卒業論文

題目

モンテカルロ法を用いたリバーシのGPUによる並列化

指導教員

大野和彦　講師

2015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

三輪隼秀(412857)

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．しかし, 強い AI を作るには膨大な計算が必要となる．そのような中，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 に注目が集まっている．それに伴い，NVIDIA 社により GPGPU 専用の SDK CUDA[1] が開発されている．そこで，本研究では最も並列化に適したモンテカルロ法を用いたリバーシプログラムを GPGPU を用いて高速化したのち, データ構造やメモリの使用方法の工夫を検討し, 適用することでさらに高速化を行った．その結果，CPU に比べて処理時間が 17.5 倍もの高速化を実現することができた.

Abstract

Today, AI of various board games develops, and something which has the ability of the professional level. But enormous calculation is needed to make a strong AI. Performance improvement is gazed at by GPGPU which uses remarkable GPU for general-purpose calculation compared with CPU. SoftWare Development Kit CUDA of GPGPU exclusice use has been developed by NVIDIA company. MonteCarlo method is most suitable for parallelization in this research,the Reversi program becomes to speed up by using a GPGPU. As a result, the processing time as compared with the CPU is able to realize high-speed even 17.5 times.

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, モンテカルロ法を採用したリバーシ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 CUDA，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. 乱数を用いて回数をこなし解を求めるため, ボードゲームにおいて評価関数を作成せずに済むのが利点である. しかし, 高い精度を得ようとすると計算回数が膨大になってしまうという弱点があるが, 同じシミュレーションを複数回行うような単純な繰り返し処理のため並列化に適している. 本研究では, 膨大な回数の試行をGPGPU によって並列化することで上記の問題点を解決し, 高速化を図ることが目的である.

2.2 GPGPU

2.2.1 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.

2.2.2 スレッドとブロック

スレッドは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が動作するのに対し, GPUでは数万〜数十万のスレッドを並列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．

2.2.3　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 を実装する上で重要となるものが, グローバルメモリ, シェアードメモリの 2 つのメモリである．グローバルメモリは最大で 12GB と容量が非常に大きいが, レイテンシが非常に大きい．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．

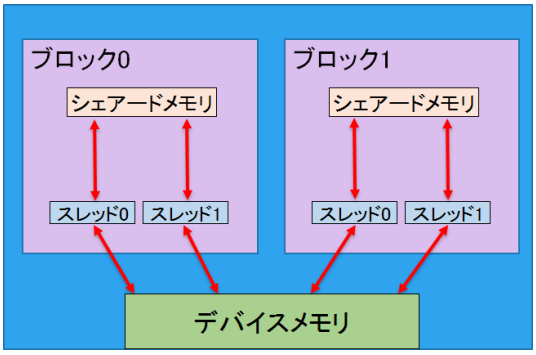

fig. 2.1: GPU の並列処理アーキテクチャ

2.2.4 CUDA

CUDA 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.

3　実装方法

3.1　概要

ボードゲームにおけるモンテカルロ法とは, ある盤面の候補手を探し, 全ての候補手でプレイアウト (終局まで打つ) を開始する. 規定回数分のプレイアウトを行い, その結果を用いて各候補手の評価値を求め最も評価値の高い手を選択する. 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規定回数分実行する. そのため, 規定回数が増えるほど処理時間も増える. そこで,GPGPU によるモンテカルロ法の高速化のために, 現在の盤面に対して横方向 (x), 縦方向 (y) にコマが置けるかどうかの検証を行い, コマが置ける場合はその場所に置いた状態から終局まで打ち進めることを試行回数分まで繰り返す処理を並列化する. 従って CUDA 化の実装としては, スレッドを 3 次元化し, x,y の 2 次元でリバーシの盤面である 8×8 のマスに対応させた. スレッド内では x,y を組み込み変数である threadIdx.x,threadIdx.y で取得した. また, 試行回数についてはスレッドの z 成分である threadIdx.z とブロック数で実現するようにした.

