CUDA 将棋：GPGPU による並列ゲーム木探索

○ 木村 健[†] 小谷 善行[††]

本稿では GPGPU（グラフィックスプロセッサの汎的計算への応用）を用いた将棋のゲーム木探索手法を提案する．チェスや将棋などのゼロ和完全情報ゲームにおける並列ゲーム木探索の研究は決して新しい研究ではない．しかしながらさまざまな技術が年を経るごとに開発されているにつけ，並列探索の技術も変化を必要としている．GPU 上における並列計算環境である CUDA と，CUDA ライブラリの一部である CUDA 版 C++ STL とも言える Thrust を用いた，並列ゲーム木探索手法を提案する．提案手法は 1993 年に Transputer 上に実装された YBWC をベースにしており，高いスピードアップが期待できる．ただし CUDA 固有の高スループット実現の難しさもあり，提案手法の実際の実装についての知見が必要である．

CUDA Shogi: Parallel Game Tree Search by Using GPGPU

Takeshi KIMURA[†] and Yoshiyuki KOTANI[††]

In this paper, we introduce a Shogi(Japanese Chess)'s game tree search method that uses GPGPU(General Purpose GPU). This is not new research field that parallel game tree search for zero-sum perfect information games (such as Chess and Shogi). The technology of parallel game tree search need change to fit new technology such as GPGPU. In this paper, we introduce the way of searching game tree in parallel which uses CUDA(Parallel environment for GPGPU) and Thrust(C++ STL like CUDA based library). The proposal method is based on YBWC which implemented 1024 nodes Transputers in 1993, so high speed-up is guaranteed. But there are difficulties of implementing high throughput program in CUDA, so individual implementation and their performance findings are needed.

1. はじめに

近年 GPGPU(General Purpose Graphics Processing Units) の高まりとともに，GPGPU の様々な応用分野において GPGPU 対応ソフトが CPU 専用ソフトに対し数十倍から数百倍の性能をあげており，関心が集まっている．特に n 体問題のように並列に解くことが可能な問題において顕著な性能をあげている．

本来 GPGPU はシェーダー技術においてフラグメントシェーダーを使って，並列計算ができないかという Question から始まって，その後 NVIDIA 社の CUDA 技術のような汎用計算向けの並列計算環境が整うこととなった．

† 東京農工大学 工学府 電子情報工学専攻
Department of Electronic and Information Engineering Graduate School of Engineering, Tokyo University of Agliculture and Technology
†† 東京農工大学 工学研究院 先端情報科学部門
Division of Advanced Information Technology & Computer Science, Institute of Engineering, Tokyo University of Agriculture and Technology

GPU の計算性能（主に FLOPS）は CPU の成長がやや鈍化しているのに対し，その構造的特徴から CPU よりもより線形に成長するのではないかといわれている．このため今後数年において GPU の性能が飛躍的に進化すると，その GPU に向けて書かれたソフトは大きな修正なく，成長の恩恵を受けることができると予測する．

本稿では GPGPU を用いた並列ゲーム木探索を行う将棋プログラムについて述べる．具体的には 1993 年に Transputer 上に実装された YBWC という並列ゲーム木探索手法について，GPGPU にフィットするような形の将棋プログラムの実装手法を提案する．

本稿では 3 節において GPGPU 及び CUDA の概要について述べる．そして 4 節において Thrust ライブラリの概要について述べ，5 節において YBWC について述べ，6 節において提案手法について述べる．

2. 関連研究

3) は alpha-beta 法についての詳しい数理的解析を始めて行った．4) は PVS と呼ばれる並列探索手法に

ついて述べている．この手法は長男を最初に探索したら，次は二男以降を並列に探索する手法である．本格的な massively parallel search の研究は 1)2) によって塗り替えられた．この論文以前と以降で並列探索の研究は研究内容が大きく異なる．10) の研究は minmax search を GPGPU で実現するものであるが，YBWC は用いていない点において本研究と異なる．14)15) は囲碁における GPGPU の研究である．これらの論文は alpha-beta search ではなくモンテカルロ search と呼ばれる手法を採用している．

