卒業論文

題目

GPGPUを用いた UCT探索の高速化

指導教員

大野和彦　講師

2016年

三重大学 工学部 情報工学科
コンピュータソフトウェア研究室

袖山竜太朗 (413827)

内容梗概

今日, 様々なボードゲームの AI が発展し, プロレベルの実力を持つものも開発されている. 特に, UCT 探索を用いた囲碁 AI の成長が顕著である. UCT 探索はモンテカルロ法を木探索に適用したものであり, コンピュータ囲碁だけでなく, ゲーム AI 研究に大きく貢献した. しかし, これには膨大な計算が必要となる. そこで, 本研究では, CPU に比べ性能向上がめざましい GPU を汎用計算に用いる GPGPU に注目し, UCT 探索の並列化及び最適化を行った. その結果, 勝率に影響を与えることなく, CPU に比べて約 533 倍の高速化を実現した.

Abstract

Today, AI of various board games has evolved, and some AI with professional level ability are also being developed. Among them, the Go AI using UCT search has achieved remarkable growth. In recent years, they have reached a victory for professional players. UCT search is the improvement of Monte Carlo method to tree search. It is said that it revolutionized not only computer Go world but also game AI research. However, this requires huge calculations. Therefore, in this research, we focus on GPGPU, which uses remarkable GPU for general-purpose calculation compared with CPU, and this was used to speed up for the part suitable for parallelization of the UCT search. As a result, compared to the CPU, it was able to realize about 533 times faster speed without affecting winning percentage.

目 次

図 目 次

表 目 次

1　はじめに

1.1　研究目的

現在, 様々なボードゲームが存在するが, コンピュータゲームプレイヤではゲームの進行をゲーム木と呼ばれる木で表現し, その中で有利な手を求め解を得る. このゲーム木の探索には膨大な計算が必要であり, より高性能なコンピュータが求められている. そういった中で, 近年 CPU に比べ性能向上の著しいグラフィックエンジンに, 画像処理だけでなく CPU で行うような汎用的な計算をさせる GPGPU 技術への関心が高まっている. 本研究では NVIDIA 社が提供する GPGPU 用の SDK である CUDA を利用し, UCT 探索を採用した Nip のゲームプレイヤに GPGPU での並列化を適用し高速化を図った.

1.2 論文の構成

2 章では背景としてゲーム木探索の主なロジックと CUD, GPU の並列アーキテクチャについて解説する. 3 章では実装方法と更なる高速化について述べ, 4 章で単純な CUDA 版と CPU, 単純な CUDA 版と高速化を施した版の性能比較の評価結果を示す. 最後に 5 章でまとめを行う.

2　背景

2.1　コンピュータゲームプレイヤ

2.1.1　ゲーム木探索

一般的に, ゲームをコンピュータで解く場合にはゲーム木を作成し, その木の上で最適な解を探索する方法をとる. そのような方法の一つとしてミニマックス法が挙げられる. ミニマックス法とは, 対戦相手が（コンピュータが考える）最善の手を打ち返してくると想定し, ゲーム木の末端のノードにて盤面の評価を行う方式である. この場合, ゲーム木の深さによって計算量が大きく変わり, より先の手順まで読もうとすると膨大な計算によって時間がかかる. これの改良版にあたるアルファ・ベータ法についても, ゲーム木において不必要な計算を省いてはいるが, 同様に読む手順の量によって時間がかかってしまう. その他に, 後述するモンテカルロ法がある.

2.1.2　モンテカルロ法

モンテカルロ法とは, 乱数を用いたシミュレーションを何度も行うことにより近似解を求める計算方法である. この性質の利点は, ボードゲームにおいて評価関数を作成せずに済むことである. それに対して欠点となるのが, 高い精度を得るために試行回数が膨大になってしまうことである. しかし, 同じシミュレーションを順に処理していくような単純な繰り返し処理のため並列化に適している.

2.2 UCT 探索

2.2.1 Multi-Armed Bandit

モンテカルロ法でのプレイアウトの方法を改良し, 有利な手に多くのプレイアウトを割り当て, プレイアウトの回数が閾値を超えたら, 木が成長するように一段深い階層まで検索するアルゴリズム「モンテカルロ木探索」を持った「CrazyStone」の登場は, コンピュータ囲碁界だけでなく, ゲームＡＩ研究に革命を起こしたと言える. その理論的背景は「Multi-Armed Bandit」という, 統計学や機械学習の分野で研究されてきた問題の解決である. Multi-Armed Bandit 問題とは, 複数のスロットマシンがある状況で, 与えられたコインを使ってできるだけ多くの報酬を得るための戦略を考えるものである. この最善の戦略は 1985 年に見つけられているが, 計算量, メモリ消費ともに大きいため実用にはならない. そのため, 計算量が少なくて優秀な戦略が必要となる. Multi-Armed Bandit 問題にモンテカルロ法をあてはめると「全部に同じ枚数を投入して平均結果を比べる」ということになり, 優秀な戦略とは言い難い.

