Vol.2011-GI-26 No.10
2011/7/1

GPGPUによるモンテカルロ碁のシミュレーションの並列処理

岩川 夏季[†1] 成見 哲[†1] 村松 正和[†1]

GPU(Graphics Processing Unit) を汎用的に使うことを GPGPU(General-purpose computing on Graphics Processing Units) という．本研究では，モンテカルロ碁におけるシミュレーションを GPGPU により並列化することを目指した．結果として，GPGPU による革新的な高速化には繋がらなかったが，多くの興味深い並列アルゴリズムを構築することができた．

GPGPU for parallel processing of simulation of Monte-Carlo Go

Natsuki Iwakawa,[†1] Tetsu Narumi[†1] and Masakazu Muramatsu[†1]

Graphics Processing Unit is called GPU; General-purpose computing on Graphics Processing Units is called GPGPU. Our research purpose is to accelerate simulation of Monte-Carlo Go by parallel computing of GPGPU. We have developed several algorithms to be used by Monte-Carlo computer Go program using GPU.

1. はじめに

1.1 研究背景

コンピュータ囲碁は従来，囲碁知識などを囲碁プログラムに組み込む知識ベースが主流だった．しかし，2006 年のモンテカルロ木探索の成功により，近年はモンテカルロ碁に注目が寄せられている[1)]．特に，UCT(Upper Confidence bounds for Tree) を用いたモンテカルロ木探索（以下，UCT）は多くの囲碁プログラムに広まり，コンピュータ囲碁界の歴史を塗り変えた[2)3)4)]．現在もモンテカルロ碁は UCT の改良や知識ベースとの融合，機械学習などの多方面からのアプローチにより，飛躍し続けている．

1.2 研究目的

本研究は，並列処理を得意とする GPU（§3 参照）を用いることでシミュレーション回数の増加を目指す．モンテカルロ碁では，シミュレーション回数の増加が棋力の向上に繋がることが一般的に知られている．

GPU は，近年性能が急成長している機器であり，並列化に向いている分野で処理の高速化に成功し，現在，多分野で処理の高速化が期待されている．本研究では，NVIDIA 社[5)] の GPU を使用する．GPU を使用するための開発環境には，NVIDIA 社により提供されている C++を拡張した統合開発環境 CUDA[6)] を使用する．

本研究は，GPU 内で複数の独立なシミュレーションを並列・並行に実行し，単位時間あたりのシミュレーション回数を計測し，CPU のシミュレーション時間と比較した．

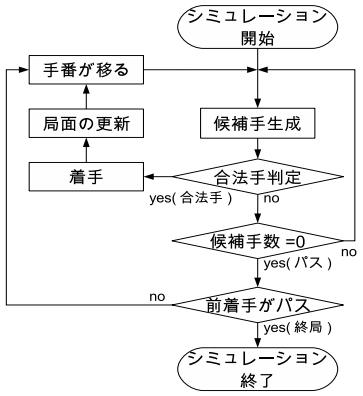

図 1 シミュレーションのフローチャート
Fig. 1 Flow of simulation.

†1 電気通信大学大学院 情報理工学研究科
Graduate School of Informatics and Engineering, the University of Electro-Communications

© 2011 Information Processing Society of Japan

2. モンテカルロ碁

モンテカルロ碁は現在の局面における候補手をシミュレーションにより評価するアルゴリズムである．モンテカルロ碁では繰り返し行うシミュレーションから得られる勝率が，評価関数[*1]の役割をしている．

従来の CPU を用いた場合の典型的なシミュレーションの流れを**図 1** に示す．自殺手は禁止手になるので，候補手生成における候補手が合法手かどうか調べる必要がある．合法手判定はシミュレーション内で非常に多く呼ばれるので，プログラムのボトルネックになりやすい．この合法手判定を高速に行うことは，高速なプログラムを作成するために重要である．

合法手判定を改善する方法として，連[*2]構造を取り入れる方法が一般に採用されている．候補手に隣接する 4 近傍点の連の情報を知るだけで合法手かどうか判定が可能である．