3. GPGPU 及び CUDA 概要

CUDA は，NVIDIA 社 GPU 用の並列計算アーキテクチャである．CUDA は NVIDIA 社 GPU のハード構成に対し，抽象化したソフトウェアアーキテクチャを提供している．本稿ではその両方について簡単に述べる．

3.1 NVIDIA 社 GPU のハード構成

まず GPU(グリッドと呼ばれる) 上には SM(Streaming Multi-Processor) という，ALU の集合体が数十基存在する．最新の GeForce GTX 580 には 16 基の SM が存在する．さらに個々の SM には 32 基の CUDA コアと呼ばれる ALU が存在する．このため GeForce GTX 580 上には全部で 512 基の CUDA コアが存在することとなる．

各 SM はローカルメモリー（シェアードメモリか，L2 キャッシュとして利用可能）を持つ．GPU 全体としてのグローバルメモリー（どの SM からもアクセスできる）は，GeForce GTX 580 の場合 1536 MB GDDR5 と広大であり，メモリインターフェース幅は 384-bit，メモリバンド幅は 192.4(GB/sec) と高速である．ただし，シェアードメモリが数クロックでアクセスできるのに対し，グローバルメモリは数百クロックを Read/Write に必要とする．メモリアクセスは個々の CUDA コアについて一つづつ行うのではなく，warp と呼ばれる単位 (= 32) で一気に Read/Write を行う．32 個の warp のうち，Read に 16，Write に 16 が割り当てられる．

3.2 CUDA のソフトウェアアーキテクチャ

NVIDIA 社 GPU のハード構成はそれぞれのグラフィックスカードによりまちまちである．これをソフトウェアレイヤーで隠ぺいする．具体的にはスレッドとブロックという二つの実行単位を用意して，ソフト側からはこれらを用いて並列計算を行う．

具体的にはスレッドは，POSIX スレッドのようなスレッドではなく，非常に light weight なスレッドである．スレッドの集まりをブロックという．スレッドとブロックにはいろいろなソフトウェア制約が有り複雑であるが，次の二点を抑えておけばよい．

- スレッドは CUDA コアを跨げない
- ブロックは SM を跨げない

つまりスレッドとは CUDA コアへの資源割り当ての仮想化されたものであり，ブロックとは SM への資源割り当ての仮想化されたものである．

通常 1 ブロックあたり数百のスレッドを実行する．これはグローバルメモリのレイテンシを隠ぺいするためである．また SM あたりいくつのブロックを実行できるかという制約も存在する．通常は数ブロックを同時にひとつの SM で実行しスループットを向上させる．

GPU デバイス上での算法の実行は通常 CUDA 言語と呼ばれる C/C++の拡張言語で行う☆．並列計算はカーネルという特殊な関数（`__global__`という修飾子を付ける）を同時に実行することで行う．いわゆる一種の SIMD である．このカーネルの実行時に 1 ブロックあたりのスレッド数と，実行するブロック数を指定することが可能である．スレッドには特殊な変数があり，自分が何番目のスレッドか，何番目のブロックかを知ることができる．この特殊変数を用いてすべてのスレッドが異なる振舞いをすることが可能であるが，通常分岐はその THEN パートと ELSE パートが順に二つ実行されるため非効率的である．ただし warp の境界で振舞いを変える場合は例外である．

GPGPU は普通の CPU の計算のように，メモリにプログラム本体をロードして実行という簡単な流れではなく，やや複雑なメモリ転送を伴う流れで行う．

(1) CPU→GPU メモリ転送（入力データの転送）
(2) GPU 上でカーネルを実行
(3) GPU→CPU メモリ転送（結果の転送）

4. Thrust ライブラリ

Thrust ライブラリ[13] は，C++ STL(C++ Standard Template Library) によく似たインターフェイスを持つ，並列計算ライブラリである．CUDA 言語によるカーネルの記述は様々な制約や同時に実行するスレッド数，ブロック数などを考慮して行わなければならないが，Thrust はそのあたりを隠ぺいしており，よりアルゴリズムの設計に集中できるように工夫がなされている．

