

平成 24 年度卒業論文

GPU を用いたルービックキューブの最短解
探索の高速化

電気通信大学 電気通信学部 情報工学科

情報工学科計算機応用学講座

学籍番号 : 0811084

氏名 : 早川 広記

指導教員 : 村尾 裕一 准教授

平成 25 年 1 月 31 日

目次

1.	はじめに	3
1.1.	研究概要.....	3
2.	GPU コンピューティング	4
2.1.	GPU とは.....	4
2.2.	GPGPU とは.....	4
2.3.	CUDA とは	4
2.4.	NVIDIA 製 GPU のアーキテクチャ概要	4
2.5.	メモリ階層	5
2.6.	CUDA の実行モデル.....	6
2.6.1.	カーネル関数	6
2.6.2.	スレッド階層	6
2.6.3.	ワープ	7
2.6.4.	メモリアクセス.....	7
3.	ルービックキューブと関連研究.....	9
3.1.	概要	9
3.2.	ルービックキューブの各部位の名称.....	9
3.2.1.	ルービックキューブを構成する物理的な部品	9
3.2.2.	各面, 各部品の識別	10
3.3.	手順の表記法, 手数の考え方	11
3.4.	キューブ理論の基本定理	11
3.5.	関連研究 (求解アルゴリズム)	12
4.	ルービックキューブのデータ構造	13
4.1.	概要	13
4.2.	シール単位レベル	13
4.2.1.	表現法	13
4.2.2.	回転操作.....	14
4.3.	機械構造レベル	14
4.3.1.	表現.....	14
4.3.2.	データ構造.....	15
4.3.3.	回転操作.....	15
4.4.	実装レベル	15
4.4.1.	表現.....	15
4.4.2.	回転操作.....	16
4.4.3.	順列を数値に変換するアルゴリズム.....	17

5.	Korf の最短解探索アルゴリズム.....	20
5.1.	概要	20
5.2.	探索アルゴリズム	20
5.3.	探索済み状態の衝突回避方法	22
5.4.	距離関数.....	22
5.4.1.	距離関数概要	22
5.4.2.	距離関数を構成する部分群	23
5.4.3.	実装時の注意	24
5.4.4.	3 方向からの距離関数適用	25
5.4.5.	組み合わせを数値に変換するアルゴリズム	26
5.4.6.	距離関数生成アルゴリズム	26
6.	GPU 用ルービックキューブソルバの設計と実装	27
6.1.	並列化方針	27
6.1.1.	CPU 側の処理.....	27
6.1.2.	GPU 側の処理	28
6.1.3.	CPU と GPU の間で転送されるデータ	28
6.2.	探索済状態の衝突回避アルゴリズムの設計	29
6.3.	距離関数の設計	29
6.3.1.	複数の距離関数の併用.....	29
6.4.	実装時の各テーブルサイズのまとめ	30
6.5.	コンスタントメモリ, レジスタファイルの活用	30
7.	パラメータ設定と評価	32
7.1.	実験環境	32
7.2.	GPU での最大探索深度及びカーネル関数 1 回で解く問題数の上限の決定	32
7.3.	1 ブロックあたりのスレッド数の決定	33
7.4.	探索済状態の衝突回避アルゴリズムの決定	34
7.5.	評価関数の評価	35
7.5.1.	性能向上比による評価.....	35
7.5.2.	GPU 上での探索で発生する問題 1 つあたりのノード数による評価	36
7.6.	コンスタントメモリ, レジスタファイル使用の効果	36
7.7.	CPU との実行時間比較	37
8.	まとめ	38
8.1.	研究まとめ	38
8.2.	課題	38
	謝辞	39
	参考文献	40

1. はじめに

1.1. 研究概要

画像処理用のハードウェアである GPU (Graphics Processing Unit) は近年大幅に性能が向上し, FLOPS 値で比較すると CPU の約 10 倍の性能を持つようになった. この演算性能を汎用演算に応用したものが GPGPU (General-Purpose computing on GPU) である. GPU は小規模なプロセッサ集合体であるため, シミュレーションや科学技術演算など, 多量のデータに対し類似した処理を加える用途において最も効率よく働く. しかし, 命令管理や分岐処理などを行うユニットが複数のプロセッサで共有されているため, 分岐が多い処理では GPU の性能を十分に活かせない. それでもなお, GPU の演算性能は高く, Rafia Inam [1] はグラフの探索問題, Stefan Edelkamp ら [2] は 15 パズルやハノイの塔などのパズルの探索問題で GPU による高速化に成功している.

本研究ではルービックキューブの最短解探索を GPU を用いて行い, 高速化の可能性を検証した. ルービックキューブは取りうる状態数が非常に多い (4.33×10^{19} 通り) パズルとして有名であり現代の CPU を用いても最短解の探索には数分から数時間をする. ルービックキューブの最短解の探索アルゴリズムの研究は複数存在するが, 最も重要なアルゴリズムは Korf のアルゴリズム [3] である. 同アルゴリズムは探索木を巡回しながら解を探すというシンプルな物であるが, 効率の良い枝狩り関数が使用されており, 探索すべき枝を上手く選択する事で高速化を実現している. 更に, 枝同士に依存関係が無いため, 探索木の葉付近を無数の独立した小規模問題としてとらえる事ができる. この性質は GPU での実装において有利な特徴と言える.

本研究では, Michael Reid により実装された Korf のアルゴリズム [4] を並列化し, CUDA C を用いて GPU 用の実装を行い, 更に, アルゴリズムの中核と言える枝狩り関数の改良を試みた. その結果, GPU の活用により CPU 1 スレッドでの実行に対し最大 21.7 倍の速度向上を実現した.

なお, アルゴリズムの絶対性能は枝狩り関数に強く依存する. 従って, Kociemba による改良 [5] や Tomas Rokicki らによる改良 [6] など, 後発の研究で設計された枝狩り関数は本研究で使用した枝狩り関数と比較すると高性能であるため, より高速に動作する. しかし, これらは数学的な複雑さを持っているため, 本研究での研究対象として除外した.

2. GPU コンピューティング

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。当初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十～数百個搭載されており、その演算能力は 1 TFlops (CPU の約 10 倍) を超える物も少なくない。

2.2. GPGPU とは

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 は高い並列性を持つ処理を得意としており、条件が揃えば CPU の 10 倍以上の性能を発揮する。

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 と呼ばれる統合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 を用いて開発を行う。

2.3. CUDA とは

CUDA (Compute Unified Device Architecture) 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。本章では、NVIDIA 製 GPU のアーキテクチャ、および CUDA の実行モデルについて記述する。

2.4. NVIDIA 製 GPU のアーキテクチャ概要

NVIDIA 製 GPU は多量の演算コアを持ち、その最小単位を SP (Streaming Processor) と呼ぶ。SP は SM* (Streaming Multiprocessor) と呼ばれるクラスタ単位で管理され、SM

* 最新の NVIDIA 製 GPU では SMX と呼ばれる演算器クラスタを用いるモデルもあるが、本稿では扱わない。

には SP が複数個搭載される。SM には複数個の SP に加え、レジスタファイルや命令発行ユニットなどが搭載される。SM 内の SP はこれらのレジスタや命令発行ユニットを共有するため、SM 内の SP は全て同じ動作をする。図 2-1 は NVIDIA 製 GPU (GF100 世代) のブロック図であり、左が全体図、右が SM 部の拡大図である。また、図中で Core と表記されている部分が SP である。

SM 1つに搭載される SP 数は GPU の世代により異なり、本研究で使用する GeForce GTX 570 と同世代の GPU (GF110 世代) では 1 つの SM に 32 個の SP が搭載される。また、GPU に搭載される SM 数は GPU 毎に異なり、GeForce GTX 570 の場合は 15 個の SM が搭載され、計 480 個の SP を持つ。

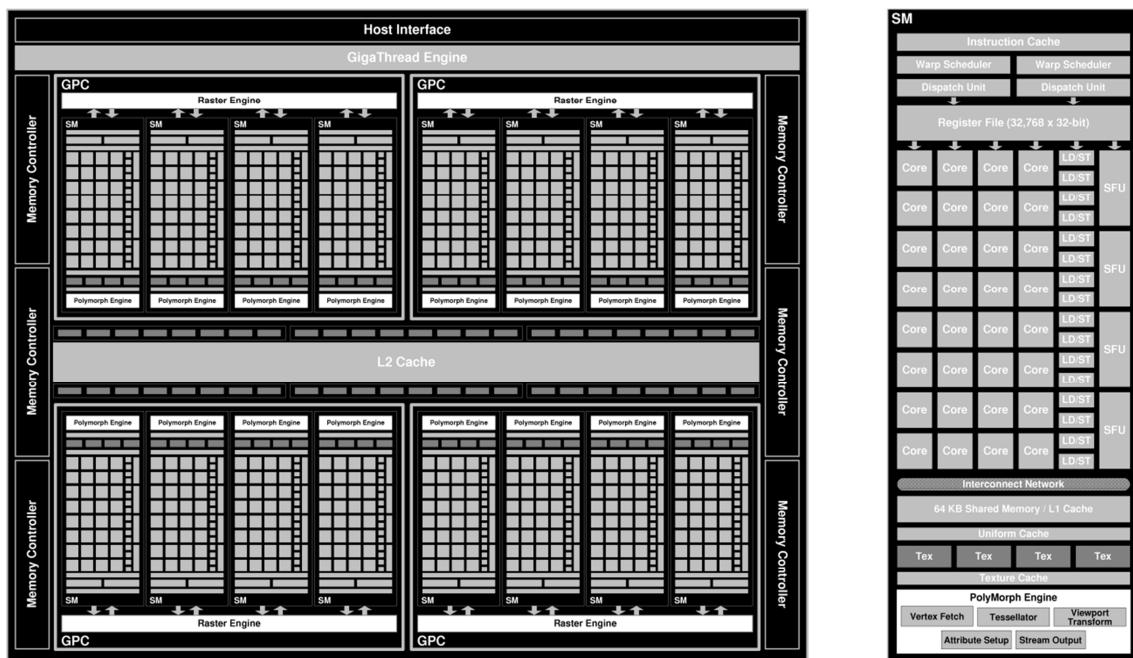


図 2-1 NVIDIA GF100 のブロック図 (NVIDIA_GF100_Whitepaper [7] より)