3. GPU

3.1 GPU の概要

GPU とは，画像処理を担当する機器である．GPU は年々進化し，その構造も変化している．本章以下の章で説明する GPU は，本研究で使用する GPU GeForce GTX 580 を対象とする．GPU は，Thread Scheduler，Global Memory，複数の SM(Streaming Multi-processor) から構成されている．Thread Scheduler はスレッド発行ユニット，Global Memory は GPU にとってのホストコンピュータの主記憶にあたる部分である．SM は「32CUDA Core+2Scheduler+1LM」で構成され，詳細図を**図 2** に示す．CUDA Core は積和算演算ユニット，SFU(Special Function Unit) は複雑な演算を行うユニット，Scheduler は各 CUDA Core への命令発行ユニットである．LM(64K Configurable Cache/Shared Memory) は SM 内の 32 基の CUDA Core が共有するメモリであり，Global Memory に比べて高速であるが，64KB しかない．SM の構成は GPU により異なり，その個数も，GPU によって異なる．GeForce GTX 580 は SM を 16 基搭載している．

3.2 GPU の特徴

3.2.1 SIMD

GPU は一つの命令を複数データに対し同時に行う SIMD(Single Instruction Multiple Data) であるので，複数データを一度に処理することを高速に行える．そのため，GPU において高速な処理を行うためには，処理をマルチスレッドの並列化に拡張することが必要である．GPU が並列処理を得意としている理由は，図 2 の様に多くの演算器を持ち，その演算器が休まず動き続けるように命令発行しているためである．GPU 内の各 SM は独立であるので，SM の数だけ並列処理ができる．SM 内の処理は，一度に処理するデータの数が多いときや，各スレッドが異なる命令を発行するときは並行処理になる．

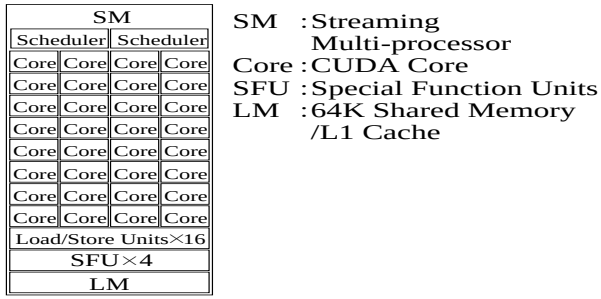

図 2 SM の構造
Fig. 2 Structure of SM.

3.2.2 GPU の不得意処理

本研究で使用するアプリケーションで，最も多く存在する GPU の不得意処理が分岐処理である．分岐処理を苦手な理由は，分岐処理実行後，各スレッドが他スレッドと異なる命令を発行することで，CUDA Core の実行効率が落ちるためである．SM 内の全 CUDA Core は同じ命令を発行しなければならないので，異なる命令を発行するときは別々に時間を費やすことになる．このような GPU の不得意処理を出来るだけ避ける必要がある．

3.2.3 Warp

CUDA には，Warp という SM 内の 32 スレッドを一塊として処理する演算単位がある．GeForce GTX 580 の SM には CUDA Core が 32 基ある．GeForce GTX 580 では，32 基の演算機を 16 基の演算機が 2 セットあると考え，一度に 2Warp を 2cycle で処理する．

3.3 GPGPU

GPGPU とは，GPU を汎用的に使用するための技術である．本研究は，GPGPU を使用することで性能向上を目指す．

GPGPU は，GPU を汎用的に使用できるようにするが，GPU を使用するアプ

⋆1 行動（囲碁では着手）や局面を評価し数値に変換する関数
⋆2 連：盤上の縦横の線で繋がった同色の石を一塊としたもの

© 2011 Information Processing Society of Japan

リケーションも，GPU に合ったものでなければならない．アプリケーションが性能向上するためには，§3.2 で述べた長所を活かし，短所を回避することが大切である．

4. 提案アルゴリズム

4.1 モンテカルロ碁のシミュレーションにおける GPU の利点・問題点

モンテカルロ碁は，並列化に向いている[7]．それは，モンテカルロ碁のシミュレーションは，独立であり，並列に他のシミュレーションが実行されていても，その影響を一切受けないからである．

