Major Project – II

Report on

Accelerated Game Tree Search Using Hybrid Multithreaded CPU and GPU Processing

Submitted to

Delhi Technological University

In partial fulfillment of the requirement for the award of the degree of

Master of Technology

In

Software Engineering

By

Aditya

2K14/SWE/02

Under the guidance of

Mr. Manoj Kumar

Asst. Professor

Department of Computer Science and Engineering

Delhi Technological University

Delhi Technological University

Main Bawana Road, Delhi

CERTIFICATE

This is to certify that the work contained in this dissertation entitle "**Accelerated Game Tree Search Using Hybrid Multithreaded CPU and GPU Processing**" submitted in the partial fulfilment, for the award of degree of M. Tech in Software Engineering, Department of Computer Science & Engineering at **Delhi Technological University** by **Aditya**, Roll No. **2K14/SWE/02**, is carried out by him under my supervision. The matter embodied in this project work has not been submitted earlier for the award of any degree or diploma in any university/institution to the best of my knowledge and belief.

Date: ………………

Mr. Manoj Kumar
(Thesis Guide)
Assistant Professor
Dept. of Computer Science Engineering
Delhi Technological University

ACKNOWLEDGMENT

I take this opportunity to express my gratitude and regards to my guide **Mr. Manoj Kumar** for her exemplary guidance, monitoring and constant encouragement throughout the course of this project. I am extremely grateful to her for her valuable guidance and suggestions.

I am obliged to the Head of Department **Prof. (Dr.) O.P. Verma** and faculty members of the Department of Computer Science & Engineering at Delhi Technological University, Delhi for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment.

I also take this opportunity to express a deep sense of gratitude to my friends for their support and motivation which helped me in completing this task through its various stages.

Lastly, I thank my parents for their constant encouragement without which this assignment would not have been possible.

Aditya
Roll No. 2K14/SWE/02

Table of contents

List of Figures 6
Abstract 7
Chapter 1 8
Introduction 8
Chapter 2 9
Literature Survey 9
2.1 Game Tree 9
2.1 Choosing a game 11
Chess, Rules and Pieces 12
Chapter 3 14
IMPLEMENTATION 14
3.1 Board Representation 14
3.1.1 Some Bitboard Constants Used 14
3.2 Move Generation 15
3.3 Evaluation of a position on the board 16
3.4 Search 17
3.4.1 Alpha Beta Algorithm 17
3.4.2 Enhancements in Alpha Beta Algorithm 19
3.5 Quiescence Search 25
3.6 Parallel Search 26
3.6.1 Threads 26
3.6.2 Young Brothers Wait Concept 26
3.6.3 Delay 27
3.6.4 Distributed Game-Tree Search 27
3.7 GPGPU Concepts and CUDA Programming Language 28
THE CUDA PARALLEL COMPUTING PLATFORM 29
3.8 APPROACH 30
Monte Carlo Tree Search 30
MCTS iteration steps 30
3.9 GPU IMPLEMENTATION 32
Hybrid CPU-GPU processing 33
Chapter 4 38
Conclusion 38
Chapter 5 39
Future Work 39

References 40
Appendix A 41
Snapshots of Code and System 41

List of Figures

Figure 1: Minimax Tree

Figure 2: NegaMax Tree

Figure 3: Chessboard starting position

Figure 4: Move pattern of pieces

Figure 5: Alpha Beta search tree

Figure 6: Principle variation search in alpha beta search tree

Figure 7: Ideal and practical speedup between n processors

Figure 8: Difference between standard C code and CUDA code

Figure 9: MCTS fundamental steps

Figure 10: Different types of parallelism

Figure 11: Block parallel scheme on hardware

Figure 12: Hybrid CPU and GPU processing

Figure 13: Speedup graph

Figure 14: Win Ratio graph comparing different parallel schemes

Figure 15: Performance comparison between GPU and multiple CPUs

Figure 16: Comparison between Hybrid and Non Hybrid approach

Abstract

Accelerating the speed of the search in a game tree of game of chess, which is an irregular large tree, is the main objective. By employing parallel processing power of modern CPUs with a stable algorithm is the main idea behind the work. Apart from CPUs, the modern era is shifting towards GPU processing for massive parallelism and more computational power is also employed to be in use of searching a game tree.

Chapter 1

Introduction

Since high speed parallel processing has evolved over the last few years, it's a high time to employ the techniques in artificial search algorithms, to assist in various applications, like game playing which is basically the application in this thesis, rest of the applications include, machine learning, path finding problems, etc.

The motivation behind this work is to combine the processing of multiple CPU cores including the GPU processing for faster (accelerated) game tree search for finding the goal. Game chosen is the game of chess because of its unrealistic complexity for modern computers and machines. Game of chess is chosen because the game tree will prove to be a better measure of performance, as well as a robust proof because of its branching factor, almost equal to 36.

We will go into the details of the game tree and most importantly how the searching techniques are employed in it to find the best move.

There can be two possibilities

1. A brute force search, which searches every legal move with a min-max algorithm. But ofcourse it has a serious downside that it will be searching almost 10^9 positions for just going to a depth of 3 (6 plies).
2. A selective search along with a quiescence search to finish off.

We would employ a selective search strategy called alpha-beta search to select only good moves and discard (prune) the bad moves, so as to shorten the tree, and proceed quickly to larger depths. Enhancements such as iterative deepening and null move pruning have been employed to increase the strength of the chess engine more.

Despite these techniques, we employ a full use of GPU processing, to increase the performance of the search algorithm more. We will go in detail of GPU processing further.

Use of multiprocessing has been prioritized more than anything else in the search. Instead of using 1 CPU for the search task, we will be using upto 64 CPUs to do the similar task using parallel SMP search algorithm.

We will describe the technologies used in section 1. In section 2, approach has been explained, section 3 and 4 cover various results and tests.

Chapter 2

Literature Survey

2.1 Game Tree

Game tree in games is a directed graph, in which vertices are the game positions, root node represents the current position of the game. The edges of the graph represent the legal moves from the position represented by the vertex. According to the rules, we can evaluate a vertex as a win, lose, draw, or a specific score on basis of some evaluation function.

For some games the tree size is unrealistically huge. For example, size of game tree of a game called checkers is 10^20, and chess is around 10^120. The total number of nodes in game tree is roughly W^D, where W stands for the number of possible moves on average also known as the branching factor, and D is the depth of the tree. One way to minimize the complexity is to use evaluation function at a particular node to determine its probability of being a good move or not, but that's what heuristic search employs. We can use the evaluation function on the leaf nodes. Here we have used iterative deepening, game playing programs depend on game tree search to find the best move for the current position, assuming the best play of the opponent. In a two-player game, two players select a legal move alternately, and both of them try to maximize their advantage.

Because of this reason, finding the best move for a player must assume the opponent also plays the best move. In other words, if the leaves are evaluated on the viewpoint of player A, player A will always play moves that maximize the value of the current position, while the opponent B plays moves that minimize the value of the following position. This gives us the MiniMax algorithm.

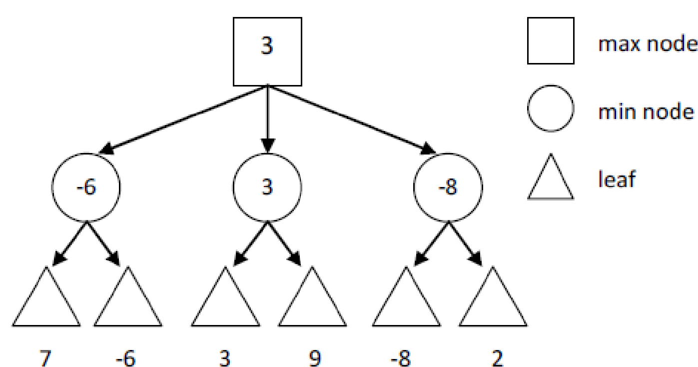


Figure 1: MiniMax Search Tree

Minimax Algorithm

```
1.    function MAXValue(N)
2.    begin
3.    if N is leaf then
4.         return the value of this leaf
5.    else
6.        let v = -∞
7.        for every successor N_i of N do
8.             let v=  max{v,  MINValue(N_i)}
9.    return v
10.   end MAXValue
```

```
1.    function MINValue(N)
2.    begin
3.    if N is a leaf then
4.        return the value of this leaf
5.    else
6.        let v = +∞
7.        for every successor N_i of N do
8.            let v=  min{v,  MAXValue(N_i)}
9.    return v
10.   end MINValue
}
```

For the sake of simplicity, we will use the variant of MinMax algorithm called NegaMax.