2.2.2 UCB(Upper Confidence Bound)

Multi-Armed Bandit 問題の疑似解を得る方法に, 「Upper Confidence Bound(UCB) ／信頼上限」という考え方がある. これをゲーム木探索にあてはめると, 確率的に結果が悪かった場合, 試した回数が多いほど悪手とみなし, 逆に少ない場合は善手である可能性があるとみなしそのままプレイアウトを続けるということである. 具体的な式は下のようになる. UCB 値が高い手からプレイアウトするように選ぶと, 浅い読みで良さそうな手を優先的に深く読む「最良優先探索」のようになる.

$$UCB値 \quad = \quad \bar{x_j} + C\sqrt{\frac{2logn}{n_j}}$$

$\bar{x_j}$ ： j番目のマシーンの報酬の期待値

c ： アルゴリズムの性格を決める定数

n_j ： j番目のマシーンに投入したコインの枚数

n ： これまでに投入したコインの枚数の合計

fig. 2.1: UCB の計算式

2.2.3 UCT(UCB applied Trees)

UCB を木探索に応用したものを「UCT(UCB applied Trees)」と呼び, UCB 値の高い候補により多くのプレイアウトを割り振り, 末端でのプレイアウト回数が閾値を超えたら, その手を展開する. UCT 探索の参考図を以下にに示す. 初期状態から現在の探索範囲を探索し, ノード 1・ノード 2・ノード 3 を得る. ノード 2 が閾値を越えた場合, ノード 2 を展開し, 一段深い階層を探索する. これを行うことにより, 探索回数が大きくなるに従って, 結果が期待値そのものに収束することが証明されている [1]. この UCT を最初に取り入れた囲碁ＡＩ「MoGo」が, 平手 9 路盤で初めてプロ棋士から勝ち星をあげた.

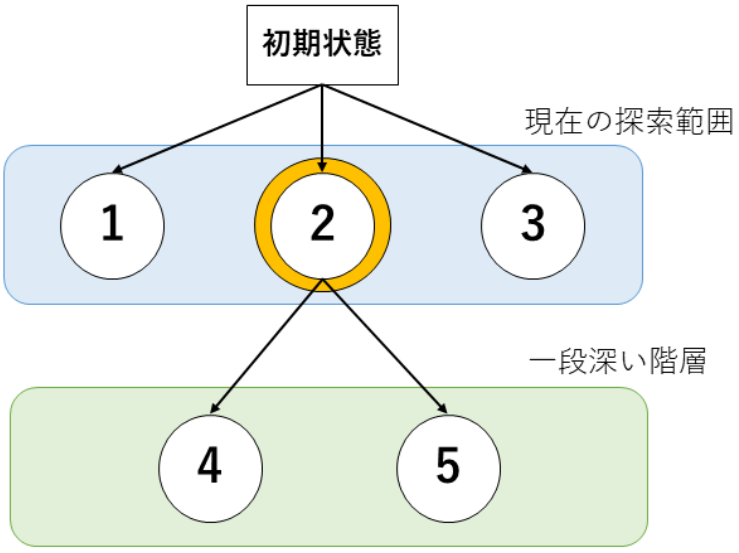

fig. 2.2: UCT 探索

2.2.4 ボードゲームにおける UCT 探索

ボードゲームにおける UCT 探索とは, 候補手のうちどれが最も勝ちに近いかを決めるための探索である. ある盤面の候補手を探し, その全てについて試行回数分プレイアウト (終局まで打つ) を行い, 得られた勝率を評価値の初期値とする. 評価値が最も高い手について, 再度試行回数分プレイアウトを行い, 候補手の評価値を更新する. これを指定した回数 (ループ回数分) 繰り返し, 最終的な評価値が最も高い手を最善手として選択する.

2.3 GPGPU

2.3.1 GPU

GPU とは, 画像処理を専門とする補助演算装置である. 一般的に, リアルタイムの画像処理は非常に高負荷な作業であるが, 処理内容そのものは単純であるためハードウェア化に向いており, CPU に比べて高性能化が著しい. この GPU の演算資源を画像処理以外の汎用計算に応用する技術を GPGPU という.

2.3.2 CUDA

CUDA[2] は NVIDIA 社より提供されている GPGPU 用の SDK であり，ユーザーは C 言語を拡張した文法とライブラリ関数を用いて GPGPU プログラムを開発することができる．CUDA プログラムでは, まず CPU 側 (ホスト) で必要なデータを作成し，GPU 側 (デバイス) に転送する．次にホストが GPU に実行させる関数 (カーネル) を起動させ，デバイスがカーネルを実行し，処理を開始する．最後に結果をホストに転送し，デバイスから受け取ったデータをホストが処理することで終了する. また, CUDA にはビルトイン変数というものが用意されており, 各スレッドは固有の ID を持ち, 配列の添え字にその ID を表す組み込み変数を利用することにより個々の要素を並列に処理することが可能である.

