Large Scale Monte Carlo Tree Search on GPU

GPUによる大規模モンテカルロ木探索

Doctoral Dissertation
博士論文

Kamil Marek Rocki
ロツキ カミル マレック

Submitted to Department of Computer Science,
Graduate School of Information Science and Technology,
The University of Tokyo
in partial fulfillment of the requirements for the degree of
Doctor of Information Science and Technology

Thesis Supervisor: Reiji Suda (須田礼仁)
Professor of Computer Science

August, 2011

Abstract: Monte Carlo Tree Search (MCTS) is a method for making optimal decisions in artificial intelligence (AI) problems, typically for move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search. Research interest in MCTS has risen sharply due to its spectacular success with computer Go and its potential application to a number of other difficult problems. Its application extends beyond games, and MCTS can theoretically be applied to any domain that can be described in terms of (state, action) pairs, as well as it can be used to simulate forecast outcomes such as decision support, control, delayed reward problems or complex optimization. The main advantages of the MCTS algorithm consist in the fact that, on one hand, it does not require any strategic or tactical knowledge about the given domain to make reasonable decisions, on the other hand algorithm can be halted at any time to return the current best estimate. So far, current research has shown that the algorithm can be parallelized on multiple CPUs.

The motivation behind this work was caused by the emerging GPU-based systems and their high computational potential combined with the relatively low power usage compared to CPUs. As a problem to be solved I chose to develop an AI GPU(Graphics Processing Unit)-based agent in the game of Reversi (Othello) and SameGame puzzle which provide sufficiently complex problems for tree searching with non-uniform structure. The importance of this research is that if the MCTS algorithm can be efficiently parallelized on GPU(s) it can also be applied to other similar problems on modern multi-CPU/GPU systems such as the TSUBAME 2.0 supercomputer. Tree searching algorithms are hard to parallelize, especially when GPU is considered. Finding an algorithm which is suitable for GPUs is crucial if tree search has to be performed on recent supercomputers. Conventional ones do not provide good performance, because of the limitations of the GPUs' architecture and the programming scheme, threads' communication boundaries. One of the problems is the SIMD execution scheme within GPU for a group of threads. It means that standard CPU parallel implementations such as root-parallelism fail. The other problem is the difficulty to generate pseudo-random numbers on GPU which is important for Monte Carlo methods. Available methods are usually very time consuming. Third of all, no current research work discusses scalability of the algorithm for millions of threads (when multiple GPUs are considered), so it is important to estimate to what extent the parallelism can be increased.

In this thesis I am proposing an efficient parallel GPU MCTS implementation based on the introduced 'block-parallelism' scheme which combines GPU SIMD thread groups and performs independent searches without any need of intra-GPU or inter-GPU communication. I compare it with a simple leaf parallel scheme which implies certain performance limitations. The obtained results show that using my GPU MCTS implementation on the TSUBAME 2.0 system one GPU's performance can be compared to 50-100 CPU threads depending on factors such as the search time and other MCTS parameters. The block-parallel algorithm provides better results than the naive leaf-parallel scheme which fail to scale well beyond 1000 threads on a single GPU. The block-parallel algorithm is approximately 4 times more efficient in terms of the number of CPU threads' results comparable with the GPU implementation. In order not to generate random numbers on GPU I am introducing an algorithm, where the numbers are transferred from the CPU for each GPU block accessible as a look-up table. This approach makes the time needed for random-sequence generation insignificantly small. In this thesis for the first time I am discussing the scalability of the algorithm for millions of threads. The program is designed in the way that it can be run on many nodes using Message Passing Interface (MPI) standard. As a method of evaluating my results I compared the results of multiple CPU cores and GPUs playing against the standard sequential CPU implementation. Therefore the algorithm's scalability is analyzed for multiple CPUs and GPUs. My results show that this algorithm implies almost no inter-node communication overhead and it scales linearly in terms of the number of simulation performed in a given time period. However, beyond a certain number of running threads, a lack of performance improvement was observed. I concluded that this limit is affected by the algorithm's implementation and it can be improved to some extent by tuning the parameters or adjusting the algorithm itself. The improvements I am proposing and analyzing are variance-based error estimation and simultaneous CPU/GPU execution. Using these two methods modifying the MCTS algorithm the overall effectiveness can be increased by 10-50% further, compared to the basis block-parallel implementation. Also, another factor considered is the criteria of estimating the performance is the overall score of the game (win percentage or the score). Not all the parameters in the MCTS algorithm are analyzed thoroughly in regarding the GPU's implementation and their importance considering scalability. This is caused by the certain limitations of the proposed evaluation method. As it is based on the average score, multiple games have to be played to get accurate results and time needed to aquire them is relatively long.

