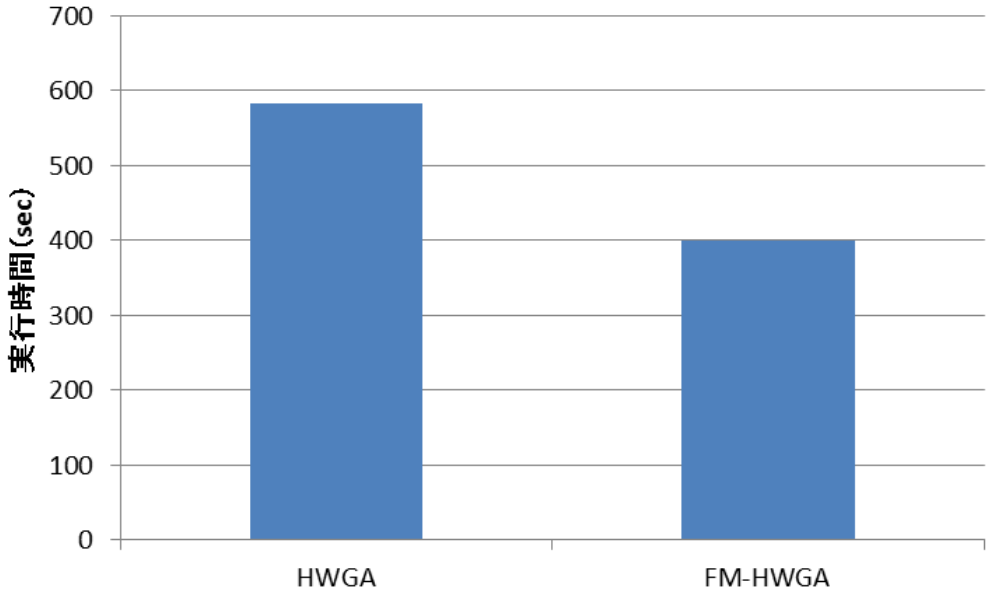

図 8-4　SUPER FLIP 求解による FM-HWGA の性能評価

8.2.5. FM-HWGA の計算機使用率に基づく評価

ここでは，8.1.1 にて，計算機使用率に基づいて可視化した CPU ボトルネックがどの程度解消されたのかを，8.1.1 と同様に計算機使用率を計測して確認する．FM-HWGA による SUPER FLIP 求解時の時間経過ごとの CPU・GPU 使用率を図 8-5 に示す．

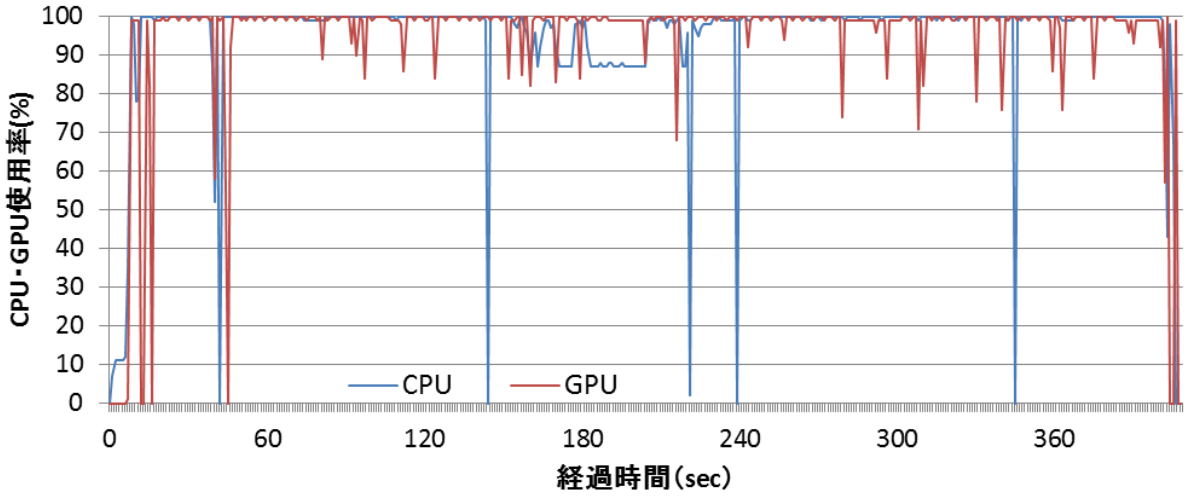

図 8-5　CPU 動作周波数が不十分な状態での FM-HWGA の GPU・GPU 使用率

図から，CPU・GPU 使用率共に，負荷の瞬断は確認できるが終始 100%に近い負荷をかけることに成功しており，計算機使用率に基づく評価ではボトルネックの解消に成功したと評価できる．