Negamax relies on the property that max(a, b) = -min(-a, -b), it relies on the zero sum property of the game. The trick is to maximize the scores by negating the returned values from the children instead of taking the minimum.

So our algorithm will become similar to the following.

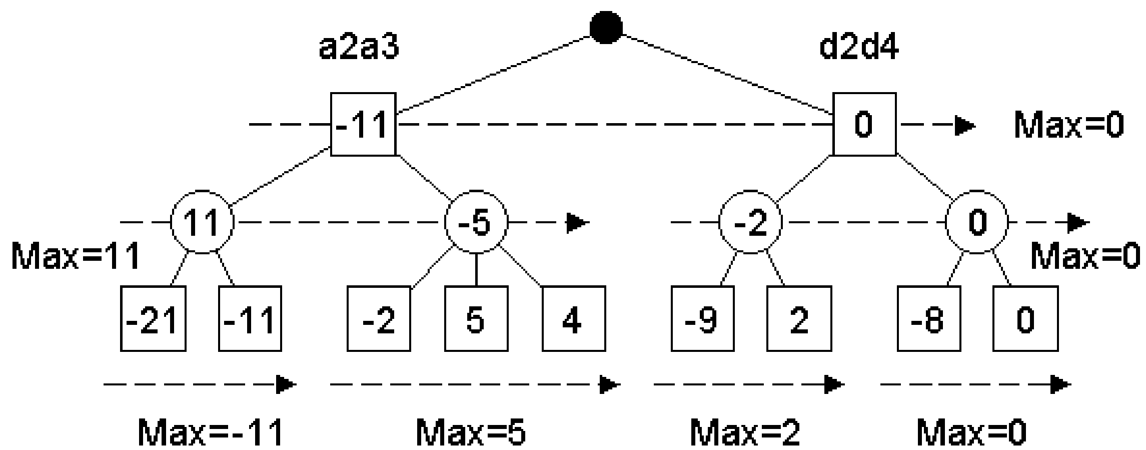


Figure 2: Negamax Search Tree

NegaMax Algorithm:

```
01 function negamax(node, depth, color)
02     if depth = 0 or node is a terminal node
03         return color * the heuristic value of node

04     bestValue := -∞
05     foreach child of node
06         v := -negamax(child, depth - 1, -color)
07         bestValue := max( bestValue, v )
08     return bestValue
```

```
Initial call for Player A's root node
rootNegamaxValue := negamax( rootNode, depth, 1)
rootMinimaxValue := rootNegamaxValue
```

```
Initial call for Player B's root node
rootNegamaxValue := negamax( rootNode, depth, -1)
rootMinimaxValue := -rootNegamaxValue
```

This will in turn later will become NegaScout with the addition of alpha beta pruning. This was just an example of algorithms to be used in the game tree.

Now let's choose which game which should select to employ these algorithms in.

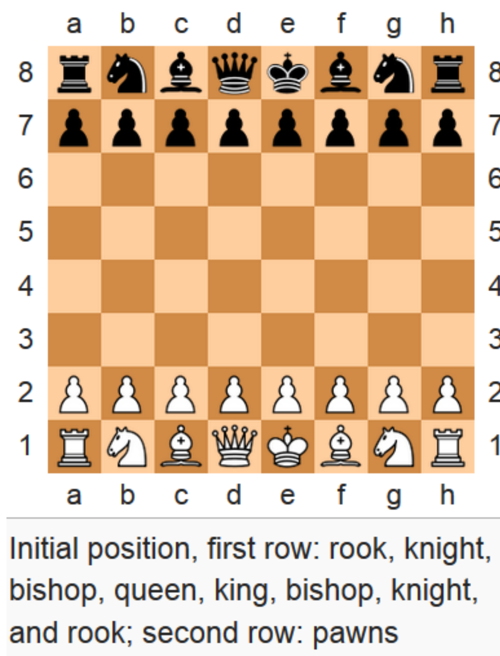
2.1 Choosing a game

For purpose of explanation and innovation, game of chess has been chosen, as it has a large game tree size, and calculations of performance can be taken robustly.

Chess game tree complexity is about 10^120 (Exponential), for modern computers chess remains unsolvable, so basically the motivation is solving the unsolvable, but still with the use of parallel search and deploying an efficient algorithm for GPU processing, the software strength will increase tremendously.

Chess, Rules and Pieces

Chess has 6 type of pieces. 2 sides, white and black. Starting board position of chess is as below



Initial position, first row: rook, knight, bishop, queen, king, bishop, knight, and rook; second row: pawns

Figure 3: Starting position of a chess game

Rules of the game.

1. Each player moves alternately.
2. Pieces are moved according to their type
3. Only one piece per square, pieces can be captured
4. Goal is to checkmate the opponent’s king (king is not left with any unattacked square)
5. If there are no moves left, and opponent is not in check, it’s a condition of a draw. (Stalemate)

Moves according to the pieces type. Black dots indicate the piece can only move on these squares.

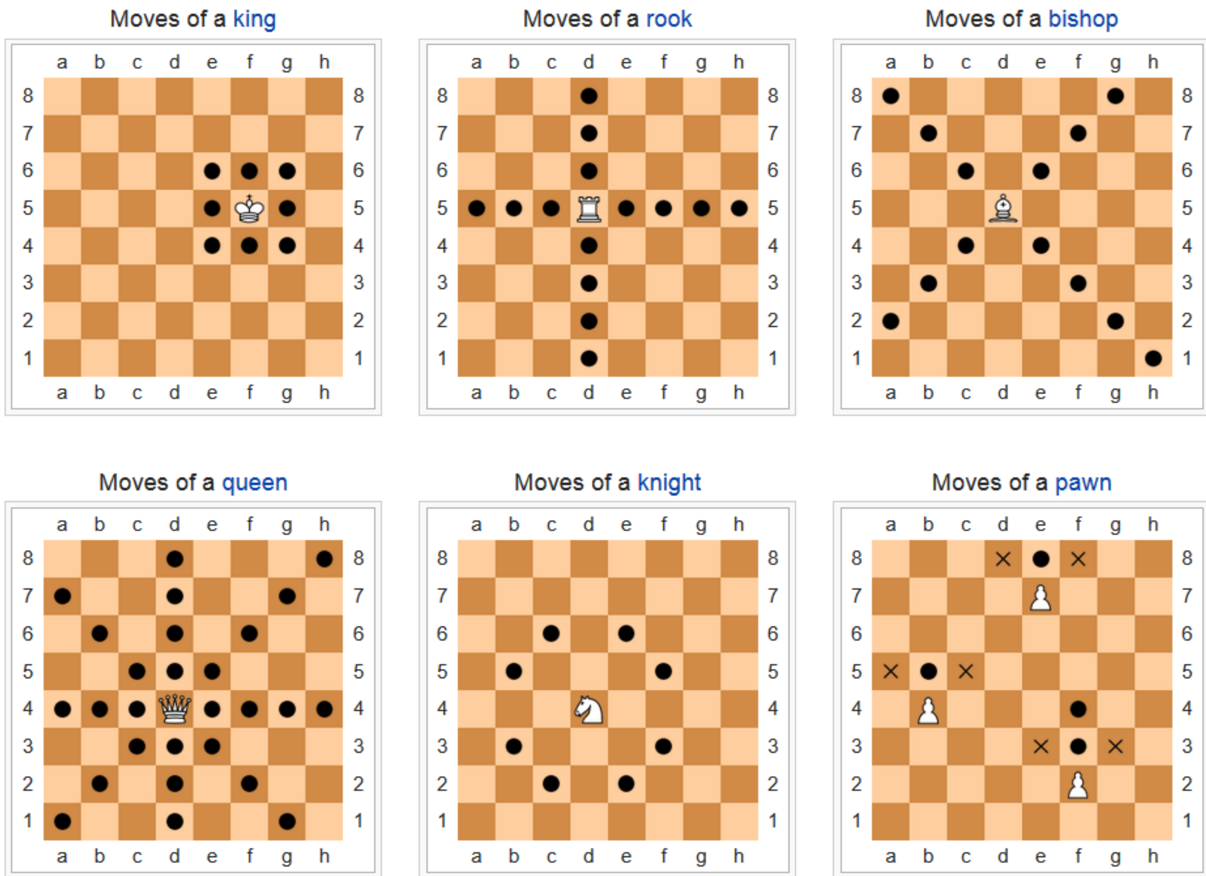


Figure 4: Moves of different pieces of chess

Chapter 3

IMPLEMENTATION

3.1 Board Representation

Board has been represented by bitboards also called bitsets or bitmaps, are among other things used to represent the board inside a chess program in a **piece centric** manner. Bitboards are finite sets of up to 64 elements - all the squares of a chessboard, one bit per square.