I am also stating the remaining problems to be solved such as estimating the algorithm's scalability for hundreds of GPUs and the overcoming the GPU latency for a single task execution. CPUs need very little time to perform a single search, whereas GPUs need to be loaded with data and run thousands of simulations at once. This implies different tree structures in the MCTS algorithm and also different characteristics of the obtained scores. Therefore I am also presenting results of my research for up to millions of GPU threads. I am discussing also several problems such as GPU and CPU implementation differences and power-usage comparisons. The GPU implementation consumes less power totally when the number of CPU threads needed to get comparable results is considered.

Keywords: Tree Search, Distributed, Parallel, Monte Carlo, GPU

Acknowledgments

This work was partially supported by Core Research of Evolutional Science and Technology (CREST) project "ULP-HPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies" of Japan Science and Technology Agency (JST) and Grant-in-Aid for Scientific Research of MEXT Japan.

I would like to express my sincere gratitude to Professor Reiji Suda for his valuable suggestions, encouragement and help in realizing my ideas. I would also like to thank Dr. Kazuki Yoshizoe for his precious suggestions leading to the presented results.

Last but not least, I wish to thank to my wife, Oana for her encouragement, patience and continuos support.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In artificial intelligence game trees have always been important as a structure used for representing states and possible outcomes, based on the decision (move) made. In order to choose the best possible move in a particular state, a look-ahead search must be performed. For small trees a thorough search is feasible given algorithms such as the minimax search or just brute force search. In case of more complex problems (such as Chess or Go) these algorithms fail. The number of performed operations grows exponentially with the tree depth and some of the problems solvable using tree search are proven to be NP-complete. Tree search algorithms play an important role in many applications such as theorem provers, expert systems and control systems. Tree searching is used whenever decision must be made. The reason why this thesis is related to game-tree search is that games provide a good testing environment for tree searching algorithm because of the precise rules and relatively large state-space. Therefore, it is easy to adjust the complexity of a particular problem, measure performance changes and compare the results with other works.