2.5. メモリ階層

GPU は複数のメモリ階層を持ち、アクセス可能なスレッドの範囲や、アクセス速度が異なる。主に、グローバルメモリ、L2 キャッシュ、シェアードメモリ、L1 キャッシュ、レジスタファイルがある。この他に、ローカルメモリ、コンстанトメモリ、テクスチャメモリがあり、これらはグローバルメモリと同じ領域に確保されるが、キャッシュの挙動が異なる。表 2-1 に各メモリの容量と特徴を示す。

CUDA では、これらのメモリ割り当てはキャッシュを除いて手動で行う必要があり、領域の確保時や変数の宣言時に指定する。GPU コンピューティングにおいて、これらのメモリ管理は非常に重要であり、主にグローバルメモリに対するアクセス遅延をいかに隠蔽するかが、高速化の上で重要な要素となる。

表 2-1 CUDA で使用できるメモリの種類 (Compute Capability 2.0 [†]以降)

グローバルメモリ	低速、容量が大きい(数百MB～数GB)。GPUチップとバス経由で接続されたDRAM上に確保される。全てのスレッド、およびホストからの読み書きが可能。
L2キャッシュ	高速、グローバルメモリのキャッシュとして働く。数百KB搭載され、コンパイラにより自動割り当てされる。中間コード(ptxファイル)から動作制御する事も可能。
シェアードメモリ	高速、容量が小さい(16KBまたは48KB)。GPUチップ上のメモリであり、各SM毎に搭載される。同一SM内のスレッド(同一ブロック中のスレッド)からアクセス可能。
L1キャッシュ	高速、容量が小さい(16KBまたは48KB)。シェアードメモリと同じ領域を使用し、シェアードメモリとL1キャッシュを合わせて64KB使用できる。コンパイラが自動で割り当てる。
レジスタファイル	同一路内からアクセス可能なレジスタ。1本あたり4byteであり、SM内に32768本存在する。
ローカルメモリ	レジスタが足りない時の退避先でありチップ外のDRAM上に確保される。使わない事が望ましいが、L1キャッシュが効くため、少量ならば速度への影響は少ない。
コンスタントメモリ	チップ外のDRAM上に確保され、GPUから書き込みができない(定数データとして扱われる)。64KB使用できる。
テクスチャメモリ	チップ外のDRAM上に確保され、GPUから書き込みができない(定数データとして扱われる)。64KB使用できる。2次元配列や3次元配列に対する空間的なキャッシュ操作が可能。

2.6. CUDA の実行モデル

2.6.1. カーネル関数

GPU は高い並列性を持つハードウェアであるが、先述の通り、搭載コア数に対し命令を管理するユニットが少ない。そのため、GPU では SPMD (Single Program Multiple Data) と呼ばれるモデルでプログラムが実行される。SPMD とは、1 つのプログラムで多量のデータを処理するモデルであり、ハードウェア上で 1 つのプログラムが多量に複製され、複製されたプログラムが並列実行され、一斉にデータ処理を行う。この SPMD の実行モデルにおける、複製元となるプログラムが、CUDA ではカーネル関数と呼ばれる。

カーネル関数は GPU 上でのプログラム実行時に最初に呼ばれる特別な関数であり、呼び出し時に GPU 内でカーネル関数を複製する個数を指定できる。カーネル関数内では、自身が何番目の複製であるかを知るためのビルトイン変数が使用でき、この変数を用いてデータの入力元の切り替えや内部処理の切り替えを行い、複製されたプログラムで処理を分担する。

2.6.2. スレッド階層

CUDA では、複製されたカーネル関数がそれぞれ 1 スレッドとして動作するが、このスレッドにも階層がある。まず、一番外側の階層をグリッドと呼び、1 つの GPU につき 1 つ存在し、1 つの GPU 内で動作しているカーネル関数の全てのスレッドはグリッドに含まれ

[†] Compute Capability : ハードウェアの世代毎に設定されている値で、この値毎にハードウェア構造およびプログラミング時の制約が異なる。本稿では 2.0 以上を仮定している。

る。次に、ブロックと呼ばれる階層があり、ブロックはグリッドの 1 段階内側にある階層であり、グリッド内に 1 つ、あるいは複数存在する。ブロックの内側の階層にスレッドがあり、ここが最も深い階層となる。

これらのスレッド階層は、GPU 上でどの範囲の実行ユニットが使用されるかを示し、グリッドが GPU、ブロックが SM、スレッドが SP に対応する。1 つのブロックは 1 つの SM 内で実行され、ブロック内のスレッドは該当 SM 内の SP で実行される。そのため、グリッド内に生成するブロック数、ブロック内に生成するスレッド数は実行効率に大きな影響を与える。これらの値はカーネル関数の呼び出し時に指定可能であり、実行効率の良い値を与える必要がある。

スレッドの階層はスレッド同士の連携のしやすさにも影響を与える。ブロック内では、スレッドは同一 SM 内での実行のため、シェアードメモリを介した高速な通信や、ハードウェア制御での高速な同期が行える。それに対し、異なる SM に存在するスレッド同士では、グローバルメモリを介した通信とソフトウェア制御の同期処理が必要となり、スレッド同士の連携がハイコストになる。

2.6.3. ワープ

SM 内のスレッドは、32 スレッドを 1 単位として実行し、この 1 単位をワープと言う。ワープ内のスレッドは全て同じ動作をし、分岐などでワープ内のスレッドが異なる処理をしなければならない場合は全てのスレッドが全ての分岐先を実行し、その後、分岐結果を評価した上で各スレッドの実行結果の整合性を取る。このように、ワープ内のスレッドが条件分岐の分岐先全てを実行しなければならない現象をワープダイバージェントと呼び、パフォーマンスに大きく影響する。また、ワープダイバージェントはワープ内の全てのスレッドの分岐先が同じ場合は発生しない。そのため、GPU 上でのプログラムは可能な限り分岐の無い実装をする、もしくは、分岐先が同一になる実装をする事が望ましい。

2.6.4. メモリアクセス

CUDA でのメモリアクセスは、使用するメモリ階層により性質が異なる。ここでは、複数あるメモリアクセスの中で代表的なメモリアクセスについて記述する。

2.6.4.1. グローバルメモリへのアクセス

グローバルメモリは DRAM[#]上に確保され、かつ GPU チップからバス経由での通信となるため、アクセス遅延が非常に大きい。しかし、バス幅が広い (GeForce GTX 570 では 320

[#] DRAM (Dynamic Random Access Memory) : キャッシュを利用した情報記憶装置であり、容量あたりの単価が SRAM (Static Random Access Memory : トランジスタのみで構成されるため非常に高速、GPU チップ上のメモリに使用されている) と比べて安いが、SRAM の 1/100 程度の性能。GPU のメインメモリとして 1 つのグラフィックボード上に数百 MB から数 GB 搭載される。

bit) ため、通信速度は高速である。そのため、GPU では複数スレッド（ワープの半分の **16** スレッド）のグローバルメモリへのアクセスを集計してアクセス回数を減らす機構を備えている。この機構をコアレッシングと呼び、たとえば、複数スレッドが 32bit の値を読み出す時、アクセス先が全て **1** 区画（**128 byte** アラインメントされた **128 byte** 領域）に存在する場合、グローバルメモリの該当区画に対して **1** 回のメモリアクセスがされ、必要な値が各スレッドに渡される。データ入出力の速度がボトルネックとなりうる処理（行列演算など）において重要な要素となる。

なお、本研究においては GPU に投入されるほぼ全てのスレッドが異なる動作をするため、コアレッシングによる動作速度改善が見込めない。そのため、この項目についての詳述はしない。

2.6.4.2. シェアードメモリへのアクセス

シェアードメモリは少量だが高速なメモリである。しかし、条件によっては性能低下の可能性もあるので注意が必要である。シェアードメモリ（L1 キャッシュ含む）は 32 個のバンクと呼ばれる区画に分割されており、複数スレッドがシェアードメモリにアクセスする際、異なるバンクへのアクセスは並列実行されるが、同一バンクへのアクセスは逐次実行となる。この現象をバンクコンフリクトと呼び、最悪の場合は性能が $1/32$ になる。