In chess, we have bitboards for every piece type.

So atleast we need six 64 bit unsigned long integers to represent 64bit number.

3.1.1 Some Bitboard Constants Used

```
a-file             0x0101010101010101
h-file             0x8080808080808080
8th rank           0xFF00000000000000
a1-h8 diagonal     0x8040201008040201
h1-a8 ant diagonal 0x0102040810204080
light squares      0x55AA55AA55AA55AA
dark squares       0xAA55AA55AA55AA55
```

3.2 Move Generation

With a board representation, one big consideration is the generation of moves. This is essential to the game playing aspect of a chess program, and it must be completely correct. Writing a good move generator is often the first basic step of creating a chess program.

3.3 Evaluation of a position on the board

Evaluation is used to heuristically determine the value of the position on the board.

We evaluate a position based on the basis of Material, mobility, pawn structure, king safety and attack threats.

Material is calculated on the basis of fixed values of pieces, pawn has a value of 1, knight and bishop have value of 3, rooks are 5 and queen is 9. King has infinite value.

Mobility is defined as the ability to maneuver the pieces on the board. It is calculated by number of undefended squares on the board on which our piece can safely move. Its calculated for every piece and then summed up.

King safety is calculated by number of adjacent zone attacks, direct attacks on king, ability of opponent to give the check and how strong the shelter around the king is.

Threat evaluation requires the knowledge of which type of piece is attacking which type of other piece.

Eg, if a knight is attacking 2 rooks at the same time, threat is very high on the opponent.

3.4 Search

As stated by Claude Shannon, there are two types of search

1. Brute Force – Search for all possibilities
2. Selective search – Search for more promising moves first

Search tree, as already stated is a subset in search space, a directed acyclic graph, with alternating white and black moves as the level increases in the tree.

Brute force technique means exploring all possible nodes in a tree. So it's a clearly impractical method of solving anything

Selective search is more efficient in solving problems, in which more promising move is evaluated in a depth first manner, before back tracking occurs.

We will use an enhanced alpha beta algorithm, an enhancement to negamax algorithm.

3.4.1 Alpha Beta Algorithm

When searching game trees in a two player game like chess, benefits can be made by pruning the branches in the tree which lead to a poor position, and does not affect the score at the root. This pruning method called Alpha Beta Algorithm. It applies to zero sum games with perfect information such as Chess and Checkers. It has 2 bounds called upper and lower bound to decide which branch to cut. This algorithm leads to heavy amount of pruning and making it easier to search the tree and doubling of search depth given the same time, over the normal MIniMax search algorithm.

Recently, Monte Carlo Tree Search (MCTS), which is a type of simulation-based best-first search algorithm, has been extended to allow for Alpha-Beta style pruning.

How it works?

For example, it is White's turn to move, and we are searching to a depth of 2 (that is, we consider all the White's moves, and all of Black's moves followed to each of those moves.) First we select one of White's possible moves - let's call this Move #1. We expand this move and every possible move to this move by black. After this, we determine whether the result of making Move #1 results in an even position or not. Then, we go further and consider another of White's possible moves (Move #2.) When we evaluate the first counter-move by black, we find that playing this results in black winning a Rook. In this situation, we can safely ignore all of Black's other responses to Move #2 because we already know that Move #1 is better. We don't care about how much worse Move #2 is. Maybe another move wins a Queen, but it doesn't matter because we know that we can achieve *at least* an even game by playing Move #1. The

full analysis of Move #1 gave us a lower bound. We know that we can get at least something, so anything that is worse so anything that can be pruned can be ignored. The situation becomes too difficult to evaluate when we go to a depth of 3 or greater, because now both players will make choices which will affect the shape of the game tree searched. So now two bounds are maintained, lower bound and upper bound (alpha and beta), we maintain a lower bound because if a move is worse we don't discard it. But we also have to maintain an upper bound because if a move at depth 3 or more leads to a sequence that is too good, the other player of course will try to avoid, because there was a better move higher up level on the game tree that he could play to avoid the situation. One player's lower bound is the opponent's upper bound.

Savings

The savings in this algorithm are noticeable and major. If a normal minimax search tree has **x** nodes, an alpha beta tree in an efficient program can have a node count close to the square-root of **x**. Nodes that can be cut actually depends on how well ordered the game tree is. If the best possible move is searched first, we then can eliminate most of the useless nodes. Of course, we cannot predict the best move in the first place, or we wouldn't have to search in the first place. Conversely, if we always searched worse moves before the better moves, we wouldn't be able to cut any part of the tree at all! For this reason, good move ordering is very important, and is the focus of a lot of the effort of writing a good chess program. Assuming constantly **b** moves for each node visited and search depth **n**, the maximal number of leaves in alpha-beta is equivalent to minimax, $\mathbf{b^n}$. Taking always the best move first, it is $\mathbf{b^{ceil(n/2)} + b^{floor(n/2)} - 1}$. The minimal number of leaves is shown in following table which also demonstrates the odd-even effect:

	number of leaves with depth n and b = 40	
depth	worst case	best case
n	b^n	$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$
0	1	1
1	40	40
2	1,600	79
3	64,000	1,639
4	2,560,000	3,199
5	102,400,000	65,569
6	4,096,000,000	127,999
7	163,840,000,000	2,623,999
8	6,553,600,000,000	5,119,999

Table 1

The Algorithm

```
int alphaBeta( int alpha, int beta, int depthleft ) {
   if( depthleft == 0 ) return quiesce( alpha, beta );
   for ( all moves)  {
      score = -alphaBeta( -beta, -alpha, depthleft - 1 );
      if( score >= beta )
         return beta;   //  fail hard beta-cutoff
      if( score > alpha )
         alpha = score; // alpha acts like max in MiniMax
   }
   return alpha;
}
```

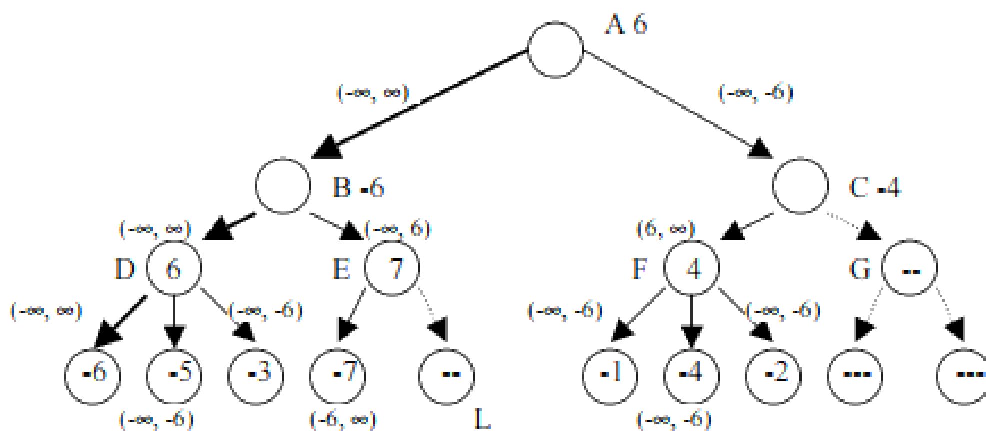


Figure 5: Alpha beta tree search

3.4.2 Enhancements in Alpha Beta Algorithm

Few enhancements that are used to speed up the Alpha Beta are described below. They are essential since concurrency will increase as we move on to parallel search. So each thread must be executing an efficient code already.

Move Ordering

The efficiency of the alpha-beta rule depends on the move search order. For instance, if we tend to swap the positions of D, E and F, G in Figure 5, then a full tree search is going to be required to see the worth at the root. To maximize the efficiency of alpha-beta cut-offs, the “best” move must be evaluated initially at every node. Hence several ordering schemes are made for ordering the moves in a very best to worst order. Some techniques are comparable to iterative deepening, transposition tables, killer moves and therefore the history heuristic have been quite eminent and reliable.

Iterative – Deepening

Iterative deepening was originally created as a time management mechanism for game tree search. It handles the matter that however we must always opt for the search depth depends on the quantity of your time the search can take. A simple fixed depth is inflexible because of the variation in the amount of time the program takes per move. So David Slate and Larry Atkin introduced the notion of iterative deepening: start from 1-ply search, repeatedly extend the search by one ply until we run out of time, then report the best move from the previous completed iteration. It seems to waste time since only the result of last search is used. But fortunately, due to the exponential nature of game tree search, the overhead cost of the preliminary D-1 iterations is only a constant fraction of the D-ply search. Besides providing good control of time, iterative deepening is usually more efficient than an equivalent direct search. The reason is that the results of previous iterations can improve the move ordering of new iteration, which is critical for efficient searching. So compared to the additional cut-offs for the D-ply search because of improved move order, the overhead of iterative deepening is relatively small. Many techniques have proved to further improve the move order between iterations. In this thesis, we focus on three of them: transposition tables, killer moves and history heuristic.