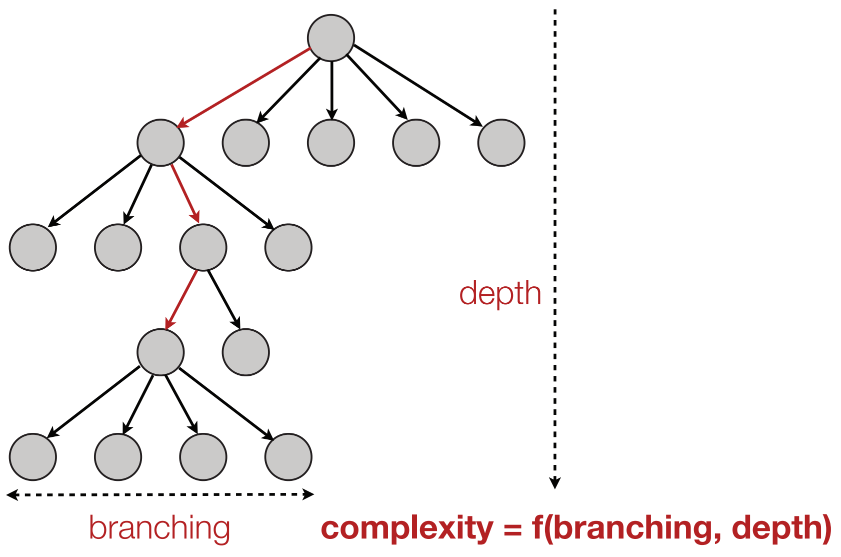


Figure 1.1: Tree traversal

Currently, parallelization seems to be the only way of accomplishing tasks like this one in shorter time. CPU clock has been fairly static in the last decade, yet the computational power kept increasing due to the rise of multi-core machines. Still GPU is far ahead in terms of parallelism and theoretical peak performance (Fig. 1.2). Modern supercomputers allow usage of thousands of CPUs and GPUs. Currently 3 out of top 5 supercomputers (Fig. 1.3, TOP500 list [1]) use GPUs. Therefore, not only is it important to have a parallelized tree searching algorithm, but also to be able to take advantage of such a number of threads. Tree searching algorithms are hard to parallelize, especially when GPU limitations are considered. And these emerging GPU-based systems and their high computational potential combined with relatively low power usage compared to CPUs were the main motivation to start this research. Finding an algorithm which is suitable for GPUs is crucial if tree search has to be performed on recent supercomputers. Conventional ones do not provide good performance, because of the limitations of the GPUs' architecture and the programming scheme, threads' communication boundaries. To be able to take advantage of its capabilities is quite difficult. The GPU program usually consists of thousands very small jobs working simultaneously to be efficient, because it is a SIMD device, therefore the algorithm design has to be different. Additionally it does not support recursive calls which is a significant difficulty when tree search is considered. In my current research I investigate how to effectively apply the parallel tree search on GPU. One of the reasons why this problem has not been solved before is that this architecture is quite new, novel applications are being developed and so far there are very few related works. The scale of parallelism is extreme here (i.e. using 1000s GPUs of 10000 threads gives 10 million threads working at the same time). The published work is related to hundreds or thousands of CPU cores at most.

My approach to solve to this problem utilizes a quite new algorithm, called 'Monte Carlo Tree Search' which I find much better and more suitable for this purpose than classic algorithms like 'Minimax'. This algorithm searches the tree in a semi-random way and provides a suboptimal result. It does not mean that the algorithm is completely random. There is a defined strategy of expanding a tree. Instead of performing a complete tree search, a number of simulations from a given state are executed to estimate the successors' values. It was the first algorithm to be at the novice human level in GO. I find the MCTS algorithm to be suitable for large scale parallelization. It requires very little communication and practically no communication between the running threads during the simulation phase which makes it especially suitable for GPUs. Although, I think that this algorithm is not perfect as I find certain issues of this approach which are presented and discussed later.

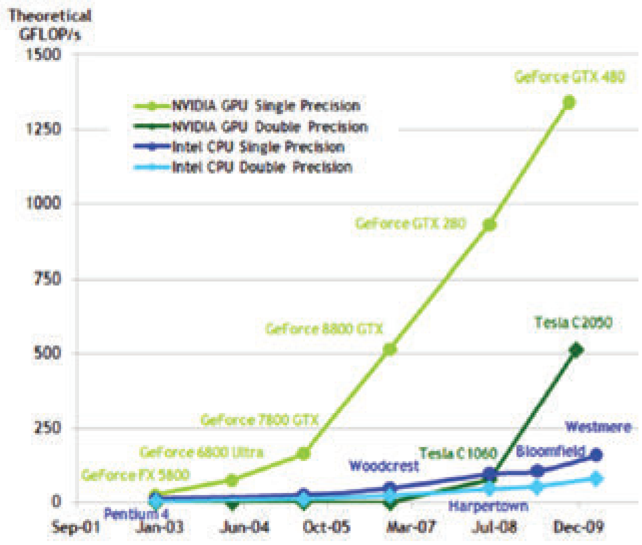
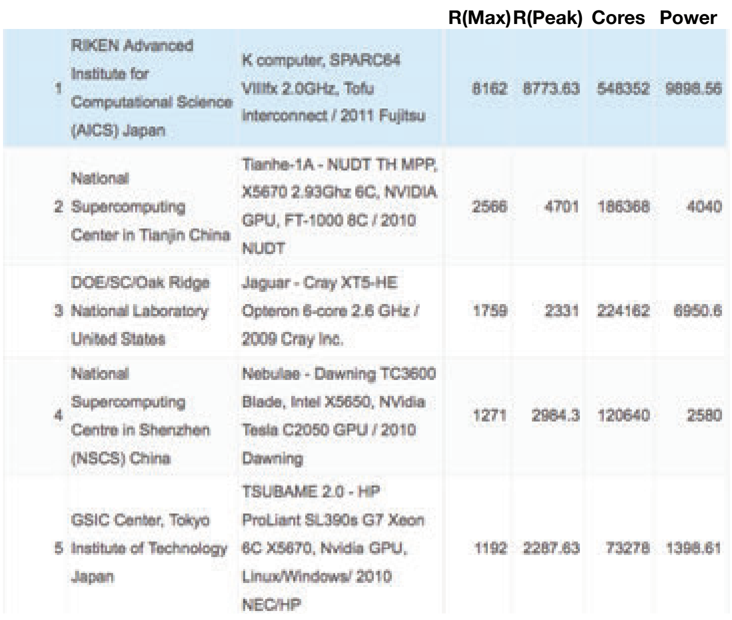