2.6.4.3. コンスタントメモリ、テクスチャメモリ

これらのメモリは低速な DRAM 上に領域が確保されるが、それぞれキャッシュが使用されるため高速な読み出しが可能である。特に、テクスチャメモリは **2** 次元配列などの不連続なアドレス空間を効率良くキャッシュする事も可能である。また、これらの領域は GPU 側では読み込み専用であり、内容の操作はホスト側から行う。これらのメモリは読み込み専用でキャッシュされるため、書き込みも想定しなければならないグローバルメモリのキャッシュと比べると効率の良い動作が可能である。

2.6.4.4. レジスタファイルへのアクセス

レジスタファイルはカーネル関数内で宣言した変数が割り当てられる **32bit** レジスタである（**64bit** 値は **2** 本のレジスタで表現される）。カーネル関数内で宣言した変数であっても配列の場合は低速なローカルメモリに割り当てられるので注意が必要である。SM 内に **32768** 本存在し、ブロック内で共有される。レジスタが不足した場合はローカルメモリに退避される。したがって、1 つのスレッド内で多量のレジスタを使用した場合はパフォーマンスの低下の原因となる。

3. ルービックキューブと関連研究

3.1. 概要

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。

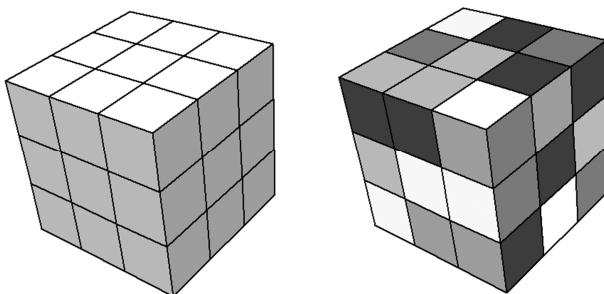


図 3-1 ルービックキューブ（左は初期状態、右は混合状態）

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関係の論述をする際、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物理的な部品

ルービックキューブは物理的に見ると、キューブ頂点を構成する 8 つの部品、キューブの辺を構成する 12 個の部品、キューブ中央を構成する 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ぶ。また、次項で詳述する表記法（シングスマスター記法）を使用するが、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 3-2）。他のスライスも同様の規則で名前が付けられる。

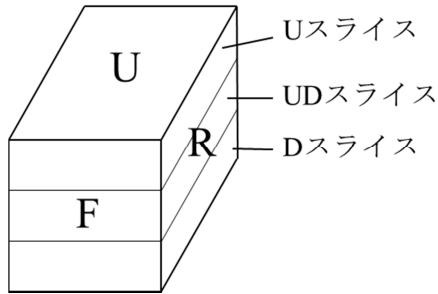


図 3-2 ルービックキューブを構成するスライスの名称

3.2.2. 各面, 各部品の識別

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. 各面の名称は, 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) とする(図 3-3).

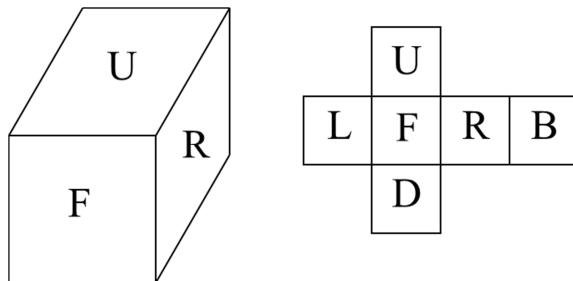


図 3-3 シングマスター記法に基づく各面の名称, 右は展開図

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. エッジキューブは同様に構成する 2 面, センターキューブは 1 面を指定する事で, それぞれの部品の識別が可能となる(図 3-4).

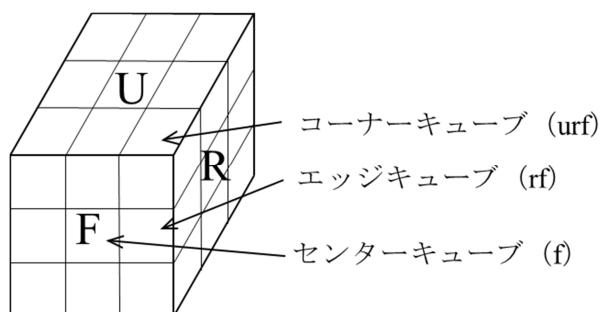


図 3-4 ルービックキューブの構成部品の名称と識別法

3.3. 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 F^2 , 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。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, F^2 , 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、面回転尺度 (face turn metric) と呼ぶ。回転角度を区別する 4 半回転尺度 (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. キューブ理論の基本定理

ここでは、ルービックキューブ研究で最も重要な、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事柄である。4.3 で定義する、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 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に対する処理の計算量が軽減される。この定理を活用したデータ構造は 4.4 にて定義する。

なお、定理内で現れる“置換”，“向き”的概念は 4.3 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
 - (2)コーナーキューブの位置がどのように置換されたか
 - (3)エッジキューブの向きがどの方向か
 - (4)コーナーキューブの向きがどの方向か

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

3.5. 関連研究（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 $4.33 \times 10^{19} \times 18 \div 2$ （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。現行の最短解探索アルゴリズムはこの巨大なグラフ上での最短経路探索を行っているが、そのためにはいくつかの段階を踏まなければならない。この章では、計算機向けのルービックキューブ求解アルゴリズムの初期の方法から、現行の方法までのアルゴリズムの紹介と変化を記す。

ルービックキューブを計算機で解くための研究は多数存在するが、最も重要なアルゴリズムは Thistlethwaite のアルゴリズムであり（1981年）[9]、ルービックキューブの任意の状態に対し 52 手以内の解を発見できるアルゴリズムを与えていた。このアルゴリズムは群論に基づく解法であり、キューブの任意の状態を、制約（エッジキューブの向き、コナーキューブの向き、…）の加わった状態に順次写してゆき、最終的に解に達する。この様にルービックキューブの状態を複数の群（ステップ）に分け、群同士の移動の探索問題にする手法を部分群法と呼び、このアルゴリズムでは、ルービックキューブを 4 つの部分群に分けて解いている。各部分群の代表元数は高々 108 万であるため、ある部分群の元を次の部分群の元に写す手順をテーブルに保持する事ができ、部分群同士の移動は、あらかじめ用意されたテーブルを参照しながら行う。このアルゴリズムは 1992 年に Kociemba により改良され、29 手以内の解を発見できるようになった [10]。このアルゴリズムでは [9] では 4 段階だった群を 2 段階に減らす事で発見出来る解を短くした。しかし、ステップ数が減った事により部分群の代表元数が増加し、群同士の移動がテーブルでは困難になった。

そこで、[10] は群同士の移動をテーブルによる方式から効率の良い探索アルゴリズムに置き換える事で、この 2 段階の求解アルゴリズムを実現した。探索アルゴリズムは一般的な深さ優先探索が用いられている[§]。後に、Richard E. Korf は最適解ソルバ [3] を設計した（1997）。[3] は深さ優先探索にて解を見つけるアルゴリズムであるが、高速化のための工夫がなされている。

また、[3] や [10] に用いられる探索アルゴリズムは、通常のグラフ探索では必須となる探索済み状態のチェックが不要となっているのも大きな特徴である。

[§] A*探索アルゴリズム [11] がベースとなっているが、A*探索アルゴリズムをルービックキューブの探索問題に最適化すると深さ優先探索に置き換わる。

4. ルービックキューブのデータ構造

4.1. 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定義を行う。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プログラム中で一般的に用いられるものであり、[4]や[10]でも同様の構造が使用されている。4.2でルービックキューブの最も基本的な表現、4.3でルービックキューブの物理的な構造の表現、4.4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、相互変換可能な対等なデータ構造であるが、キューブの問題入力やGUIによる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。

4.2. シール単位レベル

4.2.1. 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4-1に展開図にて示す。なお、図4-1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。

	0	1	2
3	U	4	
5	6	7	
8	9	10	16 17 18
11	L	12	19 F 20
13	14	15	21 22 23
	40	41	42
	43	D	44
	45	46	47

図4-1 シール毎の番号付けの例

4.2.2. 回転操作

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。

4.3. 機械構造レベル

4.3.1. 表現

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）。コーナーキューブは 8 通りの位置を持ち、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、で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、で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 4-2 に示す。
 x 印が回転量 0 を表す印（絶対位置）であり、 o 印が 3 面体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。エッジキューブは取り得る位置が 12 通りあり、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 4-3 に示す。

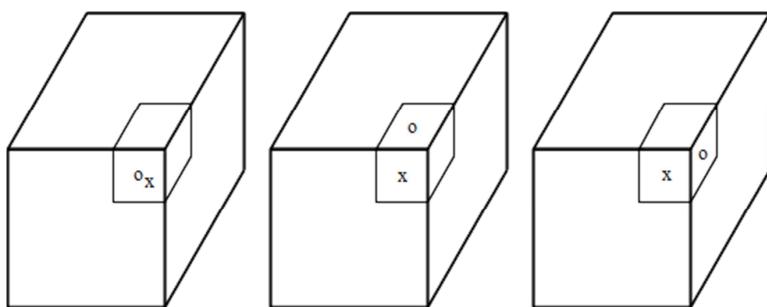


図 4-2 コーナーキューブのねじれ量の表現（左からねじれ量 0,1,2）

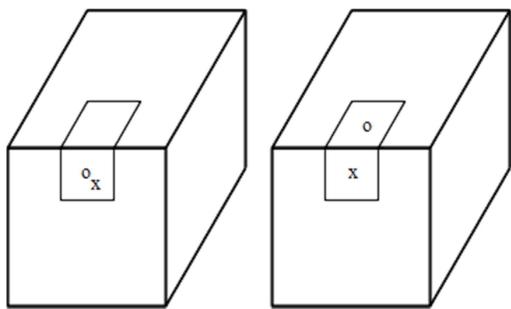


図 4-3 エッジキューブのねじれ量の表現（左からねじれ量 0,1）

4.3.2. データ構造

先述の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。

4.3.2.1. 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 通りあるので、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。

4.3.2.2. 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。

4.3.2.3. 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。

4.3.3. 回転操作

このデータ構造での回転操作は、4.2 の構造と同様に、変数の置換で表現できる。各操作は 4 か所のコーナーキューブ、4 か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。

4.4. 実装レベル

4.4.1. 表現

この表現法は、4.3 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。

4.4.1.1. 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 か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, 4.3.2.1 での 8 変数の内容は $8!$ 通りの組み合わせのどれかに属する. この 8 変数の内容を $0 \sim (8! - 1)$ の値のどれかに一意に変換する事が出来れば, 8 変数の内容を 1 つの変数に変換する事ができ, 逆変換も構成できる. 即ち, 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. この 8 変数を 1 つの数値に変換する規則は, 8 変数の状態と 1 つの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 4.4.3 に示す.