Transposition Tables

In practice, interior nodes of game trees are not always distinct. The same position may be re-visited multiple times. Therefore, we can record the information of each sub-tree searched in a transposition table. The information saved typically includes the score, the best move, the search depth, and whether the value is an upper bound, a lower bound or an exact value. When an identical position occurs again, the previous result can be reused in two ways:

1) If the previous search is at least the desired depth, then the score corresponding to the position will be retrieved from the table. This score can be used to narrow the search window when it is an upper or lower bound, and returned as a result directly when it is an exact value.

2) Sometimes the previous search is not deep enough. In such a case the best move from the previous search can be retrieved and should be tried first. The new search can have a better move ordering, since the previous best move, with high probability, is also the best for the current depth. This is especially helpful for iterative deepening, where the interior nodes will be re-visited repeatedly. To minimize access time, the transposition table is typically constructed as a hash table with a hash key generated by the well-known Zobrist method.

Killer Move Heuristic

The transposition table can be used to suggest a likely candidate for best move when an identical position occurs again. But it can neither order the remaining moves of revisited positions, nor give any information on positions not in the table. So the “killer move” heuristic is frequently used to further improve the move ordering. The philosophy of the killer move heuristic is that different positions encountered at the same search depth may have similar characters. So a good move in one branch of the game tree is a good bet for another branch at the same depth. The killer heuristic typically includes the following procedures:

1) Maintain killer moves that seem to be causing the most cutoffs at each depth. Every successful cutoff by a non-killer move may cause the replacement of the killer moves.

2) When the same depth in the tree is reached, examine moves at each node to see whether they match the killer moves of the same depth; if so, search these killer moves before other moves are searched.

History Heuristic

The history heuristic, which is first introduced by Schaeffer, extends the basic idea of the killer move heuristic. As in the killer move heuristic, the history heuristic also uses a move’s previous effectiveness as the ordering criterion. But it maintains a history for every legal move instead of only for killer moves. In addition, it accumulates and shares previous search information throughout the tree, rather than just among nodes at the same search depth. Below we illustrate how to implement the history heuristic in the alpha-beta algorithm. The bold lines are the part related to the history heuristic. Note that every time a move causes a cutoff or yields the best minimax value, the associated history score is increased. So the score of a move in the history table is in proportion to its history of success.

```
// pos : current board position
// d: search depth
// alpha: lower bound of expected value of the tree
// beta: upper bound of expected value of the tree
// Search game tree to given depth, and return evaluation of root.
int AlphaBeta(pos, d, alpha, beta)
{
        if (d=0 || game is over)
                return Eval (pos);              // evaluate leaf position from current player's standpoint
        score = - INFINITY;                     // preset return value
        moves = Generate(pos);                  // generate successor moves
        for i =1 to sizeof(moves) do  // rating all moves
                rating[i] = HistoryTable[ moves[i] ];
        Sort( moves, rating );                  // sorting moves according to their history scores
        for i =1 to sizeof(moves) do {          // look over all moves
                Make(moves[i]);                 // execute current move
                cur = - AlphaBeta(pos, d-1, -beta, -alpha);    //call other player, and switch sign of returned value
                if (cur > score) {        // compare returned value and score value, note new best score if necessary
```

```
                score = cur;
                bestMove = moves[i]; // update best move if necessary
            }
            if (score > alpha) alpha = score;  //adjust the search window
            Undo(moves[i]);                   // retract current move
            if (alpha >= beta) goto done;  // cut off
        }
        done:
            // update history score
            HistoryTable[bestMove] = HistoryTable[bestMove] + Weight(d);
            return score;
    }
```

PVS / NegaScout

NegaScout and Principal Variation Search (PVS) are two similar refinements of alpha-beta using minimal windows. The basic idea behind NegaScout is that most moves after the first will result in cutoffs, so evaluating them precisely is useless. Instead it tries to prove them inferior by searching a minimal alpha-beta window first. So for subtrees that cannot improve the previously computed value, NegaScout is superior to alphabeta due to the smaller window. However sometimes the move in question is indeed a better choice. In such a case the corresponding subtree must be revisited to compute the precise minimax value. Figure 6 demonstrates NegaScout search procedures. Note that for the leftmost child moves, line 9 represents a search with the interval (-beta, -alpha) whereas a minimal window search for the rest of children. If the minimal window search fails, i.e., (cur > score) at line 10, that means the corresponding subtree must be revisited with a more realistic window (-beta, -cur) (line 15) to determine its exact value. The conditions at line 11 show that this re-search can be exempted in only two cases: first, if the search performed at line 9 is identical to actual alpha-beta search, i.e., n=beta, and second, if the search depth is less than 2. In that case NegaScout's search always returns the precise minimax value.

```
        // d: search depth
        // alpha: lower bound of expected value of the tree
        // beta: upper bound of expected value of the tree
        // Search game tree to given depth, and return evaluation of root.
1       int NegaScout(pos, d, alpha, beta) {
2           if (d=0 || game is over)
3               return Eval (pos);          // evaluate leaf position from current player's standpoint
4           score = - INFINITY;             // preset return value
5           n = beta;
6           moves = Generate(pos);          // generate successor moves
7           for i =1 to sizeof(moves) do {  // look over all moves
8               Make(moves[i]);             // execute current move
9               cur = -NegaScout(pos, d-1, -n, -alpha);
10              if (cur > score) {
11                  if (n = beta) OR (d <= 2)
12                      score = cur; // compare returned value and score value, note new best score if necessary
13                  else
15                      score = -NegaScout(pos, d-1, -beta, -cur);
                }
16              if (score > alpha) alpha = score;  //adjust the search window
17              Undo(moves[i]);             // retract current move
18              if (alpha >= beta) return alpha;  // cut off
19              n = alpha + 1;
            }
20          return score;
        }
```

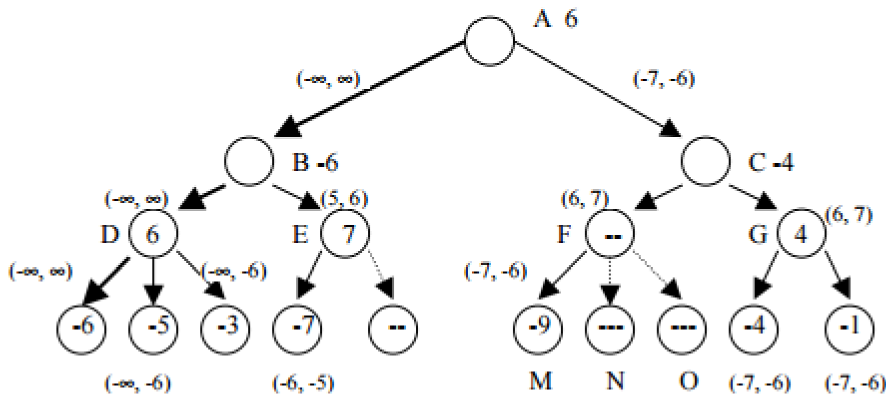


Figure 6: Alpha beta tree with PVS enhancement

A good move ordering is even more favorable to NegaScout than to alpha-beta. The reason is that the number of re-searches can be dramatically reduced if the moves are sorted in a best-first order. So other move ordering enhancements such as iterative deepening, the killer heuristic, etc, can be expected to give more improvement to NegaScout than to alpha-beta. B -6 D 6 -6 -5 -3 -7 -- E 7 C -4 F -- -9 --- --- -4 -1 G 4 A 6 (-7, -6) (-∞, ∞) (-∞, ∞) (5, 6) (-∞, -6) (-∞, -6) (-6, -5) (6, 7) (-7, -6) (-∞, ∞) M N O (6, 7) (-7, -6) (-7, -6) When performing a re-search, NegaScout has to traverse the same subtree again. This expensive overhead of extra searches can be prevented by caching

previous results. Therefore a transposition table of sufficient size is always preferred in NegaScout.

3.5 Quiescence Search

A fixed-depth approximate algorithm searches all possible moves to the same depth. At this maximum search depth, the program depends on the evaluation of intermediate positions to estimate their final values. But actually all positions are not equal. Some “quiescent” positions can be assessed accurately. Other positions may have a threat just beyond the program’s horizon (maximum search depth), and so cannot be evaluated correctly without further search. The solution, which is called quiescence search, is increasing the search depth for positions that have potential and should be explored further. For example, in chess positions with potential moves such as capture, promotions or checks, are typically extended by one ply until no threats exist. Although the idea of quiescence search is attractive, it is difficult to find out a good way to provide automatic extensions of non-quiescence positions.