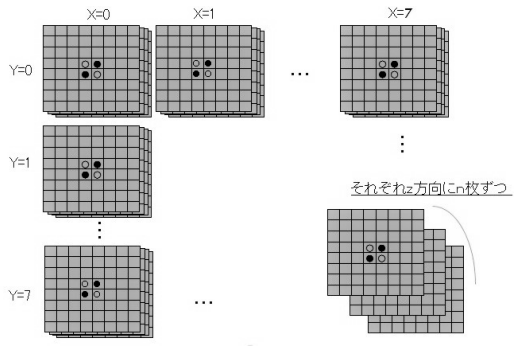

fig. 3.2: スレッド・ブロック構造

3.2 実装

3.2.1 逐次版の作成

本研究では,GPGPU を用いたモンテカルロ法の高速化が目的のため, 比較対象となる CPU のみでのモンテカルロ法を用いたリバーシプログラムの作成を行った. 逐次処理版のリバーシ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 版

プレイアウト関数において, ホスト側とデバイス側の双方で使用する関数をコピーし, デバイス用の関数を新しく作成した. デバイス用の変数については, ホスト側→デバイス側への入力として現在の盤面の状態・現在の手番, デバイス側→ホスト側への出力としてシミュレーションの結果（勝ち数）を格納する配列を用意し, デバイスメモリに割り当てた. 試行回数分のループを削除し, ブロックおよびスレッドをリバーシの盤面に対応するようにブロック数:m, スレッド数:$8 \times 8 \times n$ で起動するようにした. これにより, 評価対象のコマ位置, 縦 (y)・横 (x) には CUDA の組み込み変数である threadIdx.x,threadIdx.y を利用した. また, 試行回数はブロック数 m とスレッド数の z 成分 n の $m \times n$ となる. 多数のスレッドを起動し, デバイスメモリに対して複数スレッドで同時に加算などの書き込みを行う際に, 不整合が発生するためデバイス側→ホスト側への出力としてシミュレーションの結果 (勝ち数) を更新する際には,atomicAdd(),atomicExch() 関数[1]を使用した.

[1]あるスレッドがグローバルメモリやシェアードメモリ上のデータを読み込み, 修正し, 書き込むという一連の処理を行うとき, その処理中にそのメモリ領域に他のスレッドが書き込みを行わないようにする関数.Add=加算,Exch=置き換え

3.2.3　ビットボード化

2 次元の char 型の配列で盤面を表現していたものをビットパターンで表現するものへと変更した．これによって 2 次元の char 型配列に比べてサイズが減り, メモリのアクセス量を削減することで高速化が見込めるのに加えて, 後述する低容量のシェアードメモリにデータを割り当てできるようにした．ビットパターン化は以下のように行った．盤面を記憶する board 配列を 1 マスに対して,10:白,01:黒,00:なし, といった 2 ビットで構成し,8 マス分用意することで 16 ビットに格納した．それを行数分用意することで 8×8 の盤面を 16 バイトで表現した．実際には,char 型の 2 次元配列の場合は 64 バイトとなるため,4 分の 1 のサイズで表現できている．

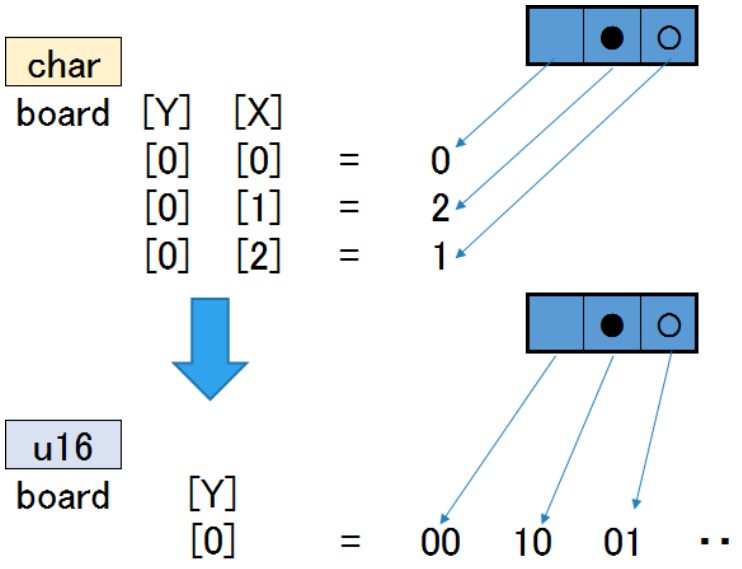

fig. 3.3: 盤面のビットボード化

3.2.4　シェアードメモリの利用

カーネル関数において頻繁に利用する盤面のデータ構造をビットパターン化したことによりシェアードメモリに配置し, より高速なアクセスを実現した. 上記の 16 バイトの盤面配列をスレッド最大数の 512×16 で確保し, シェアードメモリ内の競合を避けるためにスレッド ID を tId = threadidx.x + threadidx.y*8 + threadidx.z*64 で求めた. スレッド数分ずらすことでシェアードメモリに盤面を格納し, 使用した.