4.4.1.2. 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 個の変数を $0 \sim (3^7 - 1)$ の範囲の数値に変換する方法を考える. 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, 10 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 値を 11 桁の 2 進数として扱えば良い.

4.4.1.3. 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 組に分類し, 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 $12 \times 11 \times 10 \times 9$ 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.

4.4.2. 回転操作

回転操作は各変数に対する変換操作で記述される. 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.

4.2, 4.3 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 4-4 に示す. 図中の TwistID は回転操作 (R, F, ...) を表す. 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表現する.

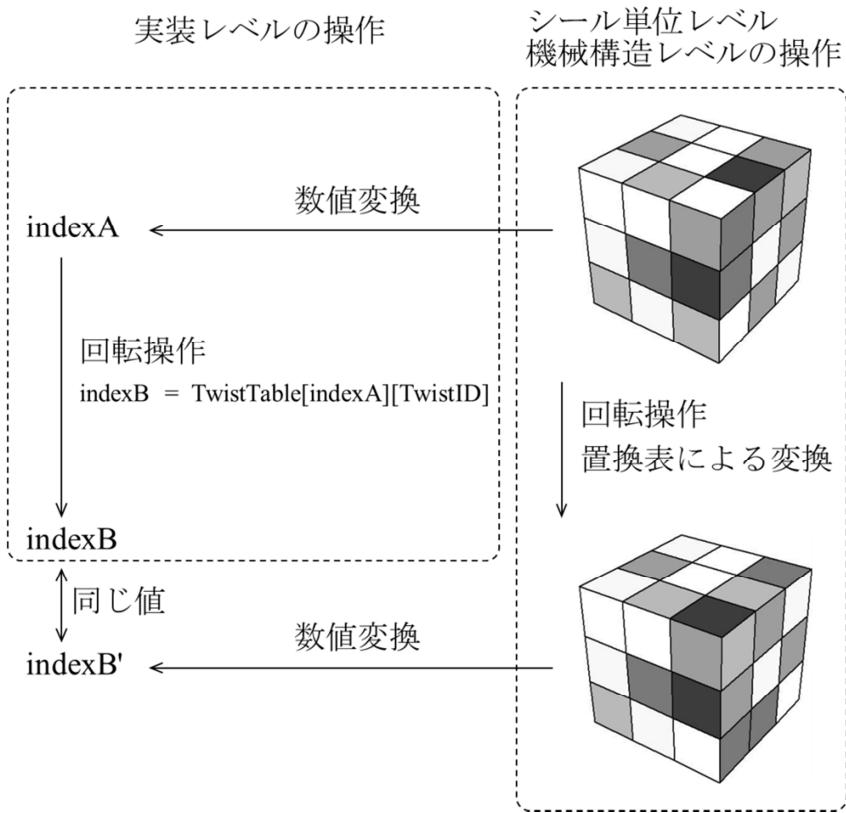


図 4-4 各データ構造の回転操作の相互関係

4.4.3. 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 4-5 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 つ目なので”1”， P の 3 は残された 2 つの値(0,3)のうち左から 2 つ目なので”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 4-6 に 8 個の値の順列を数値に変換するアルゴリズム、図 4-7 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。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` 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 4-6 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 [11] に記されている方法を使用している(図 4-8)。

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。

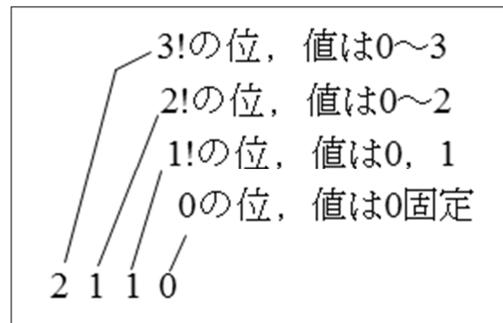


図 4-5 階乗進数の例

```

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 4-6 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 4-7 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 4-8 [11]による値 n に含まれる 1 のビットを数える関数

5. Korf の最短解探索アルゴリズム

5.1. 概要

Richard E. Korf は 1997 年に最短解を求めるアルゴリズムを設計した [3]. このアルゴリズムは [10] と同種の探索アルゴリズムを使用し、最短解を求める目的に最適化を施した物である。探索方法は一般的な深さ優先探索を用いているが、ルービックキューブを解く目的に特化している。最も特徴的なのは、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を防ぐ機構を備えていない。しかし、探索済み状態の再出現を大幅に減らす工夫がされているため、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）、その結果、高速なアルゴリズムを実現している。

5.2. 探索アルゴリズム

このアルゴリズムは、ルービックキューブの最短解を求めるために、深さ n 手の全探索を行う。探索深度は 1 から始め、深さ 1 手の全探索、深さ 2 手の全探索、…と、探索深度を 1 ずつ深くし、解の発見と共に探索を中断する。この深さ n 手の全探索は深さ優先探索にて行う。この深さ優先探索アルゴリズムを図 5-2 に示す。

図 5-2 のアルゴリズムは、簡略化してあるが、動作は Korf のものと同じである。また、図中に出現する構造体の定義は図 5-1 に示す。図 5-1 の L2 に出現する構造は省略する(4.4 の実装レベルの構造が適している)。このアルゴリズムは、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く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 では、明らかに不要な手を選ばないようにしており、その方法は 5.3 に詳述する。L34 では、距離関数を用いて不要な手を選ばないようにしている。この方法も後に詳述する (5.4).