Figure 1.2: CPU and GPU peak performance improvement (2001-2009), source: NVIDIA



			R(Max)	R(Peak)	Cores	Power
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	8162	8773.63	548352	9898.56
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	2566	4701	186368	4040
3	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	1759	2331	224162	6950.6
4	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	1271	2984.3	120640	2580
5	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows/ 2010 NEC/HP	1192	2287.63	73278	1398.61

Figure 1.3: TOP 500 list, top 5 (June 2011)

Summarizing, the importance of this research is caused by the variety of applications of tree search algorithms such as decision support, control, delayed reward problems, complex optimization or any other where [state, action] pair can be used to describe a domain. It is a very basic problem in computer science, yet challenging when it comes to large scale parallelization. Additionally, now GPUs provide a great advantage over CPUs in terms of performing multiple simulations at the same time and use less power. Especially considering modern heterogenous supercomputer systems such as the GPU-equipped TSUBAME 2.0 supercomputer in Tokyo (No.5 in Figure 1.3), this research becomes very important.

Contents

1.1 Tree search

In computer science, tree searching is one of the methods to solve formulated problems. Having defined a problem, the search is performed through its *state space*. Problem-solving agents decide what to do by finding sequences of actions that lead to desirable states. Tree searching techniques use an explicit *search tree* that is generated by the initial state and the successor function that together define the state space. A pair $\{state, possible\ actions\}$ is needed to define a node or a search node. For more detailed definition of this process and theoretical background, please refer to Russell and Norvig[2]. In short, in order to perform tree searching the following need to be defined:

- Tree root - The initial state
- Action generating function (or the generating function) - Initially the *(possible actions)* set is empty, this function determines what possible actions are possible, based on the given state. So, the form of this function is as follows:

$$actions = generateActions(state)$$

- Successor function - This is the tree expanding function, takes a pair

$$\{state, possibleaction_i\}$$

where:

$$possibleaction_i \in possibleactions$$

and

$$i \text{ is the } ith \text{ possible action}$$

and returns a child of this state - the new state, after the chosen action is performed.

- Terminal condition - Defines when the search ends. The search typically ends when a global condition is met (i.e. time limit) or when a defined (goal) state is reached.
- Search strategy - Determines in which manner the states are expanded, which state will be chosen for expansion next. This is the essence of this method, the *search algorithm*.

Figure 1.4 presents an example of tree searching steps. The sequence may vary, which means that depending on the implementation the steps' order may be different.

1.2 Game trees

There are many ways to represent tree nodes, one of the simplest ones is to define a tree node as a structure containing the following:

- STATE: the state in the state space to which the node corresponds.
- PARENT-NODE LINK: the node in the search tree that generated this node. The most important point as the it creates the tree structure.
- ACTIONS: the possible actions from a given state. If a tree node's 'ACTIONS' set is empty it is called a *terminal state* or a *leaf*.