2.3.3　スレッドとブロック

スレッドとはGPUでのカーネル実行の最小単位である．CPUではそのコア数とほぼ同数のスレッドを動作させるのが一般的だが, GPUでは数万〜数十万のスレッドを並列・並行に動作させることにより, 高性能を達成している．

ブロックはスレッドをまとめたものであり, 1つのブロック当たり最大1, 024スレッド格納できる．x方向, y方向, z方向に8スレッドずつ, つまり「1ブロックに$8 \times 8 \times 8$スレッド」と3次元的表現で管理することができる．また,「1ブロックに512スレッド」「1ブロックに16×16スレッド」と1次元的, 2次元的にまとめることも可能である．

2.3.4　ワープ

ワープとは32個のIDが連続するスレッドで構成され, ワープ内のスレッドはSIMD方式で実行される．つまり, ワープ内のスレッドはすべて同じ命令を実行する．

2.3.5　シェアードメモリ

NVIDIA 社の GPU は並列処理に用いる大量のデータを効率よく扱うために階層構造のメモリを持つ．CUDA を用いた GPGPU プログラミングを実装する上で重要となるものが, デバイスメモリ, シェアードメモリの 2 つのメモリである．デバイスメモリは最大で 12GB と容量が非常に大きいが, レイテンシも非常に大きい．これに対して, シェアードメモリは容量が小さいが, アクセスが非常に速く処理の高速化において重要である．

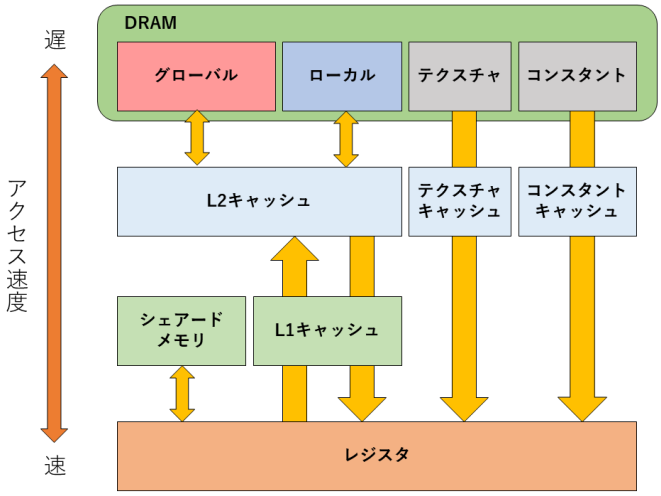

fig. 2.3: GPU の並列処理アーキテクチャ

2.3.6　コンスタントメモリ

コンスタントメモリはオンチップのキャッシュを持つ読み取り専用メモリ領域である．メモリ領域自体はデバイスメモリ上の専用領域にあるが, キャッシュがオンチップであるため, 最適化によって高速に処理することが可能となる.

2.3.7　ストリーム

CUDA におけるストリームは, CUDA の一連の非同期操作のことであり, GPU の処理を管理するキューとしての役割がある. デフォルトではストリームは 1 つのみ用意されており, GPU へ発行した命令は発行された順に従って逐次的に実行されることが保証されている . また, ストリームは複数生成することができ, プログラマが任意のストリームに命令を投入していくことが可能になっている. ストリーム間で異なる命令を処理することができるため, 分割した各データに対するメモリコピー及びカーネル実行を各ストリームに割り当てれば別々に結果を得ることも可能である. また, ストリーム同士には依存関係がなく独立に処理されるため, ストリーム同士の並列処理が可能である. これにより, 計算とデータ転送をオーバーラップすることで処理時間をさらに短縮できる.

3　実装方法

3.1　概要

逐次版では与えられた盤面からプレイアウトする動作を 試行回数分 × (候補手数 + ループ回数分) 実行する. そのため, 試行回数やループ回数が増えるほど処理時間も増える. そこで, GPGPU による UCT 探索の高速化のために, 現在の盤面に対して, 候補手の場所に着手した状態から終局まで打ち進める処理を並列化する. 従って, CUDA による並列化実装としては, 1 スレッドが 1 プレイアウトを担当し, 終局後に勝敗判定を行い, その結果を排他的加算により計算するようにした. さらに, シェアードメモリ・コンスタントメモリ・ワープシャッフル・ストリームといった CUDA 特有の機能を利用することにより, さらなる高速化を実現した.

3.2 実装

3.2.1 逐次版の作成

本研究では, GPGPU を用いた UCT 探索の高速化が目的のため, 比較対象となる CPU のみでの UCT 探索を用いた Nip プログラムの作成を行った. 逐次処理版の Nip プログラムから並列化を適用していき各段階について思考にかかる時間を計測し, 最終的にはどういったパラメータ, データ構造やメモリの使用方法が高速化に繋がるかを測定する.

3.2.2 CUDA による並列化実装

デバイス側に必要な変数を割り当て, 試行回数分のループを行うのではなく, スレッドを試行回数分起動するようにした. これにより, 逐次に行っていたプレイアウトを試行回数分, 平行並列に実行できる. しかし, これによってデバイスメモリに対する複数スレッドの同時書き込みも発生する. そのままでは結果に不整合がおこるため, 排他的に加算を行う atomicAdd() 関数を使用した.