```
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 5-1 図 5-2 で使用される構造体定義

```

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 5-2 Korf の探索アルゴリズム

5.3. 探索済み状態の衝突回避方法

このアルゴリズムでは、操作手順を工夫して選ぶ事により、探索済み状態の再探索の発生頻度を削減している。一般的な探索手段である、探索済み状態を保持する事による同一状態の再探索の発生の回避は行わない。具体的には、直前数手の操作履歴を監視し、特定パターンの操作手順を回避する事で、探索する手順を効率良く選んでいる。例えば、操作 U の直後に U^2 , U' を行う事は無い。なぜなら、 UU^2 の操作手順は U' に等しく、 UU' の操作手順は状態を変化させない。つまり、これらは探索不要な手順である。別の例を挙げると、手順 LR と手順 RL は同値である。よって、この 2 つの手順のうち、片方は探索する必要が無い。この様に、直前の手を監視しながら、直近数手の探索において同一状態が発生する事を防ぐ。Korf のアルゴリズムでは、直前 1 手を監視し、探索時に直近 3 手以内で同一状態が発生しないよう探索を行っている。直前 1 手を監視した場合に行うべき処理は次の 2 通りだけである。この 2 つの条件のみで、先の例で示された探索不要手順を回避する事が出来る。

- ・ 同じスライスを連続操作しない (UU^2 のような操作は不可)
- ・ 中間層を挟む 2 つの表層スライスの操作は特定順序のみ可能 (例、 LR は可、 RL は不可)

5.4. 距離関数

5.4.1. 距離関数概要

このアルゴリズムでは、距離関数による枝狩りを実装している。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、ある状態に対し n 手以内に解が無い、という情報を検出し、該当する状態を探索候補から除外する。

このアルゴリズムにおいて、距離関数はテーブルであり、テーブルには、ルービックキューブの部分群に対する全探索結果が格納されている。例えば、部分群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 3⁷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計算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見なす事も出来る。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、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、メインメモリに載る条件下で良い部分群を選択する必要がある。ここで、良い部分群とは、全探索

結果の最長深度が大きくなる部分群の事を表す。

5.4.2. 距離関数を構成する部分群

この章では、Michael Reid に実装された Korf のアルゴリズムが、距離関数の構成材料として使用している部分群とその実装法を記す。このアルゴリズムで使用される距離関数はルービックキューブの部分群に対する探索結果であるが、部分群として解説すると複雑なため、同値な群を持つパズルとして説明する。

アルゴリズム中で使用される部分群は、図 5・3 (図 5・4 にその展開図を示す) のような簡略化されたルービックキューブが構成する群に等しい。このキューブは完全なルービックキューブからコーナーキューブの位置情報とエッジキューブの位置情報の一部を取り去ったものである。コーナーキューブの位置は、U 面と D 面が同色であり側面が全て白色であるため、位置情報を区別できない。エッジキューブの位置情報については、エッジキューブが 2 種類（濃いグレー+白、薄いグレー+白）しか無いため、完全なキューブ (12 種類のエッジキューブ) と比較すると情報量が大きく削減されている。

図 5・3 の簡略化されたルービックキューブはとりうる状態数が 2217093120 であり、完全なルービックキューブの 1.95×10^{10} 分の 1 の状態数となる。この簡略化されたキューブの状態は、コーナーキューブの向き (3^7 通り) エッジキューブの向き (2^{11} 通り), 1 スライス (UD スライス) に属するエッジキューブの位置の組み合わせ ($C(12, 4) = 495$ 通り**) の 3 要素により決定される。

ここで、図 5・3 のキューブはエッジキューブ及びコーナーキューブの全てに向き情報が存在する事、コーナーキューブの位置はこのパズルを解く上で不要な情報である事、エッジキューブの位置情報は、UD スライスのエッジキューブ (あるいは UD スライスを除くエッジキューブ) がどの位置を占有しているか、で表現可能である事、の以上 3 点について注意されたい。

更に、重要な事は、図 5・3 の簡略化されたキューブは完全なルービックキューブからのマッピングが非常に簡潔である事である。距離関数を使用するためには、ルービックキューブの現在の状態を距離関数構成用の簡略化されたキューブにマッピングし、簡略化されたキューブから距離関数へのアクセスとなるためである。以下に、具体的なマッピング方法を記す

完全なルービックキューブの構造は 4.4 の方法が使用されていると仮定する。まず、コーナーキューブとエッジキューブの向き情報は完全に同一である。次に、UD スライスのエッジキューブが占有している位置情報であるが、元データは UD スライスの各エッジキューブの位置情報が格納されている (4.4.1.3)。ここからエッジキューブの識別情報を取り除き、占有位置の情報のみ残せば目的の情報が得られる。この処理も複雑な物ではない (5.4.5 に詳述)。また、この情報を 4.4 で定義した実装レベルの構造に追加し、元の構造のパラメータと同様に回転操作のテーブルを用意する事で、処理を更に簡潔にする事が出来る。

** $C(n, m)$ は組み合わせの数を計算する関数で、定義は $C(n, m) = n! / ((n-m)! m!)$ である。

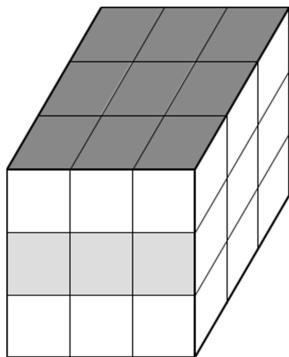


図 5-3 距離関数用の簡略化されたルービックキューブ

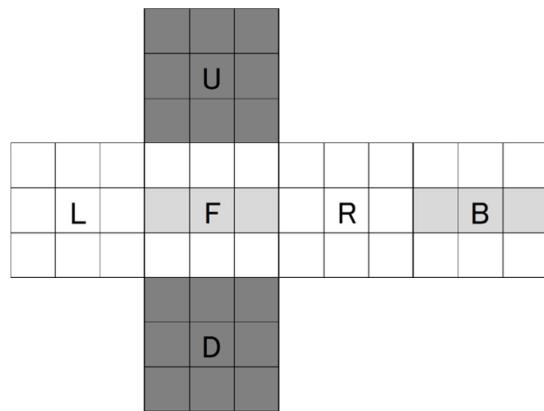


図 5-4 簡略化されたルービックキューブの展開図

5.4.3. 実装時の注意

この距離関数を実装する上で重要な点がいくつか存在する。最も重要な点は向き情報の基準の設定方法である。向き情報は 4.3.1 に示した方法を用いるが、その時の向きの基準の設定方法により距離関数の性能が変化する。ここで使用する距離関数を実装する上で適した基準設定は図 5-5 の方法である（×印が角度 0 の印）。この基準設定は図 5-4 に示した簡略化されたキューブの展開図から自然に導かれる設定である。更に、U, L2, F2 などの操作を加えても向き情報の状態が変化しないという、図 5-3 のキューブの特徴を引き継いでいる。この設定方法の場合は、この距離関数は最大で深さ 12 の距離を検出出来る。しかし、図 5-6 のような不自然な基準設定を行うと、距離関数で検出可能な最大の深さが 11 に減ってしまう。これは、向き情報が U, L2, F2 などの操作により、本来変化してはならない向き情報が変化てしまい、根に近い探索でノード数を増やしてしまうためである。従って、距離関数としてルービックキューブの部分群を使用する場合は、使用する部分群の形状に合わせて最適なデータ表現を選択しなければならない。

上記の他に対称性がある。この距離関数は対称性を考慮すれば、対称性を考慮しない場合に対し約 16 分の 1 のサイズで実現出来る。Michael Reid の実際のプログラムではこのテ

一テーブルは対称性を考慮し、約 16 分の 1 のサイズで格納している。しかし、これらは数学的な複雑さがあり、本稿のテーマ（GPU を用いる事による性能向上の可能性）と大きく異なる内容となるため、本稿では対称性の考慮は行わない。加えて、実験用プログラムにおいても、対称性の考慮をしたテーブルの使用はしていない。

図 5-5 自然な向き情報の基準設定

図 5-6 不自然な向き情報の基準設定

5.4.4. 3方向からの距離関数適用

5.4.2. で定義した距離関数はコーナーキューブ、エッジキューブの向きと UD スライスのエッジキューブに位置により定義されている。この距離関数は、ルービックキューブの対称性を考慮すると、キューブの向き情報に加えて LR スライスのエッジキューブの位置情報を加えた場合の距離関数としても使用する事が出来る。更に、FB スライスのエッジキューブについても同様である。即ち、この距離関数はルービックキューブの 3 方向 (UD スライスを使用する場合を U 面から適用、とすると、L 面、F 面からの適用) から適用する事が出来る。この時、各方向から適用した距離関数は異なる値を返す可能性が高い。この時、最も大きな値を距離関数の値として使用する事で、距離関数の精度を高める事が出来る。

5.4.5. 組み合わせを数値に変換するアルゴリズム

距離関数用テーブルのインデックス計算で使用される値に組み合わせの値がある(4個のエッジキューブが 12 カ所のどの位置を占有しているか). ここでは、この値を求めるアルゴリズムを記述する. ここに記す方法は [4] や [10] で一般に使用される方法で、[10] に詳述されているので詳細は省略する. 図 5-7 のアルゴリズムは、12 個のエッジキューブから 4 つのエッジキューブを選び、選ばれたエッジキューブが占有している位置を与え、戻り値は、4 つのエッジキューブの位置の組み合わせから、0~494 ($C(12, 4)$) の値のどれかに変換したものを一意に返す. 図中の combination 関数は $C(i, k)$ の値を返す関数である.