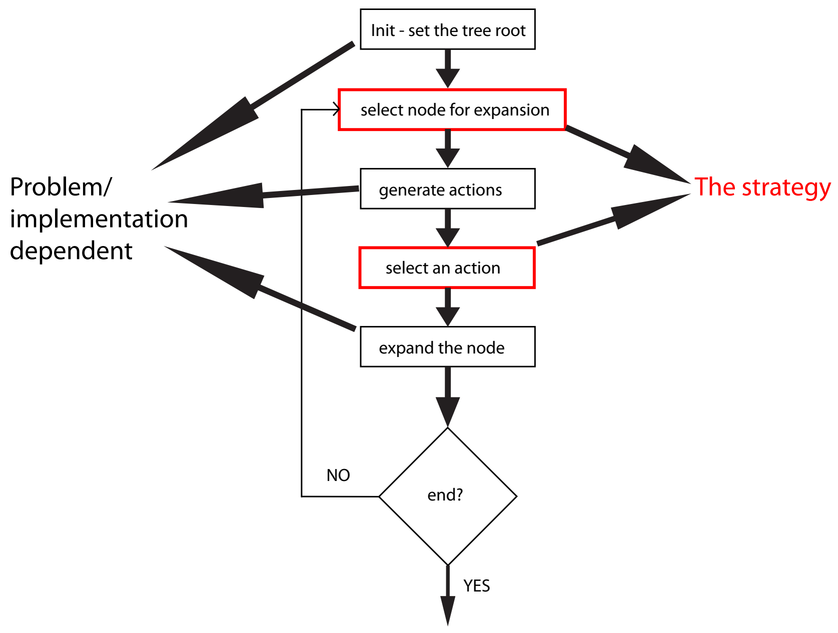


Figure 1.4: An example of tree searching algorithm

- PATH-COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers. Alternatively it may also be based on an evaluation function of a given state.

Additionally, the action that was applied to the parent to generate the node may be added to the tree node structure.

The are measures to characterize a problem's complexity such as the branching factor (may be average if nonuniform) and tree depth (also may be average). The branching factor is just a number of children being generated for a single node. If the branching factor is uniform, then the possible number of states and the state space's size equals $(branchingfactor)^{depth}$.

Figure 1.5 presents an example tree for the tic-tac-toe game. For more details about the tree search and game trees, please refer to the 3rd chapter of the aforementioned book by Russell and Norvig[2] as being a really exhaustive source of information.

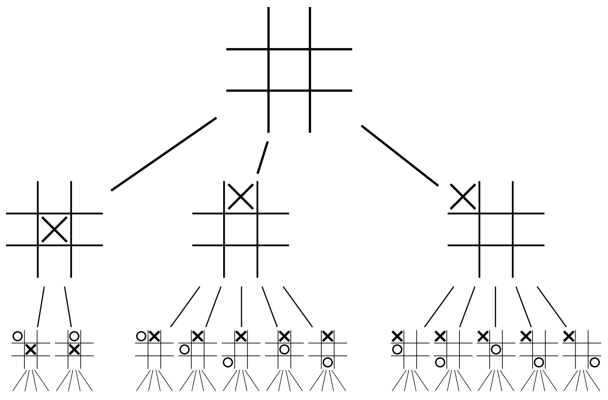


Figure 1.5: Simple Tic-tac-toe game tree, source: Wikipedia

1.3 Reversi game overview

Reversi (also know as Othello or Iago) is a game involving abstract strategy and played by two players on a board with 8 rows and 8 columns and a set of distinct pieces for each side. The pieces are typically dark for one the players and light for the other. Each player takes turns adding one piece to the board in a location that converts at least one existing piece from the opponent's color to his color. The player's goal is to have a majority of their colored pieces showing at the end of the game, turning over as many of their opponent's pieces as possible. In Reversi, the pieces on the board do not *move* as they do in Checkers or Chess, nor are they removed. Once a piece is placed on the board it can only be flipped from one player's color to the other.

Players take alternate turns. If one player cannot make a valid move, the play is passed back to the other player. When neither player can move, the game ends. This occurs when the grid has filled up or when neither player can legally place a piece in any of the remaining squares. This means the game ends before the grid is completely filled. This possibility may occur because one player has no pieces remaining on the board in his or her color. In over-the-board play this is generally scored as if the board was full (64-0). The initial state consists of 2 black pieces and 2 white pieces, at each step one player may add one piece at a time, thus the maximal number of steps equals 60. For more detailed rules' description please refer to [19]. Figure 1.6 shows the 8x8 board in the initial state (A) and an example of a final state (B). In this example, the game ended because no move is possible, all the fields are occupied. The black player has 60 points, the white player has 4 points, therefore the black player wins 60-4 and the score difference equals 56. Figure

1.7 presents the initial state and some following ones.