8.3. 改良型 HWGA（2）MM-HWGA（Multithreaded Master HWGA）の提案

ここでは，マスタースレッドをマルチスレッド化し，図 7-6 の灰色部の処理の高性能化を図り，HWGA のボトルネック解消を試みる．マスターの処理がマルチスレッド化するため，ここで提案する手法を MM-HWGA（Multithreaded Master HWGA）と呼称する．

8.3.1. CPU で行う処理の設計

ここでは，CPU＋GPU のワーカーが処理するタスクの収集を CPU スレッド全てで行うことで，ワーカーへのタスク供給のボトルネックを解消する方法を提案する．具体的な設計は図 8-6 に示す．構想はマスタースレッドが行う探索深度を浅くし，図中のオレンジで示すノードを収集する．このノードを CPU の他スレッドが並列に処理し，黄色ノードを収集することで，CPU 性能を使い切り CPU＋GPU のワーカーへ渡すタスク収集の高速化を図る．

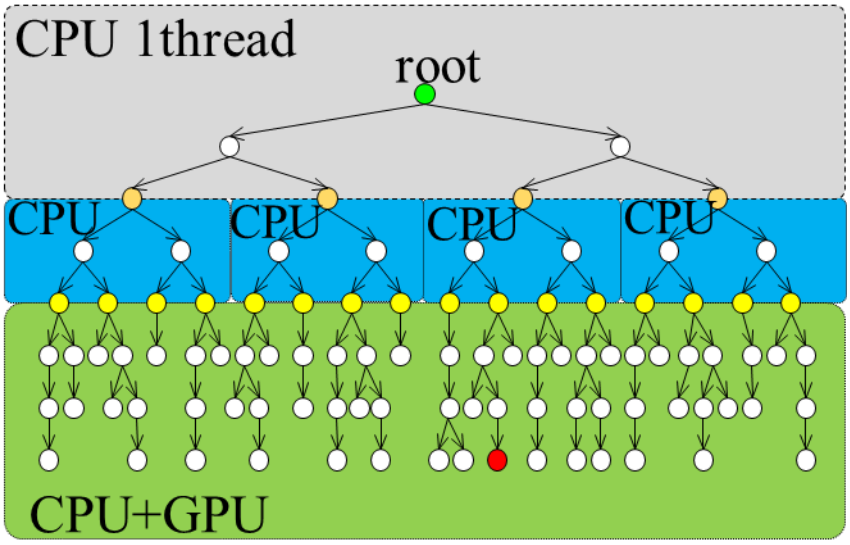

図 8-6　MM-HWGA の処理の略図

8.3.2. CPU スレッドのアルゴリズム修正

図 8-6 に示す動作を実現するには，7.5 で示したアルゴリズムの修正を行わなければならない．8.3.2.1 と 8.3.2.2 に CPU マスタースレッドの修正アルゴリズムと，CPU ワーカーの修正アルゴリズムを示す．なお，GPU ワーカーのアルゴリズムの修正は不要である．

8.3.2.1. CPU マスタースレッドのアルゴリズム修正

図 8-7 にマスタースレッドの修正アルゴリズムを示す．アルゴリズム中で扱うキューが 2 本に増えたので，図 8-6 中のオレンジのノードを管理するキューを CPU キュー，黄色のノードを管理するキューを GPU キューと命名する．マスタースレッドは図 8-6 中のオレンジのノードを収集しながら CPU キューに順次投入し，CPU ワーカースレッドにタスク供給を行う．GPU へのタスク供給もマスタースレッドにて行うが，GPU キューへのタスク供給は CPU ワーカースレッドが行う．