```
1: unsigned int edge_pos_combination(unsigned int pos[4]){
2:     int ret, i, k;
3:     ret = 0;
4:     k = 3;
5:     for(i = 11; i >= 0 && k >= 0; i--){
6:         if((i == pos[0]) || (i == pos[1]) || (i == pos[2]) || (i == pos[3])){
7:             k--;
8:         }else{
9:             ret += combination(i, k);
10:        }
11:    }
12:    return ret;
13: }
```

図 5-7 12 個中 4 つのエッジキューブの占有位置の組み合わせを値に変換する関数

5.4.6. 距離関数生成アルゴリズム

距離関数の生成は単純な探索を行う. まず、簡略化されたキューブの初期状態からインデックスを生成し、そのインデックスに対応する距離を 0 とする. 次に、キューブの初期状態を操作して隣接するキューブを生成する. 同時に距離 0 に隣接するインデックスを生成し、距離を 0+1 とする. すでに値が入っている場合は小さい方で上書きする. 同様の処理を全てのインデックスに対応した距離がセットされるまでループする.

6. GPU 用ルービックキューブソルバの設計と実装

この章では、Korf のアルゴリズムを並列化し、GPU で処理するためのアルゴリズムを設計する。

6.1. 並列化方針

ルービックキューブの最短解探索では、最大深度 N を固定した深さ優先探索が行われ、解が存在する場合は、 N 手目に解が存在する（5.2）。この探索の性質を利用して並列化を行う。 N 手の探索を行う場合に、探索深度を n, M に分ける事を考える（ M は固定）。探索の初期ノードから深さ n の探索を行うと、残りの探索深度 M の状態が k 個出現する（ただし、枝狩りにより深度 M に満たない場合有り）。この k 個の探索深度 M の状態を並列処理することで、実行時間の短縮を目指す（図 6-1）。ここで、重要な事は、 M の値を固定する事である。GPU で並列処理を行う場合、ある実行ブロックが終了するためには、ブロック内に存在するスレッド全ての終了を待つ必要がある。そのため、1 スレッドの処理量は一定であることが望ましい。しかし、 M の値は 1 変化すると処理量が 10 倍程度異なる。よって、GPU で処理をさせるためには、同一の M を持つ探索問題を与える必要がある。以上を考慮し、GPU 用のルービックキューブソルバを設計した。

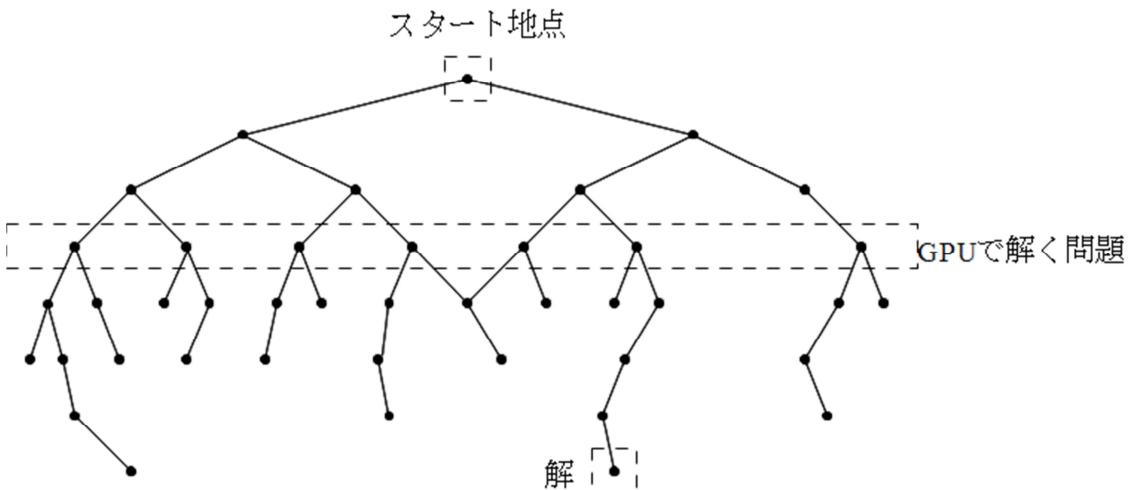


図 6-1 ルービックキューブの探索木と並列処理部分のイメージ。

6.1.1. CPU 側の処理

ルービックキューブの解探索では、探索深度 N の探索を、 $N=1$ から $N=20$ まで順次行い、解の発見と共に終了する。6.1 で示した並列化を行う条件は $N \geq M$ であるため、 $N < M$ の範囲の探索は CPU のみで行う。 $N \geq M$ の場合、CPU 側では深さ $(M-N)$ の探索を行い、発生したノードを保存する。この時保存されるノードは $(M-N)$ 手の操作が加えられたノードである。深さ $(M-N)$ の探索の発生ノード全ての出現、または保存されたノード数が一定値 K に達した時点で、保存されたノードを GPU に転送し並列で解き、GPU の応答を待

つ。ここで、保存するノード数の上限 K を設定する理由は、カーネル関数の実行時間を一定以下にするためである。CUDA ではカーネル関数の実行時間が一定時間を超えるとタイムアウトし、カーネル関数が強制終了する。これは、カーネル関数の実行中はシステム全体が応答不能に近い状態になるため、カーネル関数中に無限ループなどが発生した場合に強制終了させる機構が必要なためである。そのため、カーネル関数 1 回の実行で解く問題数に上限を設け、実行 1 回あたりの時間を制限し、長時間の処理は複数回のカーネル関数の実行をする事で対応しなければならない。この時、GPU で行う探索の最大深度 M 、GPU で並列に解く問題数の最大数 K の設定を行う必要があるが、これらの値は実験的に最も良い値を使用する。 M は最も処理時間が短くなる値、 K はカーネル関数 1 回の実行時間が 1 秒以内（タイムアウトはデフォルトでは 10 秒（Linux の場合））になる値とする。

6.1.2. GPU 側の処理

GPU では、CPU から転送された探索深度 M の問題を並列に全て解く。転送された問題の中に解ける物が存在した場合、その手順をメモリに書き込む。解が無かった場合は手順は書き込まれない。CPU はこの領域に解が書き込まれたかどうかを検査し、解けたかどうかの判断をする。

また、GPU では、問題 1 つにつき 1 スレッドで解き、ブロック数 (gridDim) とブロック内のスレッド数 (blockDim) は実験で最も良い結果が出た物を使用する。また、GPU で処理する問題数が blockDim の倍数でない場合は誤動作の原因となるため、CPU 側で予めダミーの問題を用意し、GPU で解く問題数を調整しておく。

6.1.3. CPU と GPU の間で転送されるデータ

6.1.3.1. CPU から GPU への問題の転送

ルービックキューブの状態は比較的大きな構造を持つため、そのままの構造を GPU に転送すると、転送量が増加し、オーバーヘッドの増加に繋がる。そこで、転送する情報は、解くべきルービックキューブの最初の状態と、深さ ($M \times N$) の探索で加えられた手順を転送する。手順は 1 手につき 18 通り考えられるため、5 bit の領域が必要である。この手順を 32bit 整数に格納する事を考えると、変数 1 つにつき 5 手の格納が可能となる。実際の実装では、手数を格納する変数を 2 つ用意し、10 手までの手順の格納を可能にしている。このような情報を元のルービックキューブの状態と共に転送する事で、元の構造ではキューブ 1 つあたり 24byte (実装に依存する) の転送が必要なのに対し、8byte の転送で目的の状態を再現する事が出来る。GPU 上での各スレッドは、自身が解くべき状態を問題の元データとそれに対する手順を用いてスレッド内で生成し、その問題を解く処理を行う。

6.1.3.2. GPU から CPU への、解情報の転送

GPU から CPU へ転送する情報は、手順情報を格納する要素数 20 の配列である。ルービ

ックキューブはどのような状態からでも 20 手で解く事ができるため [6], 配列の要素数は 20 で良い。手順情報の初手は無効な値で初期化しておき, GPU 内で解が発見されたら, 解の書き込みと同時に上書きされる。CPU 側は, この初手に格納されている値を見る事で, 解が発見されたかどうかを知る事が出来る。また, 発見された解の手数は, GPU 内での探索深度が M に固定されているため, 特別な変数を用意する必要が無い。ここで, 注意が必要な点がある。それは, ここで書き込む解は CPU 側での探索も含む完全な解である必要がある事である。CPU 側では, GPU に転送した問題のうち, 何番目の問題が解かれたかを知る事が出来ない(あるいは解けた問題のインデックスを転送する必要がある)。そこで, GPU 側で解を格納する時に, 6.1.3.1 で使用した GPU 内のスレッド自身が解くべき状態を作るために用いた手順と, GPU 内で発見された手順とを組み合わせて完全な解として格納する。このようにする事で, CPU と GPU との間の解のやりとりを簡単な処理にする事が出来る。

6.2. 探索済状態の衝突回避アルゴリズムの設計

この章では, 探索済み状態の衝突回避の方法を検討する。基本的な概念は 5.3 の方法と同様であるが, その方法を拡張して性能改善を試みる。5.3 では, キューブに加えた手順を, 直前 1 手を監視し, 明らかに無駄な手を選択しないようにしていた。これを, 直前 2 手の監視, 3 手の監視…と, 監視する手数を増やせば, 無駄な探索を削減出来ると考えられる。そこで, 直前に監視する手数を 1,3,4 手の 3 通り用意し, 性能のテストを行い最良の方法を選ぶ事にした。2 手の監視が無いのは, 直前 1 手の監視に包含されるためである。

6.3. 距離関数の設計

解探索時における発生ノード数は距離関数の実装にも大きく依存する。例えば, 5.4 で定義した距離関数では, U2 や L2 などの 180° 回転の操作では評価値が変化しない。即ち, 初期状態から 180° 回転のみでシャッフルされた問題は探索開始時点で距離関数が 0 を返す状態となり, 解に到達するまでに多量の探索ノードが発生する。つまり, 距離関数には不適なパターンが存在する。これは, 距離関数を部分群に基づき定義する以上, 避ける事が出来ない(距離関数を定義した部分群を変化させずに全体の状態を変化させる事が可能なため)。しかし, このようなパターンは GPU での実行に悪影響を及ぼす。例えば, 実行ブロック内に不適なパターンを持つ問題を抱えるスレッドが 1 つのみ存在する場合, ブロックを処理する実行時間の殆どをこの 1 つの問題に費やす事になるためである。ここでは, この不適なパターンを減らす方法を提案する。

6.3.1. 複数の距離関数の併用

5.4 の距離関数ではコーナーキューブの位置を全く評価できない。そこで, コーナーキューブの位置, 向きを部分群とした距離関数を新たに定義する。このコーナーキューブの全探索の最大深度は 11 である。この距離関数は, 5.4 の距離関数では機能しないパターンであった, コーナーキューブの位置は解から離れているが, 向きが揃っている状態をカバーする事が出来る。よって, 5.4 の距離関数と, ここで定義した距離関数を同時に使用する事

で、距離関数に不適なパターンの発生頻度を大きく削減する事が出来ると考えられる。しかし、この 2 つの距離関数を複合した場合でも、不適なパターンは存在する。例えば、エッジキューブが正しいスライスにあるが、内部の順序が異なる場合、などである。この場合の対応として、エッジキューブの位置の全探索を評価に加える事も可能であるが、エッジキューブの全探索は非常に大きな領域が必要なため、今回の実装からは除外した。

6.4. 実装時の各テーブルサイズのまとめ