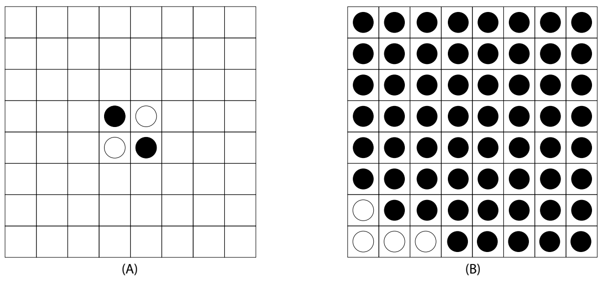


Figure 1.6: Initial state (A), Example final state (B)

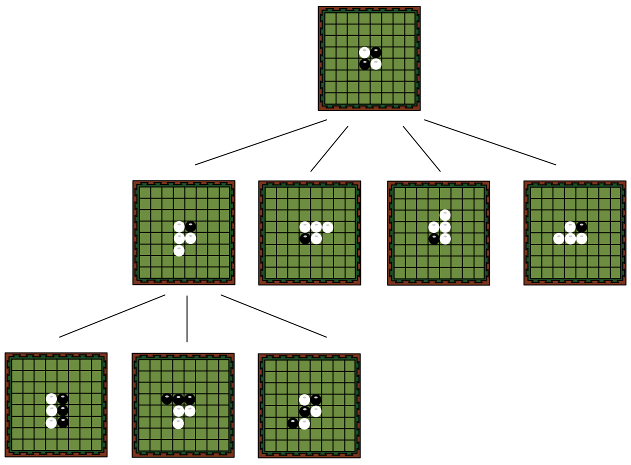


Figure 1.7: Reversi and its tree representation

1.4 Samegame puzzle overview

SameGame is a computer puzzle game featuring tile removal originally released under the name Chain Shot! in 1985 by Kuniaki Moribe (Morisuke). SameGame is played on a rectangular field, typically initially filled with four or five kinds of blocks placed randomly. By selecting a group of adjoining blocks of the same color, a player may remove them from the screen. Blocks that are no longer supported will slide down, and a column without any blocks will be trimmed away by other columns always sliding to one side (often the left). The goal of the game is to remove as many blocks from the playing field as possible. The game is over if no more blocks can be removed. This happens when either the player (1) has removed all blocks or (2) is left with a position where no adjacent blocks have the same color. In the 1st case, 1,000 bonus points are rewarded. In the second case, points will be deducted. The formula for deducting is similar to the formula for awarding points and it is iteratively applied for each color group left on the board. Here it is assumed that all blocks of the same color are connected. There are variations that differ in board size and the number of colors, but the 15 x 15 variant with 5 colors is the accepted standard and this is the case I will investigate. SameGame has be proven to be NP-complete[22]. For more detailed description of the rules, please refer to [23].

1.5 Complexity

The difficulties in human lookahead have made games attractive to programmers. Analysts have estimated the number of legal positions in Othello is at most 10^{28}, and it has a game-tree complexity of approximately 10^{58}[20][21]. I have calculated that the average branching factor equals 8.7 (Figure 1.8), and it is not uniform. The average branching factor is the average number of children for each node. The branching factor is high in the middle of the game(Figure 1.8 (A)) and it decreases as the game progresses. It has been shown that the problem is PSPACE-complete[21][30]. *'Any problem that is log-complete in PSPACE is also NP-hard, and NP-hardness alone constitutes evidence of intractability which is, for practical purposes, just about as strong as completeness in PSPACE'*[24]. The problems that are PSPACE-complete can be considered as the hardest problems in PSPACE, because any solution to such a problem could easily be used to solve any other problem in PSPACE. Generalized to an NxN board with an arbitrary initial configuration, at most $N^2 - 4$ moves can be made[31]. A similar game called 'Checkers' is shown to be EXPTIME-complete[25] (like 'Go' and Chess'). These other games are EXPTIME-complete because a game between two perfect players can be very long, so they are unlikely to be in PSPACE. But they will become PSPACE-complete if a polynomial bound on the number of moves is enforced. The definition of PSPACE-completeness is based on asymptotic complexity: the time it takes to solve a problem of size N, in the limit as N grows without bound. That means a game like checkers (which is played on an $8x8$ board) could never be PSPACE-complete (in fact, they can be solved in constant time and space using a very large lookup table). That is why all the games were modified by playing them on an NxN board instead. Table 1.1 shows complexities of some known games[32][33][34]. Combinatorial game theory has several ways of measuring game complexity. Here it presents it in regard to two measures:

- The state-space complexity: is the number of legal game positions reachable from the initial position of the game.
- The game tree size: is the total number of possible games that can be played: it's the number of leaf nodes in the game tree rooted at the game's initial position.