たとえば 32M（メガ）の int を整列させたい場合，その並列整列プログラムを実装するのは決して楽なこ

☆ ただし Python や Fortran での実行も可能である．

とではないが，Thrust を使うと，たった一行で非常に効率のよい parallel radix sort による整列が簡単に行える（図 1 : Thrust QuickStartGuide より引用）.

```
#include <thrust/sort.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(A, A + N);
// A is now {1, 2, 4, 5, 7, 8}
```

図 1 整列の例

Thrust には`std::vector` に似た`host_vector` と`device_vector` がある．これらの vector には任意の型の変数・構造体を入れることが可能である．`host_vector` と`device_vector` で相互に代入やコピーを行うと，GPU↔CPU 転送が自動で行われる．Thrust の各種アルゴリズムはこの vector を操作することで実現される．

Thrust には多くのアルゴリズム実装があるが，この中で本稿の提案手法と密接に関係あるのが`thrust::reduce` 及び`thrust::transform` 並びに`thrust::transform_reduce` である．transform は vector の各要素に対して functor と呼ばれる一種の単項演算子を適用する．このとき functor はユーザーが自在に定義することが可能であり，複雑な処理を行うことが可能である．また reduce は vector の個々の要素を二項演算子で結合し，結果を返す．簡単な例は int の足し算による合計値の算出である．

transform と reduce を一緒に適用できる`transform_reduce` を使うと，さらに複雑な計算を行うことが可能である．本稿では`transform_reduce` を使った手案手法について後に述べる．

5. YBWC

YBWC[1] は R. Feldmann によって 1993 年に発明された並列ゲーム木探索アルゴリズムである．詳しくは[2] に譲るが，大まかにその探索手法を述べると次のようになる．

5.1 ノードのタイプと同期ポイント

ゲーム木のすべてのノードを次の 3 タイプに分類する[3]．ゲーム木は well ordered であると仮定する．

- ルートノードは Type 1
- Type 1 ノードの長男ノードは Type 1，二男以降は Type 2
- Type 2 ノードの長男ノードは Type 3，二男以降は Type 2
- Type 3 ノードの子供はすべて Type 2

その上で，次のように同期をとる．

- Type 1 ノード v の子供ノードの並列探索は，v の長男が完全に探索された時だけ許される
- Type 2 ノード v の子供ノードの並列探索は，v の有望な手が完全に探索された時だけ許される
- Type 3 ノード v の子供ノードの並列探索は，常に許される

YBWC の実装はゲーム木探索をノードごとに部分問題に分割して各プロセッサに割り当てる．そして空いているプロセッサが部分問題の割り当てられているプロセッサに対して部分問題を取りに行く．このため部分問題がプロセッサ空間全体に拡散しやすく，並列度が高くなるようにアルゴリズムが設計されている．

6. 提 案 手 法

本稿の提案手法を述べる．子局面の生成を並列に行うためにカーネルを用いて可能手生成する以外は，局面情報を vector のノードに収め，`transform_ reduce` を用いて新しいノードを計算する．

最初は初期盤面から探索を始める．まず vector に初期盤面に対応する，盤面情報などを入れた構造体を格納し，子局面を生成する．初期盤面は Type 1 ノードなので，長男（Type 1）と二男以降（Type 2）に分けて子局面を生成する．このうち Type 1 ノードはさらに子局面を展開し，上記と同様にする．このようにして，葉ノードに達する一つ上の局面において，並列に評価関数を実行する．こうして一つ上の局面の評価値が決まる．その時点において α, β を更新し，さらに一つ上の局面において二男以降を同時に並列に評価関数を求める．Type 2 ノードにおいては有望な手だけを先に展開するようにする．Type 3 ノードがある場合，子供は無条件に並列に展開・評価してよい．

このようにして段々と並列で実行できるノードを増やしていく．子局面生成と評価関数の実行は一緒の vector に入っていると分岐により性能悪化を招くので，別の vector に入れておき，必要に応じて vector 間を移動するようにする．