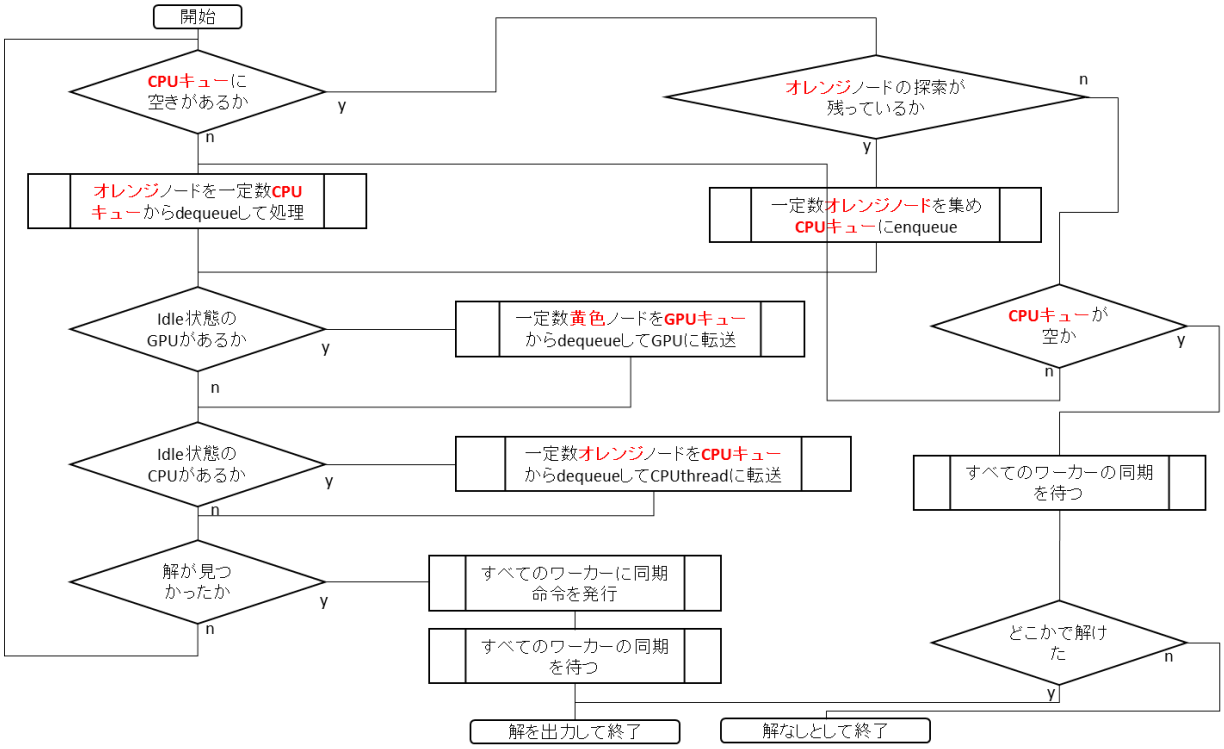

図 8-7　マスタースレッドの修正アルゴリズム

8.3.2.2. CPU ワーカースレッドのアルゴリズム修正

図 8-8 に CPU ワーカースレッドの修正アルゴリズムを示す．修正されたワーカースレッドは供給されたタスクを全て処理して探索を行うか，図 8-6 中の黄色ノードまでの探索のどちらかを行う．GPU キューにタスクが十分にある状態では GPU 用タスクを生成するのではなく，与えられたタスクの残りの探索を全て行う．