The game tree is typically much larger than the state space because the same positions can occur in many games by making moves in a different order (for example, in a tic-tac-toe game with two X and one O on the board, this position could have been reached in two different ways depending on where the first X was placed). An upper bound for the size of the game tree can sometimes be computed by simplifying the game in a way that only increases the size of the game tree (for example, by allowing illegal moves) until it becomes tractable.

Table 1.1: Complexities of some well-known games

Game	Board size (cells)	State-space complexity	Game-tree complexity	Average game length (plies)	Complexity class of suitable generalized game
Tic-tac-toe	9	10^3	10^5	9	PSPACE-complete
Checkers 8x8	32	10^{20}	10^{31}	70	EXPTIME-complete
Reversi	64	10^{28}	10^{58}	58	PSPACE-complete
Chess	64	10^{50}	10^{123}	80	EXPTIME-complete
Shogi	81	10^{71}	10^{226}	110	EXPTIME-complete
Go	361	10^{171}	10^{360}	150	EXPTIME-complete

Table 1.2: Complexity of SameGame[36]

Puzzle	Size	State-space complexity	Game-tree complexity	Average game length (plies)	Complexity class of suitable generalized game
SameGame	15x15	10^{159}	10^{85}	64.4	NP-complete

Similarly in [36] it has been presented that SameGame has an average branching factor of 20.7 and length of 64.4 (for a 15 x 15 version) which

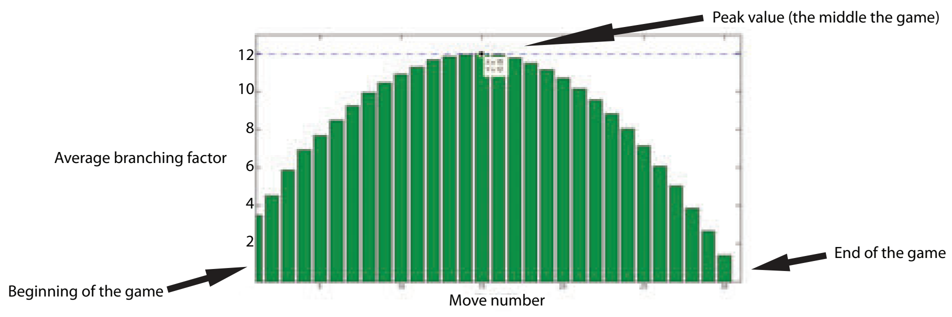


Figure 1.8: Complexity of Reversi, average branching factor throughout the game

makes it a more difficult problem than Reversi. Different board sizes will definitely produce different complexities, large boards would make the problem even more difficult. As it is presented, many two-player games are PSPACE-complete. A PSPACE-completeness result has two consequences. First, being in PSPACE means that the game can be played optimally, and typically all positions can be enumerated, using possibly exponential time but only polynomial space[31]. On the other hand, loopy two-player games are often EXPTIME-complete. Such a result is one of the few types of true lower bounds in complexity theory, implying that all algorithms require, in the worst case exponential time. Many puzzles (one-player games) have short solutions and are NP-complete. However, several puzzles based on motion-planning problems are harder, although often being in a bounded region, only PSPACE-complete. For further reading please refer to [26][28][29][35].

1.6 Typical algorithms

2-player games. Minimax[2] (sometimes minmax) is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss while maximizing the potential gain. Alternatively, it can be thought of as maximizing the minimum gain (maximin). Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision making in the presence of uncertainty. The algorithm can be thought of as exploring the nodes of a game tree. The effective branching factor of the tree is the average number of children of each node (i.e., the average number of legal moves in a position). The number of nodes to be explored usually increases exponentially with the number of plies (it is less than exponential if