Pseudocode

```
int Quiesce( int alpha, int beta ) {
    int stand_pat = Evaluate();
    if( stand_pat >= beta )
        return beta;
    if( alpha < stand_pat )
        alpha = stand_pat;

    until( every_capture_has_been_examined )  {
        MakeCapture();
        score = -Quiesce( -beta, -alpha );
        TakeBackMove();

        if( score >= beta )
            return beta;
        if( score > alpha )
           alpha = score;
    }
    return alpha;
}
```

3.6 Parallel Search

Parallel Search, also known as Multithreaded Search or SMP Search, is a way to increase search speed by using additional processors. This topic that has been gaining popularity recently with multiprocessor computers becoming widely available. Utilizing these additional processors is an interesting domain of research, as traversing a search tree is inherently serial. Several approaches have been devised, with the most popular today being Young Brothers Wait Concept and Shared Hash Table.

A subtype of parallel algorithms, distributed algorithms are algorithms designed to work in cluster computing and distributed computing environments, where additional concerns beyond the scope of "classical" parallel algorithms need to be addressed.

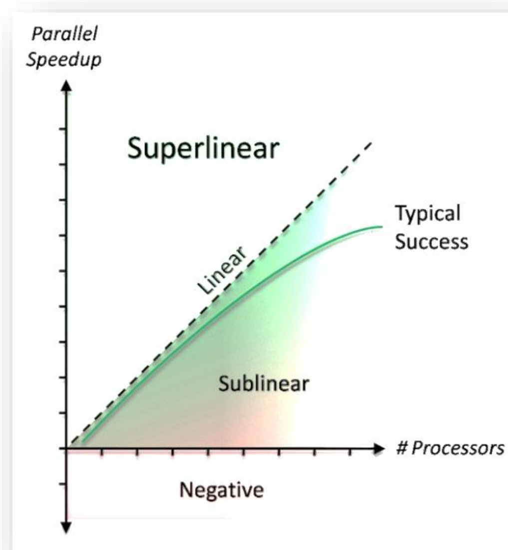


Figure 7: Comparison between Ideal and Practical speedup

3.6.1 Threads

Threads are the smallest program sequence that are managed independently by a scheduler.

3.6.2 Young Brothers Wait Concept

YBWC is a basic concept for a parallel alpha-beta search coined by Rainer Feldmann et al. in 1986, introduced 1989 to a wider audience in the ICCA Journal paper *Distributed Game-Tree Search*. Brothers are sibling nodes, the oldest brother is searched first, younger brothers, to be searched in parallel, have to wait until the oldest brother did not yield in a beta-cutoff and did possibly narrow the bounds in case of an alpha increase.

3.6.3 Delay

The Young Brothers Wait Concept, which in some situations delays the use of parallelism until subtrees are available which are relevant for the final result with high probability, prevents processors from searching irrelevant subtrees and reduces the search overhead. The use of the YBWC is possible only in combination with good load balancing possibilities.

```
The eldest son of any node must be completely evaluated before younger
brothers of that node may be transmitted.
```

3.6.4 Distributed Game-Tree Search

The Principal Variation Splitting (PVS) algorithm evaluates right sons of game-tree nodes with a minimal window and re-evaluates them only if necessary.

Alternatively, game-tree nodes are evaluated in parallel only if they had acquired an alpha-beta bound before. Yet another approach applies in a distributed chess program running on a hypercube: if the transposition table proposes some move for a game position, then this move is tried first. Parallel evaluation of the other moves is started only of the evaluation of the transposition move yields no cutoff.

3.7 GPGPU Concepts and CUDA Programming Language

General-purpose computing on graphics processing units is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). The use of multiple graphics cards in one computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing. In addition, even a single GPU-CPU framework provides advantages that multiple CPUs on their own do not offer due to the specialization in each chip.

Essentially, a GPGPU pipeline is a kind of parallel processing between one or more GPUs and CPUs that analyzes data as if it were in image or other graphic form. While GPUs generally operate at lower frequencies, they usually have many times more cores to make up for it (up to hundreds at least) and can, thus, operate on pictures and graphical data effectively much faster, dozens or even hundreds of times faster than a traditional CPU, migrating data into graphical form and then using the GPU to "look" at it and analyze it can result in profound speedup.

Originally, data was simply passed one-way from a CPU to a GPU, then to a display device. However, as time progressed, it became valuable for GPUs to store at first simple, then complex structures of data to be passed back to the CPU that analyzed an image, or a set of scientific-data represented as a 2D or 3D format that a graphics card can understand. Because the GPU has access to every draw operation, it can analyze data in these forms very quickly, whereas a CPU must poll every pixel or data element much more slowly, as the speed of access between a CPU and its larger pool of random-access memory (or in an even worse case, a hard drive) is slower than GPUs and graphics cards, which typically contain smaller amounts of more expensive memory that is very much faster to access. Transferring the portion of the data set to be actively analyzed to that GPU memory in the form of textures or other easily readable GPU forms results in speed increase.

THE CUDA PARALLEL COMPUTING PLATFORM

The CUDA parallel computing platform provides a few simple C and C++ extensions that enable expressing fine-grained and coarse-grained data and task parallelism. The programmer can choose to express the parallelism in high-level languages such as C, C++, Fortran or open standards as OpenACC directives. The CUDA parallel computing platform is now widely deployed with 1000s of GPU-accelerated applications and 1000s of published research papers.

Below is the sample of a standard C code, and in comparison to Parallel CUDA C code

Standard C Code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

Parallel C Code

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

Figure 8: Comparison of a C++ code and CUDA code

3.8 APPROACH

In this we extend the general algorithm of Monte- Carlo Tree Search, to wider and latest emerging field of GPGPU computing, where the Monte-Carlo Tree Search is implemented on GPU instead of a CPU earlier. 2 more steps or schemes are added to the algorithm. Later we will compare the performance of each scheme, and also calculate the speed-up.

Monte Carlo Tree Search

A simulation is defined as a series of random moves which are performed until the end of a game is reached (until neither of the players can move). The result of this simulation can be successful, when there was a win in the end or unsuccessful otherwise. So, let every node i in the tree store the number of simulations T (visits) and the number of successful simulations $Si.$ The general MCTS algorithm comprises 4 steps (Figure 9) which are repeated.

MCTS iteration steps

1) **Selection**: - a node from the game tree is chosen based on the specified criteria. The value of each node is calculated and the best one is selected. The formula used to calculate the node value is the Upper Confidence Bound (UCB).

$$UCB_i = \frac{S_i}{t_i} + C * \sqrt{\frac{logT}{t_i}}$$

T_i *- total number of simulations for the parent of node* i
C *- a parameter to be adjusted*

Suppose that some simulations have been performed for a node, first the average node value is taken and then the second term which includes the total number of simulations for that node and its parent. The first one provides the best possible node in the analyzed tree (exploitation), while the second one is responsible for the tree exploration. That means that a node which has been rarely visited is more likely to be chosen, because the value of the second terms is greater.

2) **Expansion**: - one or more successors of the selected node are added to the tree depending on the strategy. This point is not strict, in our implementation I add one node per iteration, so this number can be different.

3) **Simulation**: - for the added node(s) perform simulation(s) and update the node(s) values (successes, total) – here in the CPU implementation, one simulation per iteration is performed. In the GPU implementations, the number of simulations depends on the number of threads, blocks Selection Expansion Simulation Backpropagation Repeat until time is left and the method (leaf of block parallelism). I.e. the number of simulations can be equal to 1024 per iteration for 4 block 256 thread configuration using the leaf parallelization method.

4) **Backpropagation**: - update the parents' values up to the root nodes. The numbers are added, so that the root node has the total number of simulations and successes for all of the nodes and each node contains the sum of values of all of its successors. For the root/block parallel methods, the root node has to be updated by summing up results from all other trees processed in parallel.