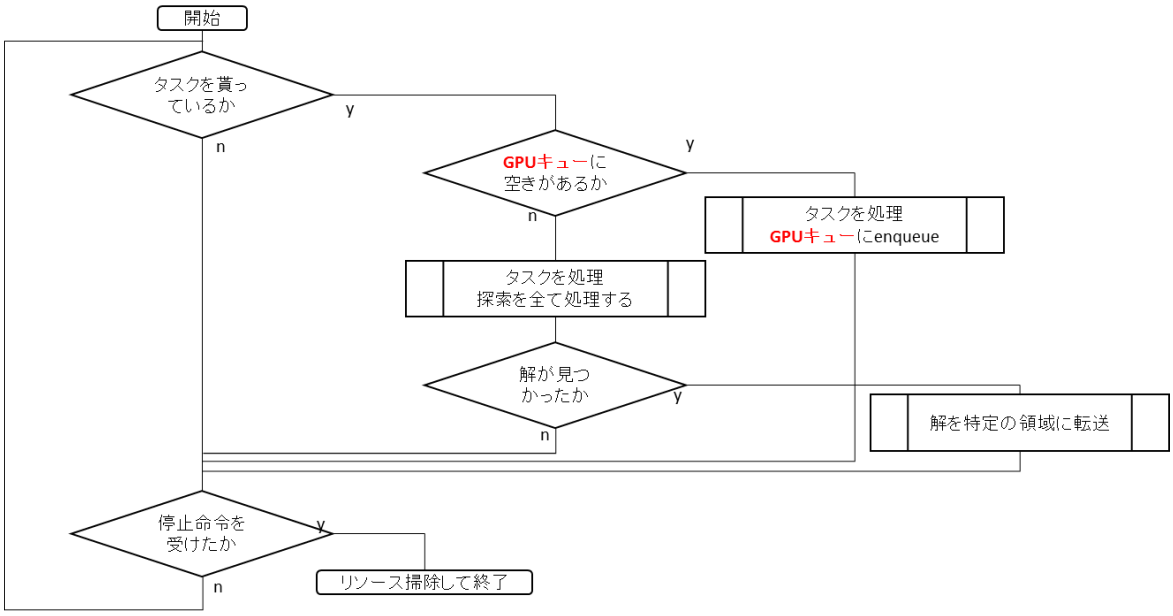

図 8-8　CPU ワーカースレッドの修正アルゴリズム

従って，図 8-6 に示した設計方針とは僅かに動作が異なり，実際には図 8-9 のような動作となる．

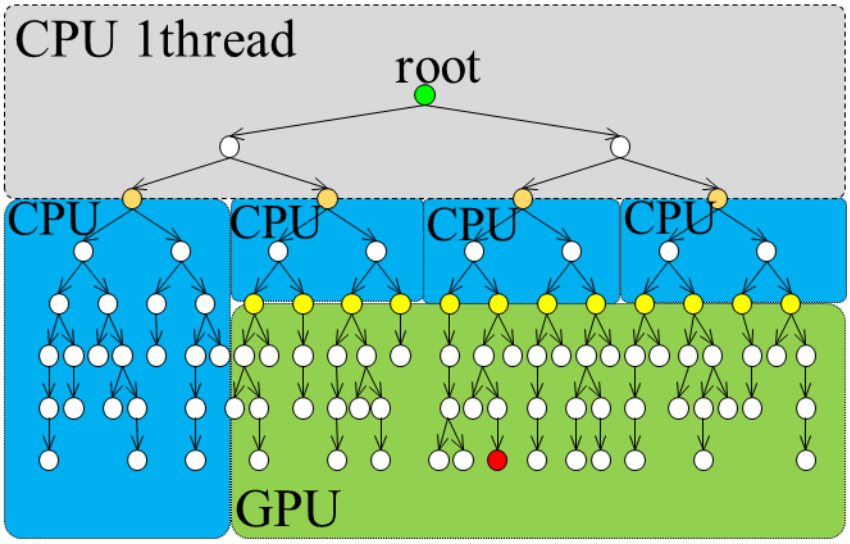

図 8-9　MM-HWGA の処理の実際の実装

8.3.3. MM-HWGA の性能評価

ここでは，SUPER FLIP の求解に要する時間を測定して，MM-HWGA の性能を検証する．図 8-10 に実験結果を示す．ボトルネックがある状態の HWGA は 581 秒に対し，MM-HWGA では 409 秒となり，30%の性能向上を実現した．

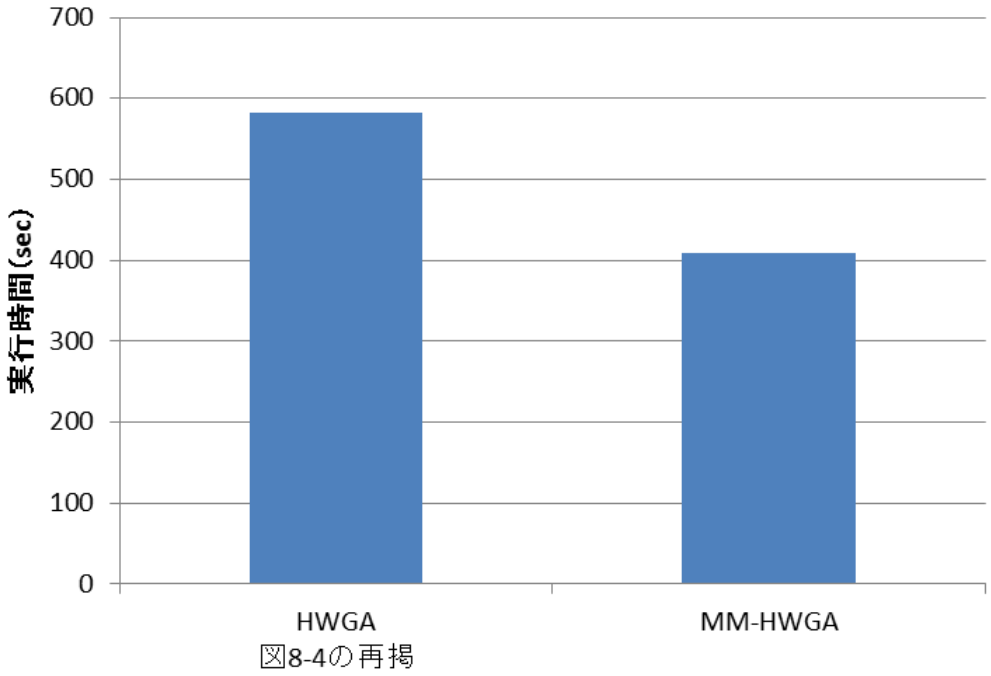

図 8-10　SUPER FLIP 求解による MM-HWGA の性能評価

8.3.4. MM-HWGA の計算機使用率に基づく評価

8.2.5 と同様に，MM-HWGA も計算機使用率に基づく評価を行う．SUPER FLIP 求解時の時間毎の CPU・GPU 使用率の変化を図 8-11 に示す．図から，CPU・GPU 使用率共に，負荷の瞬断は確認できるが終始 100%に近い負荷をかけることに成功しており，MM-HWGA においても FM-HWGA と同様に，計算機使用率に基づく評価ではボトルネックの解消に成功したと評価できる．