3.2.3　シェアードメモリの利用

カーネル関数において高い頻度でアクセスする盤面のデータ構造をシェアードメモリに割り当て, より高速なアクセスを実現した. ブロック内スレッド数分の盤面配列を確保し, 添字 $tId = threadidx.x$ として使用した. 例えば, ブロック内スレッド数が 256 の時には以下のようになる. この際, 競合読み取り, 競合書き込みを解消するため, 同ブロック内のうち 1 スレッドのみが読み書きを行うようにした.

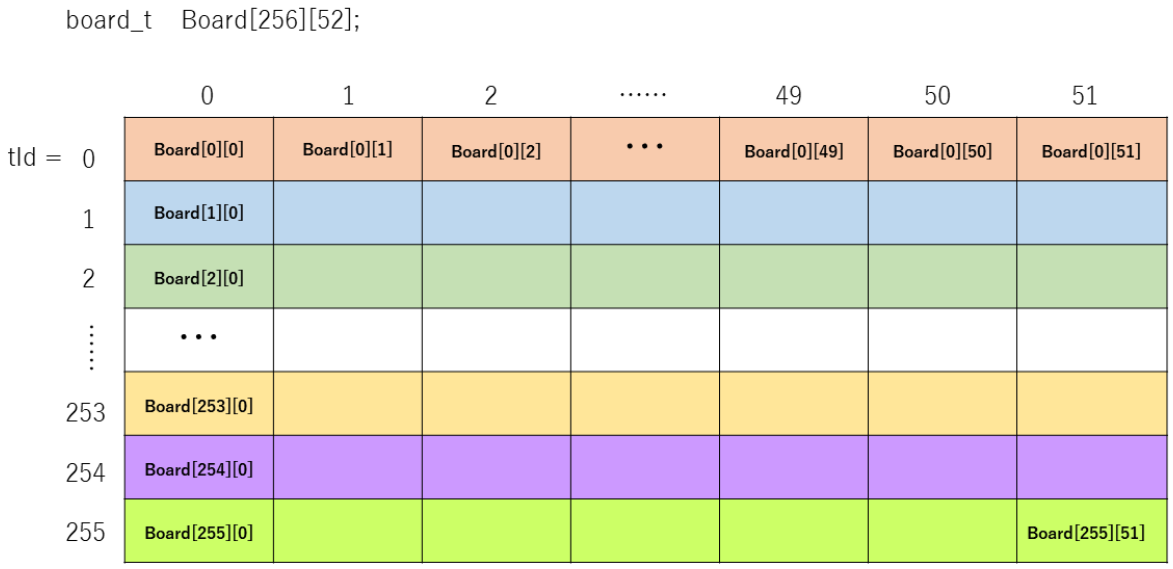

fig. 3.4: シェアードメモリ上のデータの割り当て

3.2.4　バンクコンフリクト

シェアードメモリへのリクエストが同じメモリバンクに属している複数のアドレスにアクセスする場合は, バンクコンフリクトが発生し, そのリクエストがリプレイされ, 遅延の原因となる. 今回は, ブロードキャストアクセスと呼ばれる, 1 つのバンク内の 1 つのアドレスにアクセスする状況が発生したため, ワープシャッフルを利用することにより改善することができた.

3.2.5　ワープダイバージェンス

GPU は比較的シンプルなデバイスであり, 複雑な分岐予測メカニズムは組み込まれていません. ワープ内のスレッドはすべて同じサイクルで全く同じ命令を実行しなければならないため, ワープ内のスレッドが異なる分岐パスを選択した場合, ワープは各分岐パスを逐次的に実行します. 今回はこれを避けるために, 三項演算子を利用した. これにより, わずかではあるが, ダイバージェンスを軽減することができた.

3.2.6　コンスタントメモリの利用

カーネル関数において, 各スレッドは同一盤面の状態をコピーして使用するため, コピー元の盤面データの操作を行わない. そこで, コピー元の盤面データをコンスタントメモリに割り当て, 高速なアクセスを実現した. コンスタントメモリは読み取り専用であるため, 盤面を表す構造体から書き込みを行わない部分のみを新たな構造体として, コンスタントメモリに割り当てた.

3.2.7　ビット演算

2 のべき乗での乗除算などは, ビット演算を用いたほうが高速であるというのは自明である. 今回はカーネル関数内の演算をビット演算化し, 高速化を図った.

3.2.8　ワープシャッフルの利用

CUDA には同ワープ内の別スレッドが持つレジスタの値を受け渡すための命令が備わっている. 通常, レジスタの値をスレッド間で共有するためにシェアードメモリなどを用いる必要があるが, ワープシャッフル命令を用いることで, シェアードメモリよりも高速な値渡しが可能となる. ただし, 同ワープ内 (32 スレッド) に限定されるため, 汎用性は劣る. 本研究では, 構造体の書き込みを行う部分の盤面データを同一ワープ内の各スレッドが持つ必要があることに着目し, あるスレッドが持つ盤面データに対してワープシャッフルによるスレッド間での値共有を行うことにより, 高速な値渡しをを実現した.