しかし，図 1 より，シミュレーション処理には局面に依存した分岐処理が多くある．図 1 の中だけでも分岐処理が 2 つある．また,「合法手判定」と「局面の更新」は，着手しようとしている点の隣点の状態によって次処理が変わるので分岐処理がある．

モンテカルロ碁のシミュレーションは並列性には優れているが分岐処理が多いので，異なる局面を SIMD で処理すると CUDA Core の実行効率が良くない．

4.2 GPU に適した碁盤モデル

本章では，並列処理を得意とする GPU 用のアルゴリズムについて提案する．

図 3 4 路盤におけるスレッド割り当て
Fig. 3 Allocation of threads in 4x4.

図 3 のように碁盤の各点を 1 スレッドが担当することで，各処理を並列に行えるようにする．シミュレーションに使用する配列を**表 1** に示す．表 1 の配列は各スレッドからアクセスでき，担当点の情報は，自分のスレッド番号を配列の添字として得られる．石を打つ毎に全スレッドが担当点の情報を更新する．連のダメとは，その連がダメを持つか否かの判定をする際に使用する配列であり，配列の添字が連番号に対応している．ダメの配列は常に正しい値を持つように逐次更新するのではなく，着手の際，除去する石があるかないかを調べるときに使用する．ダメの値もだが，連の更新は逐次的になる部分が多く SIMD で処理するのが困難である．本アルゴリズムでは，石の除去判定の際に必要であるダメ以外の連情報は持たない．

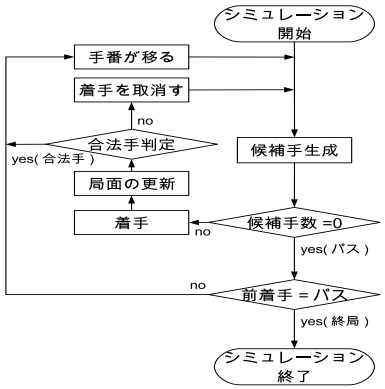

図 4 GPU でのシミュレーションのフローチャート
Fig. 4 Flow of simulation of purposed algorithm.

表 1 シミュレーションに使用する配列
Table 1 Variables for simulation.

各点：	点の色（空点，黒，白，盤外）
	連番号
	着手確率のレーティング
連：	ダメ

GPU でのシミュレーションの流れを**図 4** に示す．図 1 で問題としていた「合法手判定」は「局面の更新」後に行う．更新後に非合法手（自殺手）であった場合は着手を取消し，候補手生成に戻る．

本アルゴリズムは，1 点 1 スレッドにすることで SIMD に合った処理ができる利点がある．しかし,「合法手判定」が「局面の更新」後になるので，局面の更新が無駄処理になる不利点がある．

© 2011 Information Processing Society of Japan

Vol.2011-GI-26 No.10
2011/7/1

Initialize: $n \leftarrow 1$ とする．自分のスレッド番号を tid とする．
Loop:
  Step1: 「$n \geq$ 碁盤の全点数」ならば Loop を終了する．
  Step2: $n \leftarrow n \times 2$ とする．
  Step3: $(tid)\mathrm{mod}(n) \geq (n/2)$ を満たすならば，担当点のレーティングに $(tid - (tid)mod(n)) + (n/2 - 1))$ の点のレーティングを加える．
  Step4: 全スレッドに対し，同期を取る．Step1 に戻る．

図 5 レーティングの足し合わせ
Fig. 5 Algorithm to add ratings.

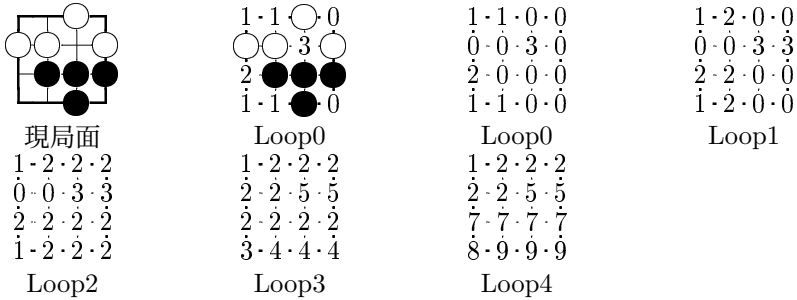

図 6 4 路盤における足し合わせの例
Fig. 6 Example of adding ratings in 4x4.