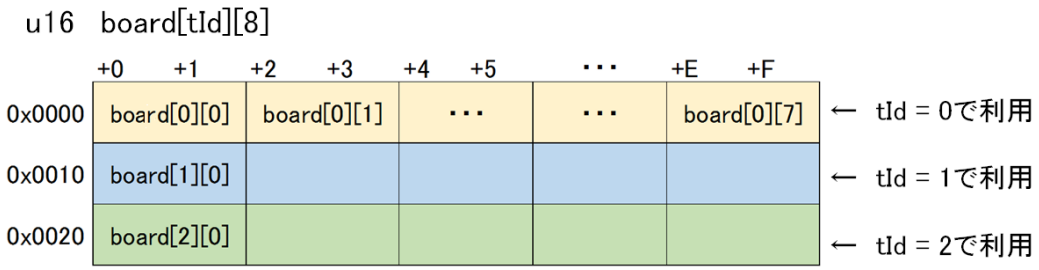

fig. 3.4: シェアードメモリ上のデータの割り当て

4 評価

まず, 単純な CUDA 化を行ったものと CPU 版との比較を行った. 試行回数・ブロック数・スレッド数のパラメータを変更し, 試行にかかる処理時間を計測した表を Table1 に示す. 次に単純な CUDA 化版とビットボード化やシェアードメモリ利用といった高速化を施した版の両方を比較した表を Table2 に示す. [2] 実行環境としてメモリ：16GB,CPU：corei7-4820K 3.70GHz,GPU:GeForce GTX TITAN を搭載した計算機を使用した. [3]

[2] パラメータはブロック数:4096, スレッド数:$8 \times 8 \times 8$.

[3] ここで表 4.1, 表 4.2 の処理時間は全て, ゲームの 2 手目にかかる思考時間であり, 配置可能マス数は 3 マスである.

表 4.1: CPU 版との CUDA 版の処理時間

試行回数	ブロック数	スレッド数	処理時間 (秒)	対 CPU 比
1,024	1,024	$8 \times 8 \times 1$	0.0414	5.99
	256	$8 \times 8 \times 4$	0.0251	9.87
	128	$8 \times 8 \times 8$	0.0250	9.91
	CPU 版		0.2478	–
16,384	16,384	$8 \times 8 \times 1$	0.387	10.6
	4,096	$8 \times 8 \times 4$	0.251	16.4
	2,048	$8 \times 8 \times 8$	0.240	17.1
	CPU 版		4.11	–
32,768	32,768	$8 \times 8 \times 1$	0.723	11.0
	8,192	$8 \times 8 \times 4$	0.521	15.3
	4,096	$8 \times 8 \times 8$	0.511	15.6
	CPU 版		7.95	–

表 4.2: 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
32,768	CPU 版	7.95	–
	CUDA 化	0.511	15.6
	ビットボード化	0.475	16.7
	シェアードメモリ利用	0.455	17.5

4.1 考察

表 4.1 を見ると, 試行回数を増やすほど CUDA 化の恩恵が大きいことが分かった. そして, ブロックの数を少なくし 1 ブロックあたりのスレッド数を増や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 においてもいえることであるので妥当である. 表 4.2 からは, 今回ビットボード化・シェアードメモリを利用したデータ構造がリバーシの 8×8 の盤面だったためか, 工夫展においては大きな速度改善には繋がらなかったが, 最終的には CPU 版のものと比較して 17.5 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, モンテカルロ法を用いたリバーシプログラムを 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 17 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, リバーシの盤面をビットパターン化しデータ構造を工夫することでメモリアクセスを効率化し, さらにシェアードメモリを用いることでより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] NVIDIA Developer Zone,https://developer.nvidia.com/category/zone/cuda-zone

[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