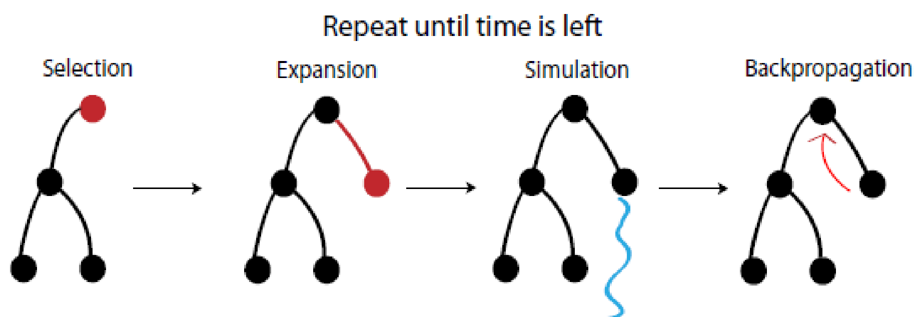


Figure 9: MCTS Steps of iteration

3.9 GPU IMPLEMENTATION

In the GPU implementation, 2 approaches are considered and discussed. The first one is the simple leaf parallelization, where one GPU is dedicated to one MCTS tree and each GPU thread performs an independent simulation from the same node. Such a parallelization should provide much better accuracy when the great number of GPU threads is considered.

The second approach (Figure 10c), is the block parallelization method. It combines both aforementioned schemes. Root parallelism is an efficient method of parallelization MCTS on CPUs. It is more efficient than simple leaf parallelization, because building more trees diminishes the effect of being stuck in a local extremum/increases the chances of finding the true global maximum. Therefore having n processors it is more efficient to build n trees rather than performing n parallel simulations in the same node. Given that a problem can have many local maximas, starting from one point and performing a search might not be very accurate in the basic MCTS case. The second one, leaf parallelism should diminish this effect by having more samples from a given point. The third one is root parallelism. Here a single tree has the same properties as each tree in the sequential approach except for the fact that there are many trees and the chance of finding the global maximum increases with the number of trees. The last, our proposed algorithm, combines those two, so each search should be more accurate and less local at the same time.

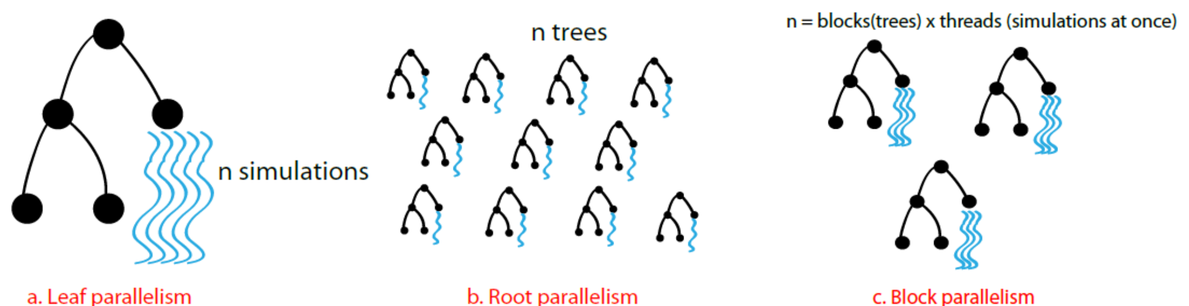


Figure 10: Different types of parallism approach

5) Leaf-parallel scheme: This is the simplest parallelization method in terms of implementation. Here GPU receives a root node from the CPU controlling process and performs n simulations, where n depends on the dimensions of the grid (block size and number of blocks). Afterwards the results are written to an array in the GPU's memory (0 = loss, 1 = victory) and CPU reads the results back. Based on that,

the obtained result is the same as in the basic CPU version except for the fact that the number of simulations is greater and the accuracy is better.

6) Block-parallel scheme: To maximize the GPU's simulating performance some modifications had to be introduced. In this approach the threads are grouped and a fixed number of them is dedicated to one tree. This method is introduced due to the hierarchical GPU architecture, where threads form small SIMD groups called warps and then these warps form blocks. It is crucial to find the best possible job division scheme for achieving high GPU performance. The trees are still controlled by the CPU threads, GPU simulates only. That means that at each simulation step in the algorithm, all the GPU threads start and end simulating at the same time and that there is a particular sequential part of this algorithm which decreases the number of simulations per second a bit when the number of blocks is higher. This is caused by the necessity of managing each tree by the CPU. On the other hand the more the trees, the better the performance. In our experiments the smallest number of threads used is 32 which corresponds to the warp size.

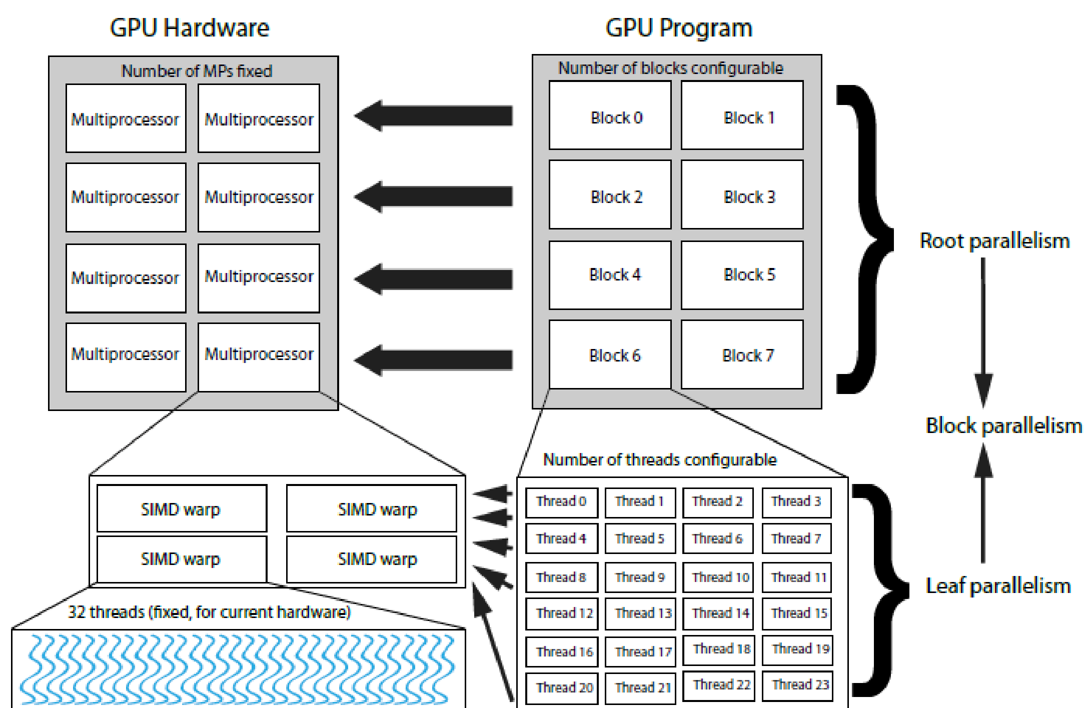


Figure 11: Block diagram with flowchart of parallelism approach on GPU hardware

Hybrid CPU-GPU processing

I observed that the trees formed by our algorithm using GPUs are not as deep as the trees when CPUs and root parallelism are used. It is caused by the time spent on each GPU's kernel execution. CPU performs quick single simulations, whereas GPU needs more time, but runs thousands, of threads at once. It would mean that the results are

less accurate, since the CPU tree grows faster in the direction of the optimal solution. As a solution I experimented on using hybrid CPU-GPU algorithm. In this approach, the GPU kernel is called asynchronously and the control is given back to CPU. Then CPU operates on the same tree (in case of leaf parallelism) or trees (block parallelism) to increase their depth. It means that while GPU processes some data, CPU repeats the MCTS iterative process and checks for the GPU kernel completion.

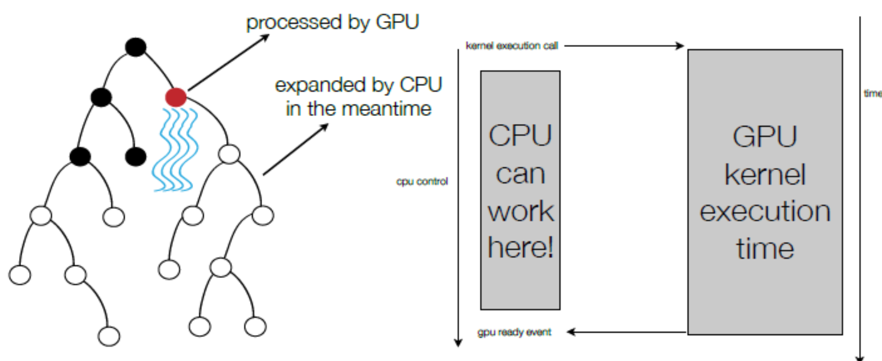


Figure 12: Hybrid approach

I compare the speed(Figure 13) and results (Figure 14) of leaf parallelism and block parallelism using different block sizes. The block size and their number corresponds to the hardware's properties. In those graphs a GPU Player is playing against one CPU core running sequential MCTS.