3.2.9　ストリームの利用

CUDA のストリームを用いて, カーネル実行とメモリ転送のオーバーラップを行った. また, ある盤面に対して, 各候補手に対して行う処理に依存関係がないことから, 1 つの候補手に行う一連の処理を 1 つのストリームに割り当て, ストリームの並列処理を行うようにした. この際, コンスタントメモリに割り当てるデータ, および盤面構造体の書き込みを行わない部分については, 一度だけメモリ転送を行うようにした. さらに, GPU と CPU の実行のオーバーラップも行うことにより, さらなる高速化を実現することができた.

4　評価

評価には Nip を用いた．Nip とは, 盤面が円形のオセロであり, 不動石がない．そのため, 評価関数の作成が困難であり, UCT に適している．

初めに, 1 手打つ (着手) までに必要な処理時間[1][2] について, 単純な逐次版と, CUDA による並列化やシェアードメモリ利用・オーバーラップといった高速化を段階的に施したものとの比較を table．4.1 に示す．次に, 試行回数とブロック数を変更し, 着手までに必要な処理時間を計測したものを table．4.2 に示す．[3] 実行環境としてメモリ：2GB, CPU：Intel(R)Pentium(R)CPU G640 2．80GHz, GPU:GeForce GTX 750 を搭載した計算機を使用した．

[1]パラメータは C=0.15, ループ回数 50 回とした．

[2]ここでの処理時間とは, 最善手の選択にモンテカルロ法を用いる AI(先手) と UCT 探索を用いる AI(後手) の対戦における, 1 ゲーム中の着手にかかる最大処理時間を 100 ゲーム分で平均したものである．

[3]測定に用いたプログラムは全ての最適化を施してある．

table. 4.1: 各実装方式での処理時間

試行回数	高速化段階	処理時間 (秒)	対 CPU 比
16384(ブロックサイズ:256)	CPU 版	668.7449	–
	CUDA 化	2.664890	250.947
	キャッシュ	2.295855	291.284
	SoA	2.268247	294.829
	シェアードメモリ利用	1.468806	455.298
	コンスタントメモリ利用	1.426397	468.835
	ダイバージェンス	1.424858	469.341
	ワープシャッフル	1.402862	476.700
	ビット演算	1.339398	499.288
	バンクコンフリクト	1.316568	507.946
	オーバーラップ	1.312127	509.665

table. 4.2: CPU 版との CUDA 版の処理時間

試行回数	ブロックサイズ	処理時間 (秒)	対 CPU 比
1, 024	32	0.244292	156.006
	64	0.244181	156.077
	128	0.244238	156.041
	256	0.245555	155.204
	512	0.255051	149.425
	CPU 版	38.111025	–
16, 384	32	1.255361	532.711
	64	1.273911	524.954
	128	1.289875	518.457
	256	1.312127	509.665
	512	1.340727	498.793
	CPU 版	668.744911	–
32, 768	32	2.447768	505.398
	64	2.343265	527.937
	128	2.373219	521.274
	256	2.413728	512.526
	512	2.462032	502.470
	CPU 版	1237.097222	–

4.1 考察

table. 4.1 からは, ビット演算・シェアードメモリ・コンスタントメモリ・ワープシャッフル・バンクコンフリクトの利用が, 大きな速度改善につながったといえる. table. 4.2 を見ると, 試行回数を増やした方が, CUDA 化の恩恵が大きいことが分かった. そして, ブロックの数を多くし, 1 ブロックあたりのスレッド数を減らすことでより高速な処理が可能となった. これは, 一般的な GPGPU において, あまり当てはまらないことである. また, スレッド数, ブロックサイズを最適化することにより, 最終的には CPU 版のものと比較して約 533 倍もの高速化を実現することができた.

5　おわりに

本研究では, UCT 探索を用いた Nip を CUDA プログラミングによる並列化, そして高速化を行い, 試行回数・ブロック数・スレッド数等のパラメータを変更し CPU 計算の場合との比較・評価を行った. 本手法を用いることにより, 最終的には CPU に比べて 533 倍程度の高速化を実現することができた. 単なる CUDA 化だけではなく, Nip の盤面データに対してシェアードメモリ, コンスタントメモリを用いることでメモリアクセスを効率化し, バックコンフリクトの解消やワープシャッフル活用より, 高速なアクセスを可能にした. また, Bit 演算やオーバーラップを行うことでさらなる高速化を実現することができた. 今後の課題としては, データ構造, メモリの使用方法の工夫を活かすことによる囲碁などの全体の局面数がより多いものへの応用が挙げられる.

謝辞

本研究を進めるにあたり, ご指導を頂いた卒業論文指導教員の大野和彦講師に感謝致します. また, 日常の議論を通じて多くの知識や示唆を頂いたコンピュータソフトウェア研究室の皆様に感謝します.