実装には複数のテーブルを用意する必要があるが、主なものを表 6-1 に示す。表中の上から 4 つのテーブルは 4.4 で定義した実装レベルの構造に対する回転操作用テーブルである。上から 5 番目のテーブルは 5.4 に示した Michael Reid の評価関数を使用するために、4.4 の構造に新たに加えたパラメータに対して作用させる回転操作テーブルである。距離関数用のテーブルサイズは図の通りであるが、距離関数 B は距離関数 A と併用する追加テーブルを使用するので、距離関数 B のサイズは 1,152,636,480Byte となる。

表 6-1 用意したテーブルとテーブルサイズ

テーブルの種類	テーブルサイズ(Byte)
コーナーキューブの位置の回転操作テーブル	1,451,520
コーナーキューブの向きの回転操作テーブル	78,732
12 個中 4 つのエッジキューブの位置の回転操作テーブル	427,680
エッジキューブの向きの回転操作テーブル	73,728
12 個中 4 つのエッジキューブの占有位置の回転操作テーブル	17,820
距離関数 A 用テーブル(1 要素 4bit)	1,108,546,560
距離関数 B 用の追加テーブル(1 要素 4bit)	44,089,920
合計	1,154,685,960

6.5. コンスタントメモリ、レジスタファイルの活用

本アルゴリズムの実装では、容量の少ないシェアードメモリやコンスタントメモリに格納可能なテーブルは僅かしかないが、僅かであってもこれらのメモリを使用する事で性能の改善が見込める。よって、表 6-1 に示した各テーブルのうち、“12 個中 4 つのエッジキューブの占有位置の回転操作テーブル”をコンスタントメモリに割り当てた。なお、表 6-1 のテーブルのうち、回転操作を行うテーブルはキューブに対する操作 1 回につき 1 回または 3 回参照される（エッジキューブの位置、占有位置を変換するテーブルは 3 回の参照）。距離関数に対するアクセスは、距離関数 A はキューブ操作 1 回の後に 1~3 回、距離関数 B はキューブ操作 1 回の後に 0~1 回のアクセスがある。

次に、レジスタファイルの活用に移る。図 5-2 のアルゴリズムでは、L3 にて探索用配列を確保し、ポインタを用いて主な処理が記述されている。これは、CPU においてメモリアクセスはコストがあまり高くなく、かつ CPU の少ないレジスタを節約できるためである。対し、GPU では 1 ブロックにつき 32768 本のレジスタを使用でき、ブロック内のスレッド

数が 128 個存在していても、1 スレッドあたり 256 本のレジスタを使用する事が出来る。更に、GPU 上ではポインタ変数は 64bit 値であり、GPU には 64bit レジスタが無いため、ポインタを使用した処理はデメリットが多い。したがって、これらの処理を、レジスタを使用した処理に書き換える必要がある。

まず、図 5-2 の `p_node` であるが、この変数は配列に対するインデックスではなくポインタとする事で CPU 側で使用する変数を 1 つ削減する効果があったが、GPU においてその効果は薄い。よって、この変数は `node_arr` に対する添え字に置き換える。次に、随所に現れる `p_node[0]` と `p_node[1]` はメモリ上の値に対する処理となるため、非効率である。そこで、レジスタ上に `curr_node`, `next_node` といった変数を用意し、ループの先頭にてレジスタにロードし、`p_node` ポインタ（配列の添え字）が動く直前にメモリにストアする。これにより、処理の多くはレジスタに対するものとなり、性能低下を防ぐ事が可能となる。

7. パラメータ設定と評価

7.1. 実験環境

本研究では、実行環境として次の 2 つの環境を使用する。表 7-1 は GPU を使わずに計測する環境、表 7-2 は GPU を使用して計測する環境である。

表 7-1 CPU 用実行環境

OS	Ubuntu 11.10
CPU	Core i5 3570K 3.40GHz
RAM	32768 MB
GCC	version 4.6.1
最適化レベル	O3

表 7-2 GPU 用実行環境

OS	Ubuntu 10.10
CPU	Core i5 2500 3.30GHz
RAM	4096 MB
GPU	GeForce GTX 570
VRAM	1279 MB
GCC	version 4.4.5
CUDA	version 4.2
最適化レベル	O3

7.2. GPU での最大探索深度及びカーネル関数 1 回で解く問題数の上限の決定

ここでは、GPU 上で行う探索の最大深度 M と、カーネル関数 1 回で処理する問題数の上限 K の決定を行う。これらの値は、カーネル関数 1 回あたりの実行時間が 1 秒未満の条件下で最も高速な値を採用する。この実験より後に値が決定する値については以下の値を使用する。

- ・1 ブロックあたりのスレッド数 : 128
- ・探索済状態の衝突回避方法 : 直前 1 手を参照
- ・評価関数 : 2 つの評価関数の併用

また、各測定では、共通の問題を 1 つ設定し(表 7-3)、その問題を解くまでの所要時間(CPU 側の制御を含む)を調べる。同時に、カーネル関数 1 回あたりの実行時間も計測する。

実験の結果、表 7-4 と表 7-5 に示す結果が得られた。表 7-4 は各条件での所要時間を示し、表 7-5 はカーネル関数 1 回あたりの実行時間を示す。なお、表 7-5 の結果は、カーネル関数が処理した問題数が K に等しい場合のみの平均値であり、問題数が K 未満の場合のカーネル関数の実行は測定結果に含まれていない。

実験結果から、KとMの値は、K=524288, M=11が最善と判断した。カーネル関数の実行時間に関する条件を緩和すれば更に良い組み合わせが存在するが（表中では(K,M)=(65536, 12)），差は2%程度であるため、今回はプログラムの安定性を優先する事にした（カーネル関数の実行時間の条件が緩いほどタイムアウトによる探索失敗の可能性が高くなる）。

表 7-3 各パラメータ設定のための共通問題

シャッフル手順	U2 L B2 R' B2 L2 F D U2 F' B2 R' B U L2 D' B'
解の長さ	17手

表 7-4 KとMを変化させた時の処理時間 (sec)

K\ M	10	11	12	13
2048	31.5	20.0	16.7	16.5
4096	22.2	12.1	9.64	-
8192	17.3	8.36	6.72	-
16384	14.9	6.71	5.66	-
32768	13.5	5.69	5.03	-
65536	12.8	5.17	4.70	-
131072	12.3	4.94	-	-
262144	12.2	4.84	-	-
524288	11.8	4.77	-	-
1048576	11.9	4.83	-	-
2097152	12.0	-	-	-

表 7-5 KとMを変化させた時のカーネル関数1回あたりの所要時間 (sec)

K\ M	10	11	12	13
2048	0.0011	0.012	0.14	1.5
4096	0.0013	0.014	0.16	-
8192	0.0018	0.019	0.22	-
16384	0.0026	0.031	0.36	-
32768	0.0042	0.051	0.64	-
65536	0.0073	0.092	1.2	-
131072	0.013	0.18	-	-
262144	0.025	0.34	-	-
524288	0.049	0.68	-	-
1048576	0.096	1.1	-	-
2097152	0.19	-	-	-

7.3. 1ブロックあたりのスレッド数の決定

ここでは、1ブロックあたりのスレッド数を決定する。ここで決定する項目以外のパラメ

ータは、以下とする。

- ・GPU 上での最大探索深度 : 11
- ・カーネル関数の実行 1 回で解く問題数上限 : 524288
- ・探索済状態の衝突回避方法 : 直前 1 手を参照
- ・評価関数 : 2 つの評価関数の併用

実験は 7.2 と同様、共通の問題（表 7-3）を解くまでの所要時間を測定した。実験結果は表 7-6 である。表から、ブロックあたりのスレッド数は 128 又は 256 が最善である事がわかるが、本プログラムでは、ブロック数が多い方が有利（ブロック単位のスケジューリングにより SM が有効に働かない時間を低減できる）と考えられるため、この値は 128 が最善と判断した。

表 7-6 ブロックあたりのスレッド数を変化させた時の処理時間 (sec)

ブロックあたりのスレッド数	16	32	64	128	256	512
実行時間 [sec]	7.73	6.19	4.80	4.77	4.77	5.60

7.4. 探索済状態の衝突回避アルゴリズムの決定

この章では、5.3 の方法がどの程度効果があるのかを検証する。加えて、同手法を拡張した場合の性能も検証する。検証は次の 4 つの探索方法にて、深さ 1~10 手の全探索を行い、発生した状態数の合計を比較する事で行う。次に、比較対象となる探索方法 (A,B,C,D) を挙げる。なお、D は参考文献 [6] で正確な値を知る事ができるため、実際には探索を行っていない。

- A : ある状態に対し、直前に加えられた 1 手を監視し、探索状態の衝突を削減する
- B : 直前に加えられた 3 手を監視
- C : 直前に加えられた 4 手を監視
- D : 探索済み状態を保存し、深さ n までの正確な状態数を調べる

これら 4 つの探索条件にて探索を行った結果、次の結果が得られた（表 7-7）。この結果より、探索済み状態の保持が無い場合においても、探索済み状態の衝突が高々 5% 程度であり、効率良く探索を行う事が可能である事が示された。加えて、直前に参照する手数を増やした場合は理論上の性能は向上するが、その変化は僅かである事が確認された。更に、参照する手数を増やすと処理が複雑になり、GPU に適さない実装になる可能性がある。従って、探索済状態の衝突回避方法は、Korf と同様、直前 1 手を参照する方法に決定した。

表 7-7 探索済み状態の衝突回避アルゴリズムで出現する探索ノード数

探索深度	A	B	C	D
1	18	18	18	18
2	261	261	261	261
3	3501	3501	3501	3501
4	46755	46740	46740	46740
5	624123	623703	621648	621648
6	8331111	8322562	8265727	8240086
7	111207591	111054223	109909521	109043122
8	1484451135	1481880139	1461482192	1441386410
9	19815150303	19773842372	19433562052	19037866205
10	264501924111	263857266067	258411015689	251285929521

表 7-8 探索済み状態の衝突回避アルゴリズムで発生する衝突ノード数の割合