4.2.1 候補手生成

候補手生成は，担当点の着手確率のレーティングを求める部分と，各点のレーティングを足し合わせる部分で構成される．

担当点のレーティングは，近傍点の点の色を取得し，その近傍点の位置と点の色の関係からレーティングを算出する．但し，石が置かれている点と眼の形になっている点，劫における非合法手の点のレーティングは 0 にする．

全スレッドが担当点のレーティングを得た後，同期を取り，自分よりも若いスレッド番号の担当点のレーティングを，自分の担当点のレーティングに足し合わせる．レーティングを足し合わせる順序を**図 5** に示す．図 5 のアルゴリズムは各スレッドが其々実行する．

レーティングの足し合わせの計算量は，図 5 の Loop 回数によって決まる．Step1, 2 より，Loop 回数は log_2(碁盤の全点数) であるので，レーティングの足し合わせの計算量は O(log_2(碁盤の全点数)) である．候補手生成の主な計算はこの足し合わせでかかる．

図 6 に 4 路盤における「レーティングの足し合わせ」の例を示す．Loop0 における盤上の数字は各点のレーティングである．Loop1 は，Loop0 の 2 列目に 1 列目の同じ行のレーティングを，4 列目に 3 列目の同じ行のレーティングを加えている．Loop2 は，Loop1 の 3 列目と 4 列目に 2 列目の同じ行のレーティングを加えている．Loop3 は，Loop2 の 2 行目に 1 行 4 列目のレーティングを，4 行目に 3 行 4 列目のレーティングを加えている．Loop4 は，Loop3 の 3 行目と 4 行目に 2 行 4 列目のレーティングを加えている．候補手を求めるには，乱数 x を生成し，x を 4 行 4 列のレーティングで割る．この余りが，(自分の担当点より一つ若い点のレーティング) $\leq$ (x の余り) $<$ (自分の担当点のレーティング) の点が候補手になる．但し，スレッド番号 0 より一つ若い点のレーティングは 0 としている．

4.2.2 局面の更新

局面の更新は，碁盤に石を置く部分と，石を置くことにより生じる碁盤のデータの更新部分により構成される．候補手点に石を置く際，その点が合法手か否か調べていないので，碁盤データ更新部分で非合法手であるか否か調べる．非合法手であった際は，碁盤データを石を置く前に戻す．また，その非合法手を候補手から除外後，再度候補手生成を行う．局面の更新の処理を**図 7** に示す．図 7 のアルゴリズムは各スレッドが其々実行する．但し，Step 毎に全スレッドは同期をとっている．

図 8 と**図 9** に 4 路盤における「局面の更新」の例を示す．各局面の Step 数は「局面の更新」のアルゴリズムにそっている．また，局面が変化しない局面は載せていない．

図 8 は Step0 において，△ に着手し，Step1 において連番号を更新している．Step2 において，連番号 5，8，9，10 のダメを 0 にセットする．Step3 において，連番号 8，9，10 のダメを 1 にする．Step4 において，連番号 5 のダメが 0 なので，連番号 5 に所属する石を除去する．

図 9 は Step0 において，△ に着手し，Step1 において連番号を更新している．Step2 において，連番号 1，3，5，6 のダメが 0 にセットする．Step3 において，連番号 1，3，5 のダメが 1 になる．Step5 において，連番号 6 のダメが 0 なので，着手 △ を取消し，連番号を Step0 の状態に戻す．

5. 実　　験

GPU と CPU の秒間シミュレーション回数を比較する．実験環境を**表 2** に示す．実験で計測した時間は，石が一つも置かれていない初期局面からのシミュレーションにかかった時間である．GeForce GTX 580 の SM は 16 基あるので，GPU では