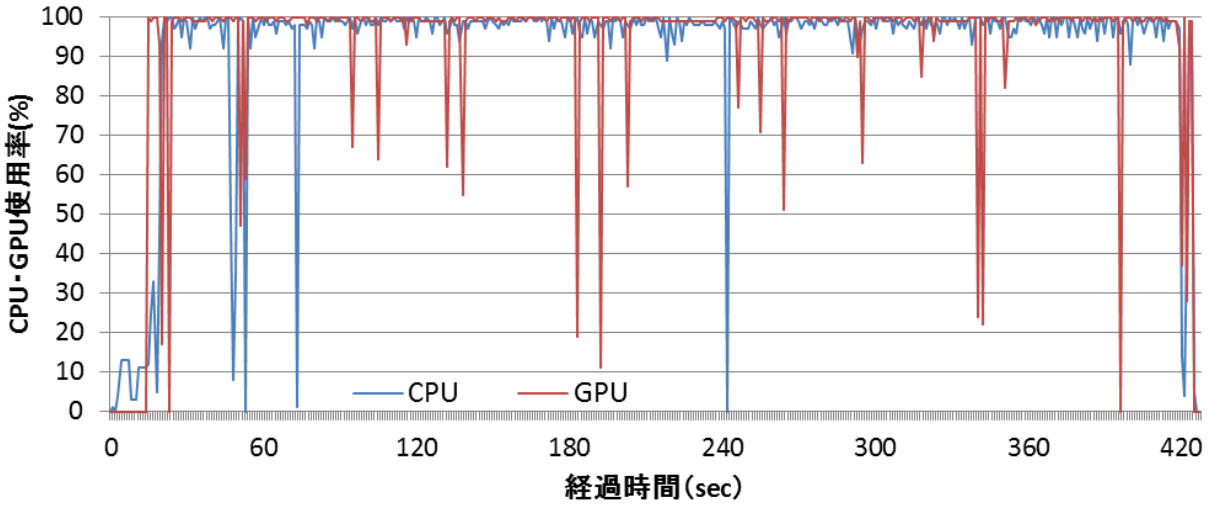

図 8-11　CPU 動作周波数が不十分な状態での MM-HWGA の GPU・GPU 使用率

9. 総合評価

ここまでで，提案手法を 3 つ提示してきた．GPU のみならず，CPU リソースも使い切るような実装を目的とした HWGA，HWGA で起こりうる CPU ボトルネックを解消する FM-HWGA，MM-HWGA，の 3 つである．ここでは，この 3 つの提案に加え，従来手法である SSGA [1]（単純な GPU アクセラレーション）の 4 つのプログラムを，CPU 動作周波数が高い状態（ボトルネックが発生しない条件）と，CPU 動作周波数が低い状態（ボトルネックが発生する条件）の 2 通りで，どのような性能となるのかを比較，検討する．

実験は SUPER FLIP を求解する場合に加え，ランダムなルービックキューブを 3000 個求解する場合についても実施した．

9.1. SUPER FLIP 求解による評価実験

9.1.1. CPU 動作周波数 3.8GHz での実験

図 9-1 に HWGA でのボトルネックが発生しない条件下（CPU 性能が十分な場合）での実験結果を示す．実験の結果から，提案手法 3 つは何れも SSGA [1]より高い性能を示し，提案手法 3 つの中での性能差は最大 6%で，提案手法は安定して高い性能を示すことが確認できた．特に，FM-HWGA と MM-HWGA の 2 つは CPU 動作周波数が低い状態を想定して設計した手法にも関わらず，FM-HWGA は最も良好な結果を示し，MM-HWGA では僅かながら HWGA より悪い結果となっているが，HWGA とほぼ同等の性能を示しており，FM-HWGA と MM-HWGA の 2 つの実装手法は CPU 動作周波数に関わらず良い結果を示す性質の良い実装手法と言える結果が得られた．