6.1 enum

vector の要素として格納されている構造体には自分が計算過程のどの状態にあるのかを表す enum が格納されている．それらの特徴について順に示す．

表 1 提案手法で使われる enum
Table 1 enums that use in the proposal method

enum 名
BEGIN
WAITFIRST
EVAL
LEAF
PASSAB
WAITVALUE

6.1.1 BEGIN

ルートノードにも適用される enum である．この enum を持つノードはノード展開（子局面生成）の対象となる．ノード展開の結果として生成された子局面は，通常長男と二男以降，あるいは Type 2 の子局面については有望な手の集合とそれ以外の子供という風に二つに分割される．

6.1.2 WAITFIRST

二男以降のノードや，Type 2 の子供については有望ではない手の集合に対し付与される enum であり，長男ノードや有望な手の α, β 値を待つ（同期する）．

6.1.3 EVAL

葉局面の親ノードがこの EVAL を付与されると，その子局面をすべて展開し，評価値を並列に計算する準備に入る．

6.1.4 LEAF

EVAL のノードから展開された葉局面で，自分の評価値を並列に計算したのちに親ノードに α, β 値を伝搬する．

6.1.5 PASSAB

長男あるいは有望な手の集合が完全に探索されたときに生成されるノードの enum で，α, β 値を二男あるいは有望ではない手に伝搬する．

6.1.6 WAITVALUE

自分の子ノードを展開したあと，自ノードの評価値を決定するために子ノードの結果を待っている状態のノードに付与される enum である．自ノードの値が決定したら，それをさらに自分の親ノードに伝える．

6.2 計 算 過 程

Thrust の vector がどのように変化するかをルートノードから順に示す．角括弧内の要素は順に，[ノード番号（リストもありうる），評価値，alpha，beta，enum] である

```
[1,-inf,-inf,+inf,BEGIN]

↓

[1,-inf,-inf,+inf,WAITVALUE]
[2,-inf,-inf,+inf,BEGIN]
[[3,4],-inf,-inf,+inf,WAITFIRST]

↓

[1,-inf,-inf,+inf,WAITVALUE]
[2,-inf,-inf,+inf,WAITVALUE]
[[3,4],-inf,-inf,+inf,WAITFIRST]
[5,-inf,-inf,+inf,EVAL]
[[6,7],-inf,-inf,+inf,WAITFIRST]

↓

[1,-inf,-inf,+inf,WAITVALUE]
[2,-inf,-inf,+inf,WAITVALUE]
[[3,4],-inf,-inf,+inf,WAITFIRST]
[5,-inf,-inf,+inf,EVAL]
[[6,7],-inf,-inf,+inf,WAITFIRST]
[8,100,-inf,+inf,LEAF]
[9,50,-inf,+inf,LEAF]
[10,25,-inf,+inf,LEAF]

↓

[1,-inf,-inf,+inf,WAITVALUE]
[2,-inf,-inf,+inf,WAITVALUE]
[[3,4],-inf,-inf,+inf,WAITFIRST]
[5,-inf,100,+inf,PASSAB]
[[6,7],-inf,-inf,+inf,WAITFIRST]

↓

[1,-inf,-inf,+inf,WAITVALUE]
[2,-inf,-inf,+inf,WAITVALUE]
[[3,4],-inf,-inf,+inf,WAITFIRST]
[6,-inf,100,+inf,EVAL]
[7,-inf,100,+inf,EVAL]

↓

.....
```

ゲーム木の構造を次に示す．ゲーム木は well ordered であると仮定し，branch factor は簡略化のため 3 とした．good Type 2 ははじめの二つとして，ノードの番号は展開順である．最左がルート局面で，右に行くほどノードが深くなる．

```
1+2+5+ 8
 | |   +9
 | |   +10
 | |+6 +11
 | |   +12
 | |   +13
 | |+7 +14
 |     +15
 |     +16
 +3 +17+23
 |  |  +24
 |  |  +25
 |  +18+26
 |  |  +27
 |  |  +28
 |  +19+35
 |     +36
 |     +37
 +4 +20+29
    |  +30
    |  +31
    +21+22
    |  +33
    |  +34
    +22+38
       +39
       +40
```