参考文献

[1] Levente Kocsis Csaba Szepesvari,Bandit based monte-carlo planning,Machine Learning:ECML 2006,282-293,2006/1/1

[2] NVIDIA Developer Zone, http://www.nvidia.co.jp/object/cuda_learn_jp_old.html

[3] 美添一樹, モンテカルロ木探索-コンピュータ囲碁に革命を起こした新手法, 情報処理学会, Vol.49 No.6, p686-693, June 2008

[4] Active Learning: Bandits, Advanced Course in Machine Learning Spring 2010, Handouts are jointly prepared by Shie Mannor and Shai Shalev-Shwartz, http://www.cs.huji.ac.il/∼ shais/Lecture1.pdf

[5] NVIDIA CUDA Reference Manual, http://www.cs.mu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/CUDA_Toolkit_Reference_Manual.pdf#page84

[6] John Cheng/Max Grossman/Ty McKercher, NVIDIA CUDA C プロフェッショナルプログラミング, 株式会社インプレス, 2015 年 9 月 21 日初版第 1 刷発行

A Nip

Nip とは, 円形盤で遊ぶ Othello & Reversi である.

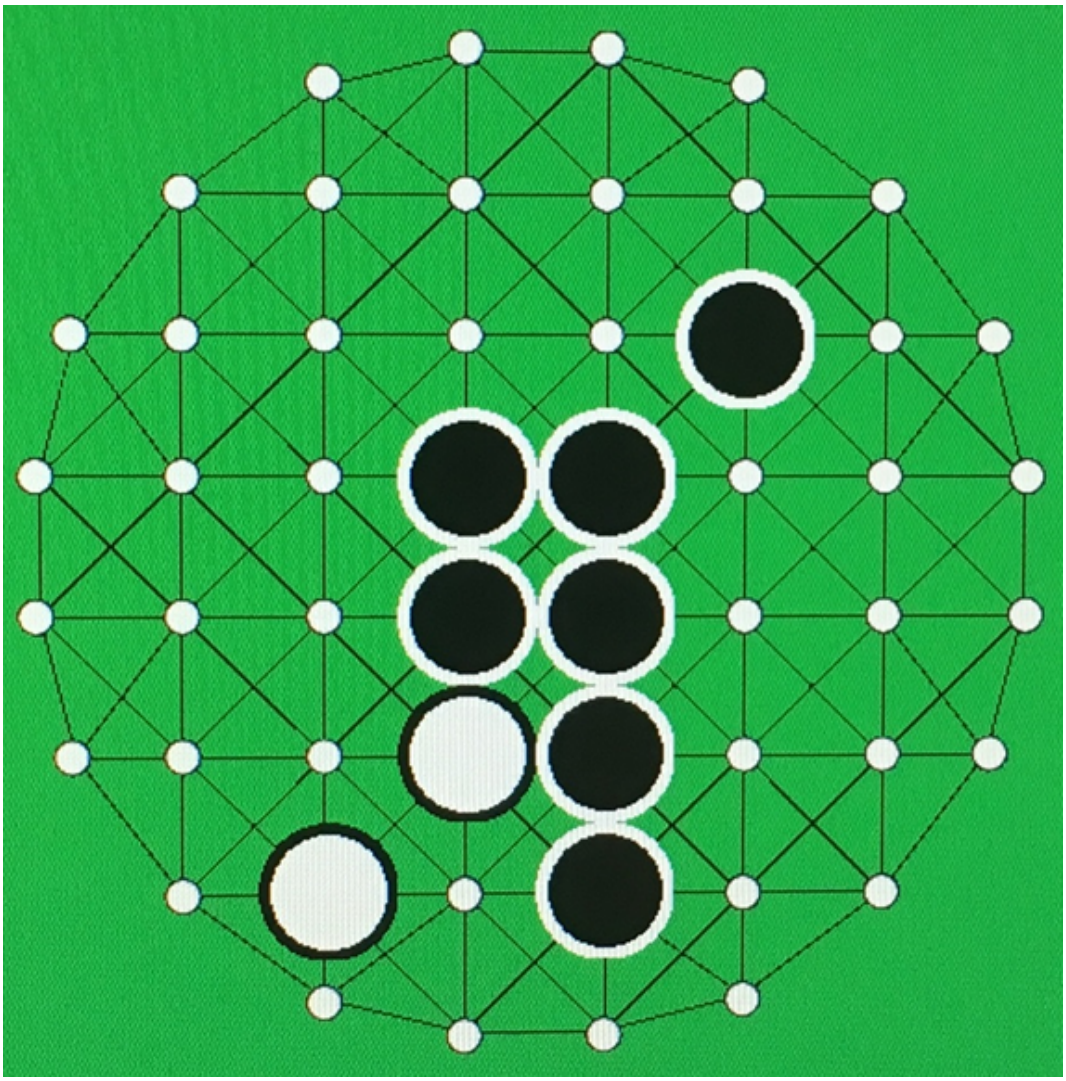

図 1.1: Nip の盤面

B　ソースからの実行形式のビルド方法

SDL2 および SDL2_image, SDL2_ttf のインストール方法.