© 2011 Information Processing Society of Japan

Step1: 担当点の連番号のコピーを取る（局面を戻すときに備える）.
Step2: 担当点が候補手ならば，石を置き，連番号を現時点の着手数にする.
Step3: 今置いた石の上下左右の連番号を其々共有メモリの変数 up, down, left, right に入れる.
Step4: 担当点の連番号が up, down, left, right のどれかと同じかつ今置いた石の色と同色ならば，連番号を現時点の着手数にする.
Step5: 担当点の連番号のダメを 0 にする.
Step6: 担当点に隣接する空点が一つ以上あるとき，担当点の連番号のダメを 1 にする.
Step7: 担当点の点の色が今置いた石と異なる色で，担当点の連番号のダメが 0 ならば，担当点を空点にしかつ今置いた石の連番号のダメを 1 にする.
Step8: 担当点の連番号のダメが 0 ならば，担当点の連番号を Step1 でコピーした連番号に戻す．また，担当点が今石を置いた点ならば空点に戻す.

図 7 局面の更新
Fig. 7 Algorithm of updating board

図 8 4 路盤における連番号の更新と石の除去の例
Fig. 8 Example of updating strings and removing stones in 4x4.

図 9 4 路盤における合法手判定の例
Fig. 9 Example of judging illegal move in 4x4.

複数局面並行シミュレーションを 16 個並列に実行した．CPU では 1CPU コアにより 1 局面をシミュレーションした.

表 2 実験環境
Table 2 Environment of experiments.

CPU：	Core i5-2500 @ 3.30GHz
GPU：	NVIDIA GeForce GTX 580
GPU の CUDA Core 数：	512 基
GPU のプロセッサークロック：	1.54GHz

表 3 9 路盤における秒間シミュレーション回数（simulations/second）
Table 3 simulations/second in 9x9.

	Random	3 × 3	M2
CPU	53475	44642	28735
GPU1	108936	102400	75294
GPU2	169696	193103	164705

表 4 19 路盤における秒間シミュレーション回数（simulations/second）
Table 4 simulations/second in 19x19.

	Random	3 × 3	M2
CPU	10515	9033	5934
GPU1	9320	9795	5614
GPU2	5333	5925	4848

表 3 は 9 路盤における，**表 4** は 19 路盤における，GPU と CPU の秒間シミュレーション回数である．シミュレーション内の候補手生成の違いから Random，3 × 3，M2 の 3 種類の秒間シミュレーション回数を計測した．Random は，シミュレーション内の候補手を全ての点に対してランダムに決める．3 × 3 は各点の 8 近傍点の石の配置から，M2 は各点からマンハッタン距離 2 までの 12 近傍点の石の配置から，その点に着手する確率を決め，着手確率をもとに候補手を決める．GPU1 は，CPU と同じソースコードを GPU 用に一部変更し，GPU にて計測したシミュレーション回数である．GPU2 は，§4 にて述べたアルゴリズムをもとに実装したプログラムのシミュレーション回数である．GPU では，シミュレーション回数が最大

© 2011 Information Processing Society of Japan

になるように，並行シミュレーションする局面数を求めて計測した．

表 3 において，GPU1 の Random と 3 × 3 では，64 局面並行シミュレーションを 16 個並列に実行しているので，計 1024 局面を並列・並行シミュレーションしている．M2 では，16 局面並行シミュレーションを 16 個並列に実行しているので，計 256 局面を並列・並行シミュレーションしている．GPU2 は，Random，3 × 3，M2，3 種類全てで，7 局面並行シミュレーションを 16 個並列に実行しているので，計 112 局面を並列・並行シミュレーションしている．

表 4 において，GPU1 の Random と 3 × 3 では，12 局面並行シミュレーションを 16 個並列に実行しているので，計 192 局面を並列・並行シミュレーションしている．M2 では，4 局面並行シミュレーションを 16 個並列に実行しているので，計 64 局面を並列・並行シミュレーションしている．GPU2 は，Random，3 × 3，M2，3 種類全てで，並行シミュレーションする局面はなく，16 局面を並列シミュレーションしている．