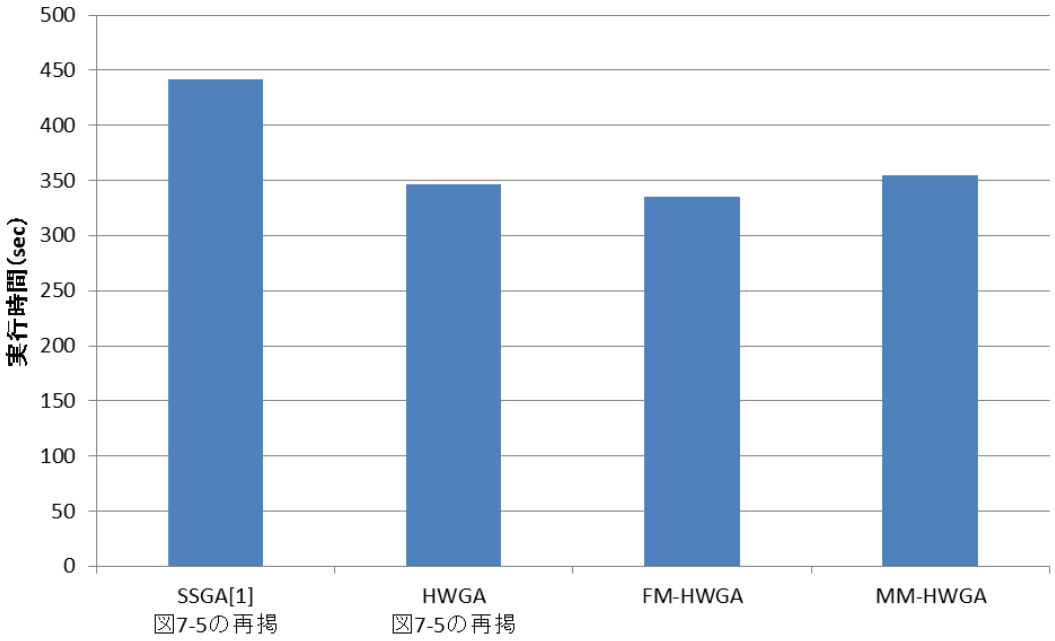

図 9-1　CPU 動作周波数が高い条件下での先行研究と提案手法の比較

9.1.2. CPU 動作周波数 800MHz での実験

図 9-2 に, HWGA ではボトルネックが発生する条件下での実験結果を示す. 図から, HWGA では CPU ボトルネックが顕著に表れ, SSGA [1]による単純な GPU アクセラレーションと比較しても非常に性能が悪くなってしまっていることがわかる. 対して, FM-HWGA と MM-HWGA の 2 つは共に高い性能を示していることが結果からわかる.

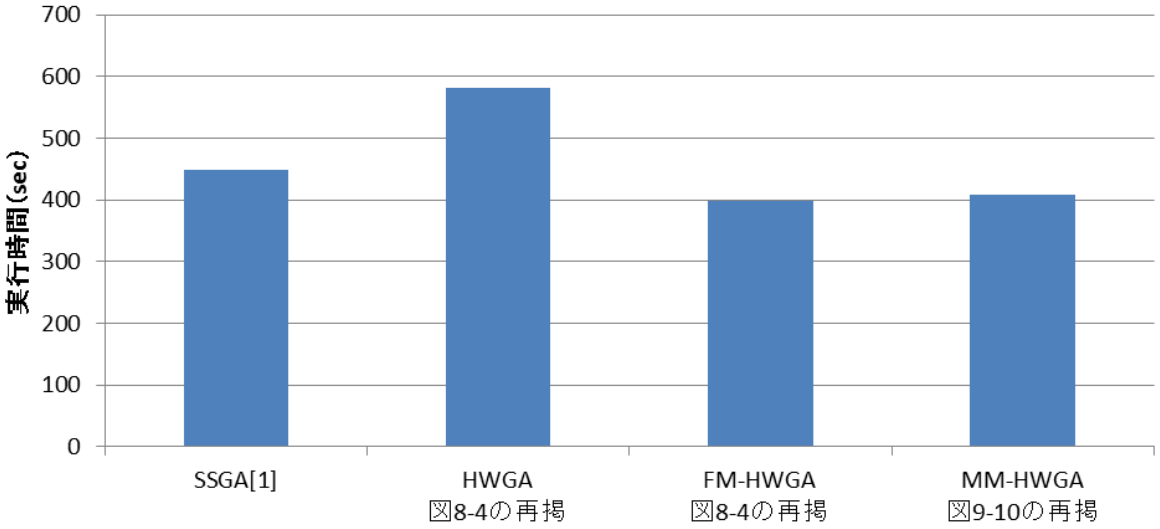

図 9-2　CPU 動作周波数が低い条件下での先行研究と提案手法の比較

9.2. ランダムキューブ 3000 個求解による評価実験

これまでの実験では, ソルバーの性能評価に SUPER FLIP の求解を用いてきたが, ここではランダムなルービックキューブを解いた場合における, それぞれの手法 (SSGA [1], HWGA, FM-HWGA, MM-HWGA) の評価を行う.

9.2.1. ランダムキューブ 3000 個の準備

評価実験ではランダムなルービックキューブを解くが, 乱数の SEED 値によって生成されるキューブが異なるため, 評価実験毎に結果が変化する可能性がある. そこで, 事前にランダムなキューブを 3000 個生成し, それぞれの評価実験では, 事前に生成された共通の問題 3000 個を解くことで, 評価が厳密に行えるように準備した.

表 9-1　評価に使用するランダムキューブ 3000 個の解の長さ分布

解の長さ	個数
14	1
15	5
16	81
17	834
18	1964
19	115

表 9-1 に実験で使用するランダムキューブ 3000 個の解の長さの分布を示す．表のように，ルービックキューブはランダムに生成した場合はほとんどが 18 手，17 手で解ける問題となり，SUPER FLIP の求解（20 手要する）と比較すると負荷が非常に軽いことが言える．この軽い負荷の条件下で各手法がどのような性能を示すのかを実験する．

9.2.2. CPU 動作周波数 3.8GHz での実験

図 9-3 にランダムキューブ 3000 個求解した場合の平均求解時間を示す．結果から，提案手法 3 つは何れも SSGA [1]より高速な結果が得られたが，HWGA と MM-HWGA については SSGA [1]とほとんど変わらない性能を示した．FM-HWGA のみ非常に高い性能を示し，SSGA [1]に対し 15%速い結果が得られた．