以下のリンクからダウンロードし，展開

https://www.libsdl.org/download-2.0.php

https://www.libsdl.org/projects/SDL_image/

https://www.libsdl.org/projects/SDL_ttf/

```
cd SDL2-2.0.4
./configure
make
sudo make install
cd..
cd SDL2_image-2.0.1
./configure
make
sudo make install
cd..
```

```
cd SDL2_ttf-2.0.14

./configure

make

sudo make install
```

コンパイルは以下の通り.

```
g++ -Wall -O3 -o nip
 -I/usr/local/include -I/usr/local/include/SDL
 -L/usr/local/lib -Wl,-rpath,/usr/local/lib
 -lSDL2 -lSDL2_image -lSDL2_ttf nip.c
```

C　プログラムの内部仕様

以下に CPU 版の内部仕様を示す.

C.1　モジュールの一覧とそれぞれの説明

表 3.3 にモジュール一覧を示す.

C.2　データ構造の一覧とそれぞれの説明

表 3.4 にデータ構造一覧を示す.

C.3　SDL2 ライブラリ

表 3.5 に SDL2 ライブラリ一覧を示す.

D　UCT 探索アルゴリズムの改良

今回は高速化に限定したが, 実装段階において, UCT 探索のアルゴリズムを改良した結果, 勝率が上がった. その手法を以下に示す.

D.1　従来手法

UCT 探索では, 全候補手の UCB1 値のうち, その値が最も高いものにさらに多くのプレイアウトを割り当て, プレイアウト回数が閾値を越えた場合, その候補手を展開する. しかし, どの程度まで深く探索するかによって実行時間が異なってくる. ある程度の勝率を得るために探索木を十分深くしなければならず, 実行時間がかかりすぎるケースも想定されうる. また, Nip プログラムでは, プレイアウト回数が 32768 回となるとモンテカルロ法と同等の強さになることが分かった. これはシミュレーションから得られる確率的な勝率が期待値に近づいたためと考えられる. この結果から, Nip の場合では, UCT 探索を用いた AI とモンテカルロ法を用いた AI の性能が同等と言える.

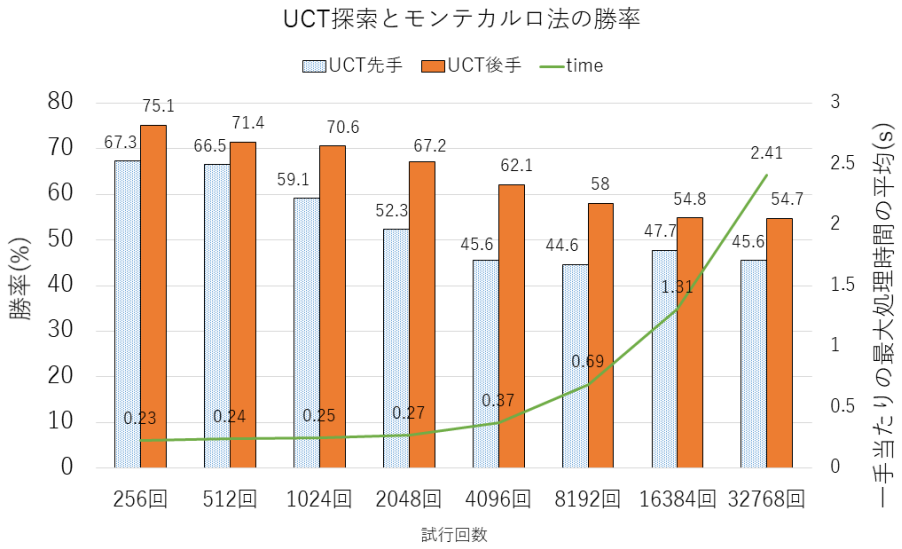

図 4.1: 実行時間と勝率の関係

D.2 提案手法

提案手法は, ある程度強い候補手全てについてさらに多くのプレイアウトを割り当てるというものである. 以下に疑似コードを示す. UCB1 値のボーダーを越えた全候補手について, 値の更新を行う. この提案手法実装により, 速度を保ったまま, 勝率について 10%程の向上がみられた.

```
int c = 0;//ループのカウント
int f = 0;//更新があるかどうか
while(c < UCB1_loop){
  f = 0;
  for(i=0;i<=key;i++){
    if(threshold < UCB1[i].val){
    //UCB1 値が threshold(ボーダー) を越えたものすべてについて UCB1 値の更新を行う
      f++;
      c++;
      N++;
      UCB1[i].Nj++;
      double x;
     //モンテカルロ法から UCB 1 値を求める
      x = ai_UCT_put(which, tip_num, lboard, UCB1[i].pos);
      UCB1[i].val = (UCB1[i].val*(UCB1[i].Nj-1) + x)/UCB1[i].Nj
                    + bias*sqrt(2*log((double)N)/UCB1[i].Nj);
    }
  }
  if(f == 0){c = UCB1_loop+1;}//更新が必要ないのであれば, ループを抜ける
}
```

E　UCT 探索の適用

以下に示すように, 今後 UCT 探索の活躍の幅が広がると考えられる.