6. 考　　察

6.1 9 路 盤

表 3 より，GPU1，GPU2 の単位時間あたりのシミュレーション回数は CPU よりも多い．特に，§4 のアルゴリズムをもとに実装した GPU2 は CPU よりも，Random において約 3.2 倍，3 × 3 において約 4.3 倍，M2 において約 5.7 倍，シミュレーション回数が多い．CPU，GPU1 のシミュレーション回数が M2 において大きく減る原因は，シミュレーション内の着手の度に近傍点の着手確率を更新するためである．しかし，GPU2 では着手の度に全点の着手確率を一から求めるので，Random，3 × 3，M2 においてシミュレーション回数に大きな差は生じない．GPU2 の 3 × 3 でシミュレーション回数が最も多いのは，シミュレーションの質が良く，短い手数で終局したためと考えられる．また，M2 のシミュレーション回数が Random とあまり変わらないのは，シミュレーションの質が良くないためか，終局に要する手数が Random の手数とあまり変わらなかったためと考えられる．

6.2 19 路 盤

表 4 より，GPU1 のシミュレーション回数は CPU とあまり変わらず，GPU2 のシミュレーション回数は CPU より少ない．

19 路盤では CPU の方が GPU1 よりシミュレーション回数が多い理由は，並列・並行シミュレーションする局面数が 9 路盤に比べて減っているためと考えられる．局面数の分だけ GPU はメモリやレジスタが必要になる．19 路盤では，9 路盤の約 4.5 倍のメモリが必要になるので，最適な並列・並行シミュレーションする局面数が減ったのだと考えられる．

9 路盤に比べて，GPU2 のシミュレーション回数が大きく減った理由は，処理が非常に増えたために効率良く並行シミュレーションすることができなかったためと考えられる．GPU1 では 1 局面に対して 1 スレッドが処理するのでメモリの増加はあっても使用レジスタの増加はあまりない．しかし，GPU2 では各点を担当するスレッドが其々レジスタを使用しているので，9 路盤に比べてレジスタを使用する量が約 4.5 倍になる．従って，この使用レジスタ量が GPU の資源を圧迫し，効率良く実行できる並行シミュレーション数が制限されたものと考えられる．

7. 結　　論

GPU にてモンテカルロ碁のシミュレーションを並列・並行シミュレーションすることに成功した．§4 で提案したアルゴリズムが 9 路盤では，有効であるが，19 路盤では有効でなかった．しかし，提案アルゴリズムは着手確率の更新が Random，3 × 3，M2 全てでほぼ同じ計算量のためシミュレーション回数に大きな差はなかった．シミュレーション回数があまり変化しないことは，着手確率を更新する近傍点をより広くしても期待できる．

今後の課題として，シミュレーションを GPU で行う UCT を実装したプログラムの開発がある．その際，並列・並行シミュレーションする局面が 1 局面 1 シミュレーションで結果を返すと通信遅延が多くなると考えられる．また，GPU で実行する毎に一度だけ行う初期化があるが，この初期化も増えることになる．例えば，1 局面 1 シミュレーションで結果を返すのに比べると，1 局面 10 シミュレーションは通信遅延と初期化のための実行時間が約 1/10 になると予想できる．そのため，効率の良い GPU でのシミュレーション回数を求める必要がある．

参 考 文 献

1) B. Brugmann, Monte Carlo Go, 1993
2) R. Coulom, Computing Elo Ratings of Move Patterns in the Game of Go, In Computer Game Workshop, Amsterdam, The Netherlands, 2007
3) R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, In P.Ciancarini and H.J.van den Herik, editors, Proceedings of the 5th International Conference on Computer and Games, Turin, Italy, 2006
4) S. Gelly, Y. Wang, R. Munos and O. Teytaud, Modification of UCT with Patterns in Monte-Carlo Go, Technical Report RR-6062, INRIA, 2006
5) NVIDIA 社, http://www.nvidia.co.jp/page/home.html
6) CUDA ZONE, http://www.nvidia.co.jp/object/cuda_home_new_jp.html
7) 加藤英樹, 竹内郁雄, 並列 MC/UCT アルゴリズムの実装, 第 12 回ゲーム・プログラミング ワークショップ 2007, 情報処理学会, pp.23-29, 2007

© 2011 Information Processing Society of Japan