探索深度	A	B	C	D
1	0.000	0.000	0.000	0.000
2	0.000	0.000	0.000	0.000
3	0.000	0.000	0.000	0.000
4	0.000	0.000	0.000	0.000
5	0.004	0.003	0.000	0.000
6	0.011	0.010	0.003	0.000
7	0.020	0.018	0.008	0.000
8	0.030	0.028	0.014	0.000
9	0.041	0.039	0.021	0.000
10	0.053	0.050	0.028	0.000

7.5. 評価関数の評価

7.5.1. 性能向上比による評価

ここでは、6.3.1に示した方法と、5.4に示したMichael Reidのオリジナルの評価関数の性能比較を行う。まず、評価関数Aを5.4に示したオリジナル、評価関数Bを6.3.1で示した方法（評価関数Aとコーナーキューブの全探索結果を併用したもの）と定める。評価方法は、CPU1スレッドでの実行に対し、GPU移植後にどれだけの性能向上比が得られたかで評価する。2つの評価関数は、単純な比較ではBの方が高性能であり、実行時間も短くなる。これはGPU上の実行でも同様である。しかし、どちらがGPU上の実装に適しているか、という判断はできない。そこで、GPUでの実行に伴う速度向上比に着目する事にした。アルゴリズムは評価関数部分以外の変化が無いため、速度向上比が高い方がGPUに適していると判断できるためである。実験方法はこれまでと同様、共通の問題（表7-3）を用いて行う。この問題をCPUとGPUで解き、実行時間を計測する事で行った。その結果、表7-9に示す結果が得られ、評価関数Aでは18.4倍の性能向上、評価関数Bでは19.7倍の性能向上、加えて、絶対性能も評価関数Bの方が良い事が示された。

表 7-9 評価関数毎の CPU 及び GPU での実行時間と比率

	CPU[sec]	GPU[sec]	比率
評価関数A	102	5.55	18.4:1
評価関数B	94.2	4.77	19.7:1

7.5.2. GPU 上での探索で発生する問題 1 つあたりのノード数による評価

GPU 上での処理は、GPU に与えた問題の処理量が平均的であるほど良い。そこで、GPU で並列処理した問題 1 つあたりのノード数平均、および、その分散値を調べる事で、7.5.1 の評価を、より普遍的な尺度で行えると考えた。そこで、GPU 上の処理を CPU でエミュレートし、上記の平均値、および分散値を測定した。解く問題は表 7-3 の共通問題を使用した。

評価関数 A での測定結果を表 7-10、評価関数 B での測定結果を表 7-11 に示す。結果より、評価関数 B では評価関数 A と比較して、平均値、および分散値の減少が確認できた。よって、GPU 上での探索ノード数の安定性（分散値）からも評価関数 B の方が優位となる結果が得られた。

表 7-10 評価関数 A の結果

深さ11手の探索の回数	3,235,033
深さ11手の探索で出現するノード数平均	422×10^3
深さ11手の探索で出現するノード数の分散	500×10^{10}

表 7-11 評価関数 B の結果

深さ11手の探索の回数	3,235,033
深さ11手の探索で出現するノード数平均	272×10^4
深さ11手の探索で出現するノード数の分散	815×10^{10}

7.6. コンスタントメモリ、レジスタファイル使用の効果

ここでは、6.5 に示した、コンスタントメモリとレジスタファイル使用の効果を確認する。なお、これ以前の実験ではこれらの改良はすでに有効になっている。そこで、この章にてこれらのメモリ、レジスタファイルの使用を無効化した場合の実行時間を測定し、効果があることを確かめる。実験は表 7-3 の共通問題を解くための所要時間を測定する事で行った。表 7-12 にその測定結果を示す。表より、これら 2 つの改良を有効にした場合と無効にした場合とでは性能が約 2 倍程度異なり、行った改良が有効である事が確かめられた。

表 7-12 コンスタントメモリ、レジスタファイル使用の効果 (sec)

	コンスタントメモリ有り	コンスタントメモリ無し
レジスタの活用有り	4.77	5.33
レジスタの活用無し	9.05	9.24

7.7. CPU との実行時間比較

ここでは、CPU のみでルービックキューブを解く場合と、GPU を用いて解く場合との詳細な比較を行う。比較方法は、解の長さが同じ問題をランダムに 100 個解き、1 個あたりに要した平均時間を比べる。解の長さは 14~18 の 5 通りとする。なお、各問題は事前に解の長さを知っている必要があるため、ランダム生成したキューブを解き、解の長さ毎に分類し、解の長さ毎に 100 個集めた物を用意した。従って、プログラム毎に解く問題は共通である。比較対象は、CPU 1 スレッド、CPU 4 スレッド、GPU の 3 通りで行った。CPU 4 スレッドの実行で使用するプログラムは GPU 向けのアルゴリズムを Open MP にて並列化したものである。そのため、丁寧に CPU 向けのチューニングをした場合と比較すると性能は劣ると考えられるが、現行 CPU 1 つ分の性能の参考値として比較対象として加える事にした。表 7-13 に解の手数毎の処理時間、表 7-14 に GPU での処理時間を 1 に規格化した結果を示す。結果から、GPU での実行により、CPU 1 スレッドに対し最大 21.7 倍の速度向上を実現した。

解の手数	CPU 1 thread [sec]	CPU 4 thread [sec]	GPU [sec]
14	0.0732	0.0396	0.0317
15	0.970	0.263	0.0861
16	11.6	3.02	0.625
17	121	33.1	5.58
18	791	218	36.7

表 7-13 手数毎の処理時間の比較

解の手数	CPU 1 thread	CPU 4 thread	GPU
14	2.31	1.25	1.00
15	11.27	3.05	1.00
16	18.56	4.83	1.00
17	21.7	5.93	1.00
18	21.6	5.94	1.00

表 7-14 手数毎の処理時間の比率

8.まとめ

8.1. 研究まとめ

本研究では、GPUを用いてルービックキューブの最短解探索の高速化を図った。研究ではKorfのアルゴリズムを使用し、同アルゴリズムが生成する探索木の枝同士の独立性に着目して並列化を行った。CPUで探索木の根付近の探索を行い小規模問題を多量に生成し、生成された小規模問題をGPUで並列に探索するという処理モデルの実装を行った。また、探索中に使用される評価関数の構成法によりGPU移植時の性能向上率が変化する事を示し、GPU移植に有利な評価関数の構成法の1つを提示した。最終的なGPU実装においてはCPU1スレッドでの実行に対し最大21.7倍の速度向上を達成した。

8.2. 課題

本研究では、評価関数の構成時、キューブの対称性などを考慮したテーブルサイズの削減を行っていない。そのため、対象性を考慮した実装と比較すると、評価関数の性能は悪いものとなる。たとえば、[5]では、本研究で使用した評価関数より広い領域をカバーできる評価関数を使用し、Michael Reidの評価関数の13倍の性能を実現している。即ち、現在のGPUによるルービックキューブソルバの絶対性能が更に10倍以上向上する可能性が示唆されている。

更に、本研究では、CUDAで性能を出すには欠かせないシェアードメモリやコンスタンスマトリックスを少量しか使用していない。サイズの大きなテーブルが多く存在する本アルゴリズムでは、これらの使用は困難であるが、テーブルアクセス回数を増やす代わりにテーブルサイズを削減する、といった改良の可能性は残されている。

以上のように、本研究におけるアルゴリズムでは、GPU移植という意味では未成熟な部分が複数存在する。したがって、これらを改善する研究が行われる事を期待したい。

謝辞

本研究を行うにあたり， 実験環境の整備， 研究のサポートをして頂いた村尾裕一准教授に心より感謝申し上げます.

参考文献

- [1] Rafia Inam, “A* Algorithm for Multicore Graphics Processors,” *CHALMERS UNIVERSITY OF TECHNOLOGY Department of Computer Science and Engineering Division of Computer Engineering*, 2010.
- [2] Stefan Edelkamp and Damian Sulewski, “Parallel State Space Search on the GPU,” 2009.
- [3] Richard E. Korf, “Finding Optimal Solutions to Rubik’s Cube,” *Proceedings of the Workshop on Computer Games (W31) at IJCAI-97*.
- [4] Michael Reid. “Optimal Rubik’s cube solver,” 1997.
http://www.math.ucf.edu/~reid/Rubik/optimal_solver.html
- [5] Herbert Kociemba. “The Optimal Solvers.”
<http://kociemba.org/math/optimal.htm>
- [6] Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. “God’s number is 20,” 2010. <http://www.cube20.org/>
- [7] NVIDIA Corporation. “Whitepaper NVIDIA GF100,” 2010.
www.nvidia.com/object/IO_86775.html
- [8] David Joyner and 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.: 共立出版, 2010.
- [9] Morwen Thistlethwaite. “The 52 Move Strategy,” 1981.
<http://www.jaapsch.net/puzzles/thistle.htm>
- [10] Herbert Kociemba. “The Two-Phase-Algorithm,” 1992.
<http://kociemba.org/twophase.htm>
- [11] Jr. Henry S. Warren et al., 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.