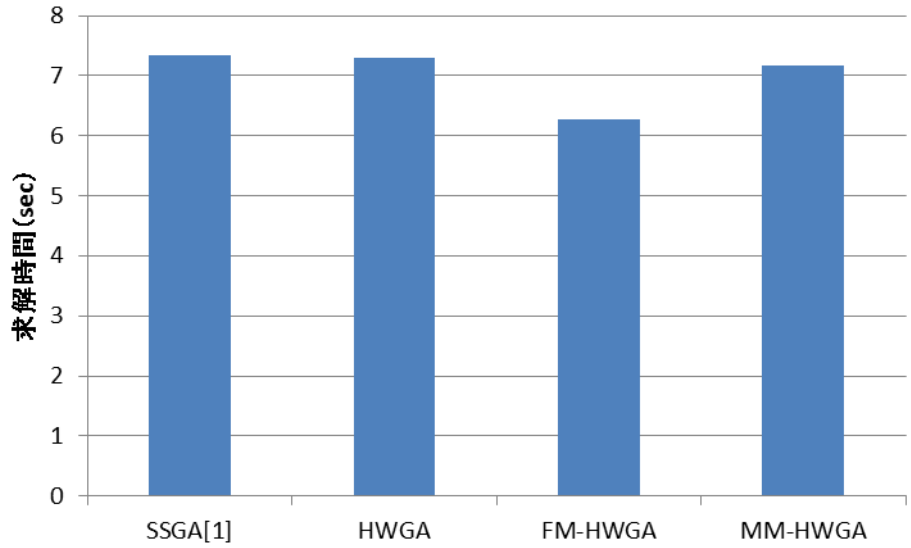

図 9-3　CPU 動作周波数 3.8GHz でのランダムキューブ 3000 個の平均求解時間

9.2.3. CPU 動作周波数 800MHz での実験

図 9-4 にランダムキューブ 3000 個求解した場合の平均求解時間を示す．CPU 動作周波数が低い条件下では，提案手法である HWGA と FM-HWGA については SSGA [1]に劣る結果となった．対し，MM-HWGA のみ SSGA [1]と比較して 4%高速な結果が得られた．HWGA が SSGA [1]に対し悪い結果を示すのは想定通りであったが，FM-HWGA が SSGA [1]に対しほぼ同等の性能であり，僅かに劣る結果となったのは想定外であった．要因は不明だが，HWGA と同様に CPU ボトルネックが発生した，もしくは FM-HWGA は他手法と比較すると理論上の計算量が 8%程度多いため，その計算量の増加分が結果に見える形で現れたと考えられる．

意図的に高負荷をかける SUPER FLIP 求解によるテストでは FM-HWGA と MM-HWGA 共に良好な結果が得られたが，汎用性を重視したランダムキューブ求解によるテストでは，CPU 動作周波数が不十分な条件下においては MM-HWGA による実装が最適であると言える結果となった．

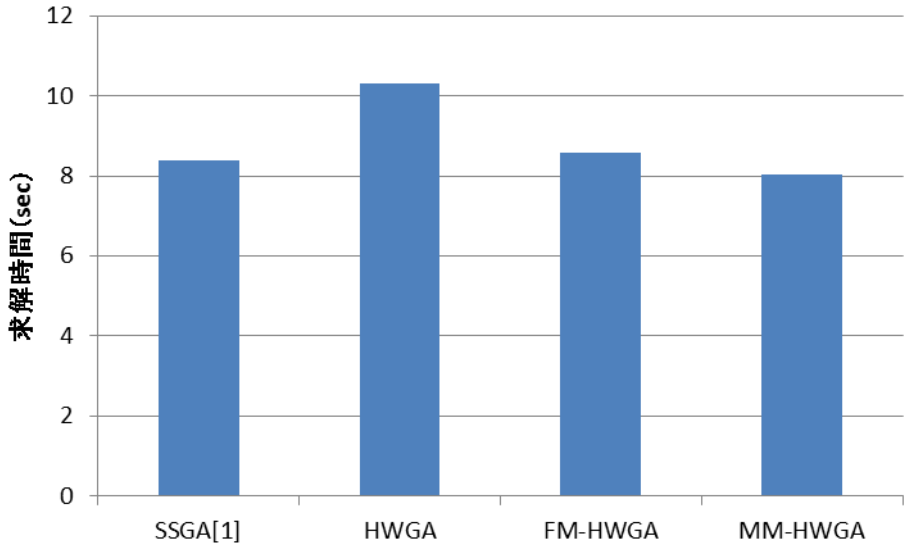

図 9-4　CPU 動作周波数 800MHz でのランダムキューブ 3000 個の平均求解時間

10. まとめ

10.1. 研究のまとめ

本研究では，IDA*アルゴリズムを CPU，GPU が混在したマシン上で効率良く働くプログラムの実装方法を検討した．本研究で提案した手法は 3 つあり，第一に，単純に CPU と GPU を全て使う実装をした HWGA，第二と第三に HWGA において CPU ボトルネックが発生したときに解決する手法である FM-HWGA と MM-HWGA である．