コンテンツの取捨選択
例. クックパッド
　　バンディット問題を用いて, コンテキストごとに最適なコンテンツを表示
　　参考 URL : http://techlife.cookpad.com/entry/2014/10/29/102036

マリオのプレイ動画
最適なプレイを学習するのに
A* アルゴリズム　　-> 敵の隙間を最短で駆け抜けるような非人間的な動き
モンテカルロ木探索　-> 人間のように慎重に敵を倒しながら進んで行く
　　参考 URL : http://aial.shiroyagi.co.jp/2014/11/ibis2014/

表 3.3: モジュール一覧

型	関数名	説明
int	main	初期化およびメインループ
void	ev_esc	メインループから抜ける
void	player_turn_pp	player vs player の時の player ターン
void	AI_turn_p	player vs player の時の AI ターン
void	player_turn_cc	cpu vs cpu の時の player ターン
void	AI_turn_c	cpu vs cpu の時の AI ターン
void	conect	チップの連結
void	put	チップを置く
int	put_check	チップが置けるか調べる
void	player_put	player ターンの put 処理
void	AI_put	AI ターン put 処理
void	copy_board	ボードの状態をコピー
int	branch_check	チップが置けるか調べる
int	winer	勝者判定
int	tree_search	ランダムにチップを置いていく
int	tree_create	N 回 tree_search 実行
int	ai_montecarlos	チップを置ける頂点それぞれの勝ち数を返す
void	ai_montecarlo	最も勝率のよい頂点にチップを置く
void	setBoard	ボードの初期化
void	drow_board	描画
void	images_load	画像読み込み
void	images_check	読み込めたか確かめる
void	make_images_texture	画像 Texture の生成
void	images_release	画像 Texture の解放
void	strings_check	読み込めたか確かめる
void	strings_load	文字読み込み
void	make_strings_texture	文字 Texture の生成
void	strings_release	文字 Texture の解放

表 3.4: データ構造一覧

型	関数名	値	説明
enum	Mode	Select, Put, End	各ターンで行う処理の種類
enum	Turn	Player, AI	ターンの種類
enum	Board	Black, White, Gray, Nothing	ボードの頂点の状態
enum	Key	Up, Down, Stay	キーボードの状態
enum	Checkertype	Begining, Ending, Finishing	ランダムにチップを置いていく時に使用
#define	FONT_PATH	”font. ttf”	TTF で読み込む font ファイル
#define	PI	3. 141592	円周率
#define	SIDE_NEXT_NUM	10	チップの周囲のマス (8+円周 2)
const int	SCREEN_WIDTH	640	Window の横
const int	SCREEN_HEIGHT	640	Window の縦
const int	BOARD_VERTEX_NUM	52	ボードの頂点の数
struct board_t {			ボードの頂点の構造体
int	x		頂点の x 座標
int	y		頂点の y 座標
int	borw		頂点の色
int	my_num		頂点の番号
int	side[SIDE_NEXT_NUM]	0:up	自分の上の頂点番号
		1:up right	自分の右上の頂点番号
		2:right	自分の右の頂点番号
		3:down right	自分の右下の頂点番号
		4:down	自分の下の頂点番号
		5:down left	自分の左下の頂点番号
		6:left	自分の左の頂点番号
		7:up left	自分の左上の頂点番号
		8:Circle right	自分の円周右の頂点番号
		9:Circle left	自分の円周左の頂点番号
board_t *	next[SIDE_NEXT_NUM]	0:up	自分の上の頂点との連結
		1:up right	自分の右上の頂点との連結
		2:right	自分の右の頂点との連結
		3:down right	自分の右下の頂点との連結
		4:down	自分の下の頂点との連結
		5:down left	自分の左下の頂点との連結
		6:left	自分の左の頂点との連結
		7:up left	自分の左上の頂点との連結
		8:Circle right	自分の円周右の頂点との連結
		9:Circle left	自分の円周左の頂点との連結
}			

表 3.5: SDL2 ライブラリ一覧

関数名	説明
SDL_Init	SDL の初期化
TTF_Init	TTF の初期化
SDL_CreateWindow	Window 生成
SDL_CreateRenderer	Render の生成
SDL_GetError	SDL の Error を検出
IMG_GetError	IMG の Error を検出
TTF_OpenFont	Font ファイルを読み込む
SDL_QueryTexture	画像のサイズを取得
SDL_SetRenderDrawColor	Render の Color を設定
SDL_RenderClear	Render の内容を消去
SDL_RenderPresent	Render の内容を表示
SDL_Delay	一定時間待つ
IMG_Quit	IMG 終了
SDL_DestroyRenderer	SDL の Render を廃棄
SDL_DestroyWindow	SDL の Window を廃棄
SDL_Quit	SDL 終了
TTF_CloseFont	Font ファイルを閉じる
TTF_Quit	TTF 終了
SDL_PollEvent	SDL の Event を取得
SDL_RenderCopy	Render へ Copy
IMG_Load	画像を Load
SDL_CreateTextureFromSurface	Surface から Texture を生成
SDL_FreeSurface	Surface を解放
TTF_RenderUTF8_Blended	文字コード指定