7. 今後の研究

本研究にはまだ改良の余地がある．実装がないのももちろん大きな問題であるが，アルゴリズムをより具体的にかつ詳細に定義し，どのようにノードの展開を制御するかを明確に定義する必要がある．

また本稿では alpha-beta 法の β カットについての具体的な手法を述べなかった．今後の研究では探索ノード数を劇的に減らせる β カットについてもどのようにアルゴリズムに組み込むか具体的な手法を編み出していきたい．

アルゴリズムをより明確にした上で，まずは小さなゲームでの実装を行い，その後将棋の実装を行いたい．具体的には本研究を実際の実装に落とし込みたい．

8. おわりに

GPGPU を用いた並列ゲーム木探索について述べた．提案手法は YBWC をベースにしたもので，高い並列度が期待できる．早く実装を行い，結果を見える形で publication などにつなげたい．

参考文献

1) R. Feldmann, P. Mysliwietz, and B. Monien: *Game Tree Search on a Massively Parallel System*, in Advances in Computer Chess VII, H. Van Den Herik, I. Herschberg, and J. Uiterwijk (Eds.), University of Limburg, Masstricht, the Netherlands, pp. 203-218, (1994).
2) R.Feldmann: *Game Tree Search on Massively Parallel Systems.* PhD thesis, University of Paderborn, (1993).
3) D.E. Knuth, R.W. Moore: *An Analysis of Alpha - Beta Pruning.*, Artificial Intelligence, Vol. 6, pp 293–326, (1975).
4) T. A. Marsland, M. Campbell, *"Parallel Search of Strongly Ordered Game Tree"*, in ACM Computing Surveys, Vol 14, No. 4, pp. 533–551, (1982).
5) Curt Powley, Richard E. Korf, and Chris Ferguson, *"Parallelization of Tree-Recursive Algorithms on a SIMD Machine"*, AAAI Technical Report SS-93-04, (1993).
6) Christopher F. Joerg and Bradley C. Kuszmaul *"Massively Parallel Chess"*, Proceedings of the Third DIMACS Parallel Implementation Challenge, Rutgers University, New Jersey, (1994).
7) Richard E. Korf, David Maxwell Chickering, *"Best-first minimax search"*, Artificial Intelligence 84, pp. 299–337, (1996).
8) Lubomir Lackovic, *"Parallel Game Tree Search Using GPU"*, 7th Student Research Conference in Informatics and Information Technologies IIT.SRC, (2011).
9) Yoshimasa Tsuruoka, Daisaku Yokoyama, Yakashi Chikayama, *"Game-Tree Search Algorithm Based on Realization Probability"*, ICGA Journal, 25(3):145-152, (2002).
10) Kamil Rocki and Reiji Suda, *"Parallel Minimax Tree Searching on GPU"*, R. Wyrzykowski et al. (Eds.): PPAM 2009, Part I, LNCS 6067, pp. 449–456, (2010).
11) Avi Bleiweiss, *"Playing Zero-Sum Games on the GPU"*, presentation of GPU technology conference, (2010).
12) Nadathur Satish, Mark Harris, Michael Garland, *"Designing Efficient Sorting Algorithms for Manycore GPUs"*, To Appear in Proc. 23rd IEEE International Parallel and Distributed Processing Symposium, May (2009).
13) Nathan Bell and Jared Hoberock, *"Thrust: A Productivity-Oriented Library for CUDA"*, to appear in GPU Computing Gems: Jade Edi-

tion, Morgan Kaufmann Publishers, (2011).
14) 田野文彦, “**グラフィックスエンジンを用いたゲーム探索の高速化**”, 東京大学大学院工学系研究科電気系工学専攻融合情報学コース　修士論文, (2010).
15) 岩川夏季, 成見 哲, 村松 正和, “*GPGPU*によるモンテカルロ碁のシミューレーションの並列処理”, 情報処理学会研究報告, Vol.2011-GI-26 No.10, (2011).