SUPER FLIP 求解という高負荷なテストにおいて，第一の提案手法である HWGA は CPU 性能が十分な場合は従来手法である SSGA [1]に対して良好な結果が得られたが，マシンの構成，特に CPU 性能に制限がある場合に性能劣化が顕著にみられる結果となった．対して，FM-HWGA と MM-HWGA の実装では，CPU の性能に依存せずに安定して高い性能が得られる結果となった．

対して，ルービックキューブソルバとして汎用的なテストであるランダムキューブ求解においては，SUPER FLIP 求解の高負荷テストと同様の結果は得られず，CPU 性能が十分な場合は FM-HWGA が比較した 4 手法（SSGA [1]，HWGA，FM-HWGA，MM-HWGA）の中で最も高い性能を示したが，HWGA, MM-HWGA の 2 つの提案手法は従来手法である SSGA [1]とほとんど変わらないパフォーマンスとなった．また，CPU 動作周波数を落とした条件でのテストでは，MM-HWGA が最も高速となり，良好な結果が得られると期待していた FM-HWGA が SSGA [1]に僅かに劣る結果となった．

上記を総合すると，HWGA, FM-HWGA, MM-HWGA の 3 つの提案手法は高負荷テスト，低負荷テスト，高 CPU クロック，低 CPU クロックなど様々な状況下において，必ずしも従来手法である SSGA [1]より高性能な結果が得られるとは限らないが，実験した全ての状況下において，SSGA [1]を性能で上回る手法を 1 つ以上提案できた．従って，本研究の課題である，計算機の計算リソースを全て効率良く使用した IDA*アルゴリズムの高性能化は実現できたと言える．しかしながら，ある手法が様々な環境下で必ず良いといえる結果は得られなかったため，更なる検討の余地が残る結果となった．

10.2. 今後の課題

本研究では，提案手法の 1 つである HWGA において CPU ボトルネックが表面化するケースの再現として CPU クロックを落とす手法を用いた．しかし，このボトルネックは本来ならば GPU 増設などの手段によって計算機の総合的な性能向上と同時に表面化する問題である．従って，改良型 HWGA として FM-HWGA，MM-HWGA の 2 つを提案したが，これらの手法が，例えば GPU4 台構成のマシン上で効率良く動作するかは検証が必要である．

また，本研究は民生用 CPU および民生用 GPU を用いた方法の検討のみであったため，Intel®Xeon Phi が搭載された場合や，CPU スレッドのみで数百スレッド使用可能な場合など，検討しきれていないハードウェア構成が複数存在する．よって，後発研究によりこれらのハードウェア構成で本研究の追試が行われ，より良い実装手法が提案されることを期待したい．

参考文献

[1] H. Hayakawa , H. Murao, “Optimal Rubik's Cube Solver on GPU,” GPU Technology Conference, 2013.

[2] R. E. Korf, “Finding Optimal Solutions to Rubik‘s Cube Using Pattern Databases.,” Proceed-ings of the Workshop on Computer Games , 1997.

[3] J. Sanders, E. Kandrot , 株式会社クイープ（訳）, CUDA BY EXAMPLE 汎用 GPU プログラミング入門, 株式会社インプレスジャパン, 2011.

[4] 倫. 奥山, 文. 伊野 , 兼. 萩原, “CUDA による全点対最短経路問題の高速化,” 情報処理学会研究報告計算機アーキテクチャ（ARC） , 2008.

[5] Rafia Inam, “A* Algorithm for Multicore Graphics Processors,” *CHALMERS UNIVERSITY OF TECHNOLOGY Department of Computer Science and Engineering Division of Computer Engineering,* 2010.

[6] Stefan Edelkamp , Damian Sulewski, “Parallel State Space Search on the GPU,” International Symposium on Combinatorial Search (SoCS 2009), 2009.

[7] Tomas Rokicki, Herbert Kociemba, Morley Davidson , John Dethridge, “God's number is 20,” 2010. http://www.cube20.org/.

[8] David Joyner , 川辺浩之 訳, “群論の味わい 置換群で解き明かすルービックキューブと 15 パズル,” 2010.

[9] Herbert Kociemba, “Cube Explorer,” 2014. http://kociemba.org/.

[10] Henry S. Warren, Jr., 滝沢 徹（訳）, 玉井 浩（訳）, 鈴木 貢（訳）, 赤池 英夫（訳）, 葛 毅（訳） , 藤波 順久（訳）, ハッカーのたのしみ 本物のプログラマはいかにして問題を解くか, SiB access, 2004.

謝辞

本研究を行うにあたり，研究のサポートをして頂いた笠井裕之准教授に心より感謝申し上げます.