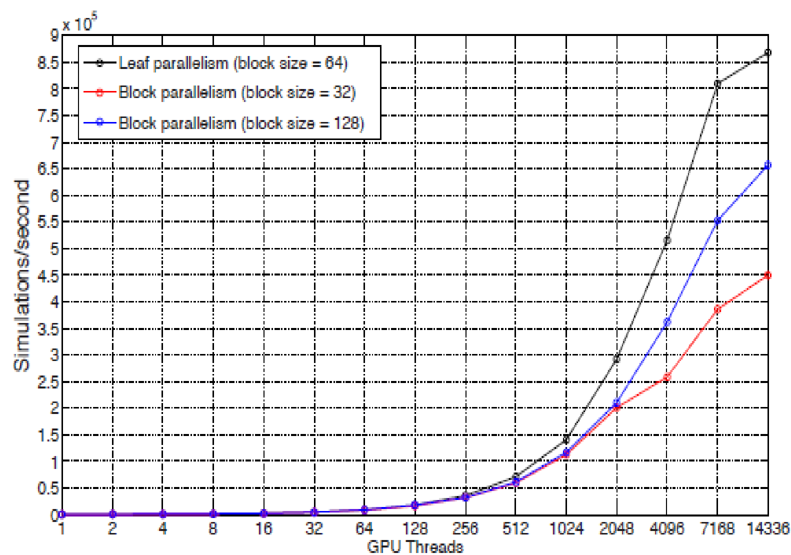


Figure 13: Simulations per second or speedup graph

The main aspect of the analysis is that despite running fewer simulations in a given amount of time using block parallelism, the results are much better compared to leaf parallelism, where the maximal winning ratio stops at around 0.75 for 1024 threads (16 blocks of 64 threads).

The results are better when the block size is smaller (32), but only when the number of threads is small (up to 4096, 128 blocks/trees), then the lager block case (128) performs better. It can be observed in Figure 5 that as I decrease the number of threads per block and at the same time increase the number of trees, the number of simulations per second decreases. This is due to the CPU's sequential part.

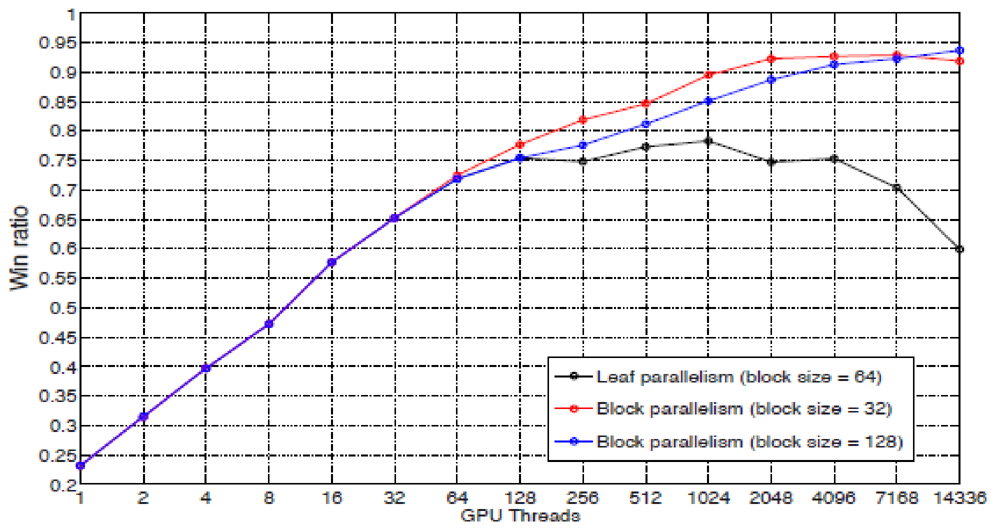


Figure 14: GPU Outperforms CPU (Results)

In Figure 7 and 8 I also show a different type of result, where the X-axis represents current game step and the Y-axis is the average point difference between 2 players. In Figure 7 I observe that one GPU outperforms 256 CPUs in terms both intermediate and final scores. Also I see that the characteristics of the results using CPUs and GPU are slightly different, where GPU is stronger at the beginning. I believe that it might be caused by the larger search space and therefore I conclude that later the parallel effect of the GPU is weaker, as the number of distinct samples decreases. Another reason for this is mentioned depth of the tree which is lower in the GPU case.

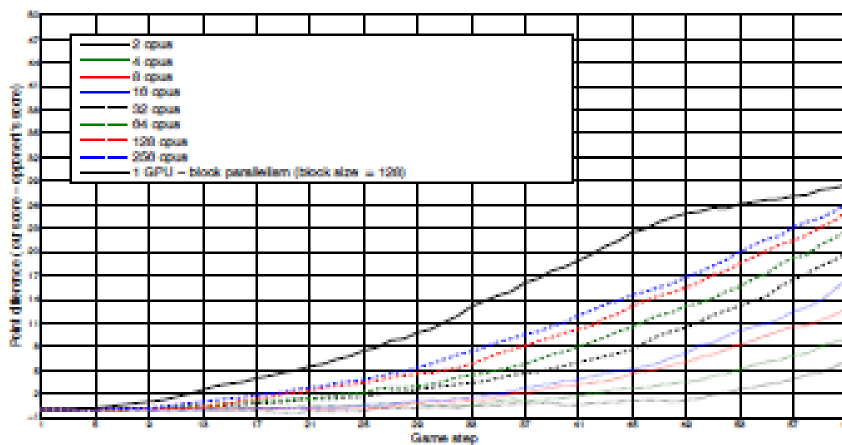


Figure 15: Comparison of 1 GPU against many number of CPUs

Also I show that using our hybrid CPU/GPU approach both the tree depth and the result are improved as expected especially in the last phase of the game.

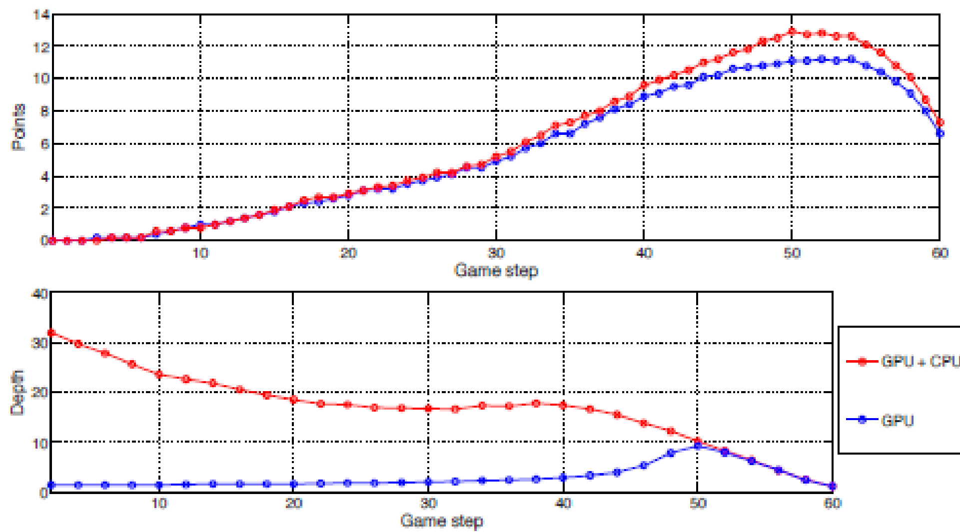


Figure 16: Comparison between hybrid approach and normal approach

Chapter 4

Conclusion

Performance comparison was done between a hybrid approach and non hybrid approach and the results proved it to be successful. So we have a code which performs better than previous approach.

Chapter 5

Future Work

- Application of the algorithm to other domain. A more general task can and should be solved by the algorithm.

- Scalability analysis. This is a major challenge and requires analyzing certain number of parameters and their effect on the overall performance. Currently I implemented the MPI-GPU version of the algorithm, but the results are inconclusive, there are several reason why the scalability can be limited including Reversi itself.

- Application of algorithm to games with a huge branching factor like chess, checkers etc. will simplify the current processing of such games.

- Major speed-up in search used in artificial intelligent machines.

- The evolution in game tree search algorithms as well as parallel libraries and hardware will not only lead to solve complex games that were not solvable before, but will also lead to solve more problems in real life. Many algorithms designed during the past to run in parallel on CPU or GPU were able to reach 6x speedup on CPU for 8-core processors and 40x speedup in 14 depth on GPU.

References

[1] Yun-Ching Liu, Yoshimasa Tsuruoka (**2015**). *Adapting Improved Upper Confidence Bounds for Monte-Carlo Tree Search.*

[2] Chu-Hsuan Hsueh, I-Chen Wu, Wen-Jie Tseng, Shi-Jim Yen, Jr-Chang Chen (**2015**). *Strength Improvement and Analysis for an MCTS-Based Chinese Dark Chess Program.*

[3] General Purpose Computing on GPUs, gpgpu.org

[4] CUDA Programming, http://www.nvidia.com/object/cuda-parallel-computing-platform.html

[5] Chess Programming Hub, https://chessprogramming.wikispaces.com/

[6] Kamil Rocki, Reiji Suda, Large-Scale Parallel Monte Carlo Tree Search on GPU

[7] Monte Carlo Tree Search (MCTS) research hub, http://www.mcts-hub.net/index.html

[8] Kocsis L., Szepesvari C.: Bandit based Monte-Carlo Planning, 15th European. Conference on Machine Learning Proceedings, 2006

[9] Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik: Parallel Monte-Carlo Tree Search, Computers and Games: 6th International Conference, 2008

[10] Rocki K., Suda R.: Massively Parallel Monte Carlo Tree Search, Proceedings of the 9th International Meeting High Performance Computing for Computational Science, 2010

[11] Romaric Gaudel, Michle Sebag - Feature Selection as a one player game (2010)

[12] O. Teytaud et. al, High-dimensional planning with Monte-Carlo Tree Search (2008)

[13] Manohararajah V. Parallel Alpha-Beta Search on Shared Memory Multiprocessors. M.Sc. Thesis. Toronto: University of Toronto; 2001

[14] Lu C-PP. Parallel Search of Narrow Game Trees. M.Sc. Thesis. Edmonton: University of Alberta; 1993.

[15] Sanders J, Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. New Jersey: Addison Wesley; 2010.

Appendix A

Snapshots of Code and System

1. Evaluation function

```
// Initialize attack and king safety bitboards
init_eval_info<WHITE>(pos, ei);
init_eval_info<BLACK>(pos, ei);

ei.attackedBy[WHITE][ALL_PIECES] |= ei.attackedBy[WHITE][KING];
ei.attackedBy[BLACK][ALL_PIECES] |= ei.attackedBy[BLACK][KING];

// Do not include in mobility squares protected by enemy pawns or occupied by our pawns or king
Bitboard mobilityArea[] = { ~(ei.attackedBy[BLACK][PAWN] | pos.pieces(WHITE, PAWN, KING)),
                            ~(ei.attackedBy[WHITE][PAWN] | pos.pieces(BLACK, PAWN, KING)) };

// Evaluate pieces and mobility
score += evaluate_pieces<KNIGHT, WHITE, Trace>(pos, ei, mobility, mobilityArea);
score += apply_weight(mobility[WHITE] - mobility[BLACK], Weights[Mobility]);

// Evaluate kings after all other pieces because we need complete attack
// information when computing the king safety evaluation.

score +=  evaluate_king<WHITE, Trace>(pos, ei)
        - evaluate_king<BLACK, Trace>(pos, ei);


// Evaluate tactical threats, we need full attack information including king
score +=  evaluate_threats<WHITE, Trace>(pos, ei)
        - evaluate_threats<BLACK, Trace>(pos, ei);

// Evaluate passed pawns, we need full attack information including king
```

2. Iterative deepening

```
// Step 10. Internal iterative deepening (skipped when in check)
if (    depth >= (PvNode ? 5 * ONE_PLY : 8 * ONE_PLY)
    && !ttMove
    && (PvNode || ss->staticEval + 256 >= beta))
{
    Depth d = 2 * (depth - 2 * ONE_PLY) - (PvNode ? DEPTH_ZERO : depth / 2);
    ss->skipNullMove = true;
    search<PvNode ? PV : NonPV, false>(pos, ss, alpha, beta, d / 2, true);
    ss->skipNullMove = false;

    tte = TT.probe(posKey);
    ttMove = tte ? tte->move() : MOVE_NONE;
}
```

3. Quiescenece Search

```
Value qsearch(Position& pos, Stack* ss, Value alpha, Value beta, Depth depth) {

  const bool PvNode = NT == PV;

  assert(NT == PV || NT == NonPV);
  assert(InCheck == !!pos.checkers());
  assert(alpha >= -VALUE_INFINITE && alpha < beta && beta <= VALUE_INFINITE);
  assert(PvNode || (alpha == beta - 1));
  assert(depth <= DEPTH_ZERO);

  StateInfo st;
  const TTEntry* tte;
  Key posKey;
  Move ttMove, move, bestMove;
  Value bestValue, value, ttValue, futilityValue, futilityBase, oldAlpha;
  bool givesCheck, evasionPrunable;
  Depth ttDepth;

  // To flag BOUND_EXACT a node with eval above alpha and no available moves
  if (PvNode)
      oldAlpha = alpha;

  ss->currentMove = bestMove = MOVE_NONE;
  ss->ply = (ss-1)->ply + 1;

  // Check for an instant draw or if the maximum ply has been reached
  if (pos.is_draw() || ss->ply > MAX_PLY)
      return ss->ply > MAX_PLY && !InCheck ? evaluate(pos) : DrawValue[pos.side_to_move()];
```

4. Multithreaded splitting of search

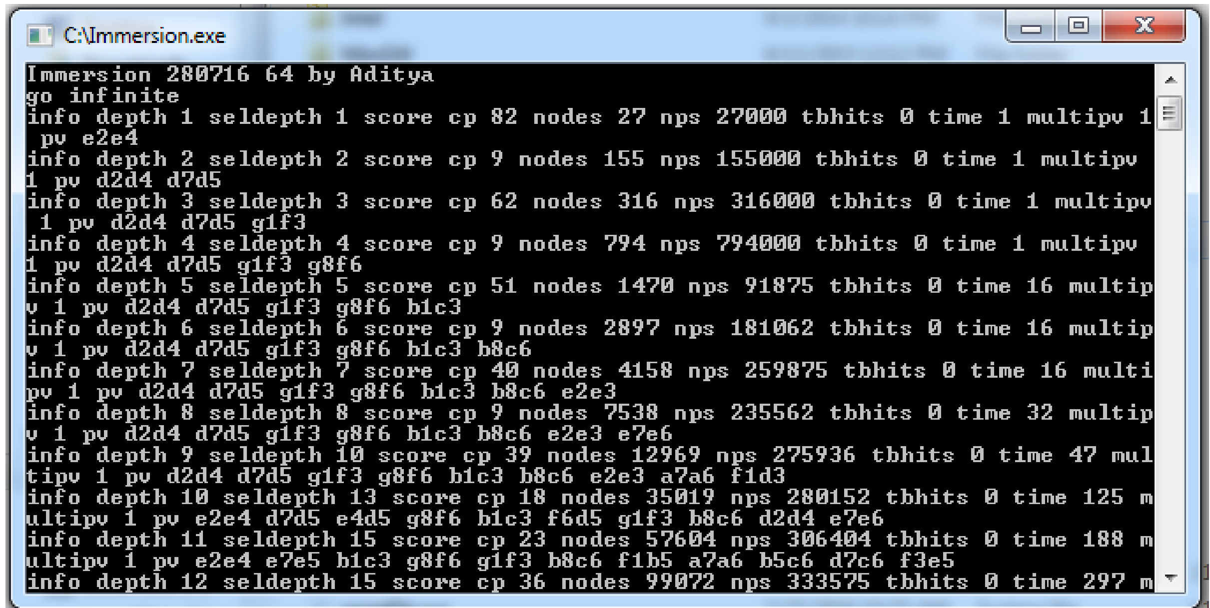
```
      // Step 19. Check for splitting the search
      if (   !SpNode
          &&  Threads.size() >= 2
          &&  depth >= Threads.minimumSplitDepth
          //&&  Threads.available_slave(thisThread)
          &&  (   !thisThread->activeSplitPoint
               || !thisThread->activeSplitPoint->allSlavesSearching)
          &&  thisThread->splitPointsSize < MAX_SPLITPOINTS_PER_THREAD)
      {
          assert(bestValue > -VALUE_INFINITE && bestValue < beta);

          thisThread->split<FakeSplit>(pos, ss, alpha, beta, &bestValue, &bestMove,
                                       depth, moveCount, &mp, NT, cutNode);

          if (Signals.stop || thisThread->cutoff_occurred())
              return VALUE_ZERO;

          if (bestValue >= beta)
              break;
      }
```

5. **Command line version run**



6. **GUI version of the engine (GUI used: Deep Fritz)**

7. **8 CPUs usage, significant speedup and depth increase**