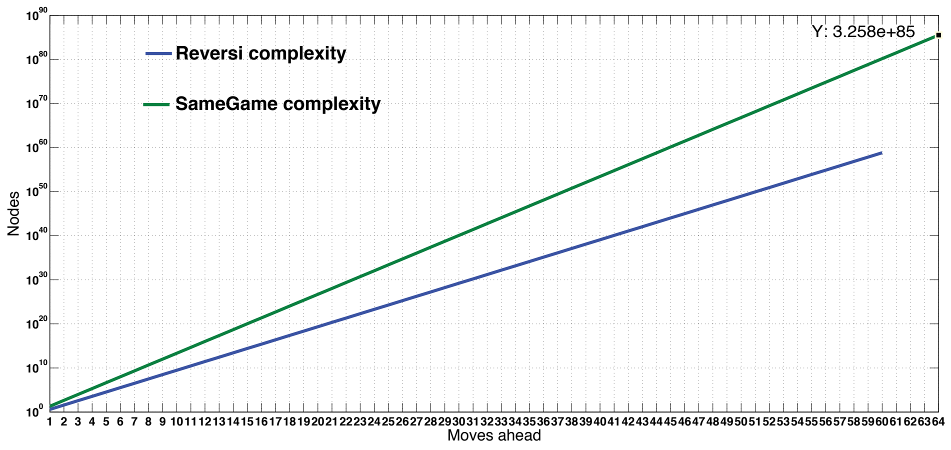


Figure 1.9: Comparison of Reversi and SameGame complexities

evaluating forced moves or repeated positions). The number of nodes to be explored for the analysis of a game is therefore approximately the branching factor raised to the power of the number of plies. It is therefore impractical to completely analyze games such as chess using the minimax algorithm.

The performance of the simple minimax algorithm may be improved dramatically, without affecting the result, by the use of alpha-beta pruning. Other heuristic pruning methods can also be used, but not all of them are guaranteed to give the same result as the un-pruned search. But still even for Reversi, solving the problem completely is impossible in reasonable amount of time.

1-player games (puzzles). The classical approach to puzzles involves techniques such as A*[37] and IDA*[38]. A* is a best-first search where all nodes have to be stored in a list. The list is sorted by an admissible evaluation function. IDA* is an iterative deepening variant of A* search. It uses a depth-first approach in such a way that there is no need to store the complete tree in memory. Both methods are heavily dependent on the quality of the evaluation function. Even if the function is an admissible under-estimator, it still has to give an accurate estimation. For an instance, these classical methods fail for SameGame because it is not easy to make an admissible under-estimator that still gives an accurate estimation. Therefore, there are problems which cannot be solved using classical methods.

A more universal approach. In both cases a priori domain specific knowledge can be used. In Reversi or Go best known agents use patterns and opening games to decrease the state-space size. It is not always possible to know something about a problem and it makes a certain algorithm less universal. Also in cases of some problems, such as Go, it is very hard to estimate an intermediate state of game, which makes the minimax algorithm with alpha-beta pruning quite useless. Therefore, I think that it is important to find an algorithm which can solve any kind of problem without any domain knowledge, treating it as an abstract problem. The thesis addresses this case, I am trying to solve the problem without such kind of a priori knowledge or database and when it is crucial to search big state-space very quickly. That is why the Monte Carlo Tree Search algorithm is very important and useful.

1.7 Assumptions, requirements and limitations

1.7.1 GPU

The implementation of these games is done using NVDIA Tesla GPUs and CUDA programming platform[27]. I also used the TSUBAME 2.0 supercomputer, where one node is equipped with a NVIDIA TESLA C2050 GPUs and Intel Xeon X5670 CPUs. All the following matters in this section make the GPU programming very different from programming GPUs and a programmer has to be aware of the problems and limitations. Also please note, that in this thesis, two problems (Reversi and SameGame) are analyzed and the results may depends on one's implementation. Therefore, they may be slightly different depending on the problem, its characteristics, complexity and implementation.

1.7.1.1 The complexity of GPU architecture

One of the difficulties when programming GPU is its architecture which differs from the CPU's one.

The hardware structure overview and the execution scheme. Unlike current CPUs, GPU chips do not feature big cache memory areas. Most of the transistors are used for computation. A GPU chip comprises multiple processing elements called stream processors (SM) or multiprocessors (MP) - Figure 1.10 (Here denoted as Multicore). Each multiprocessor has several processing cores. Using the GPU from host for general purpose computation

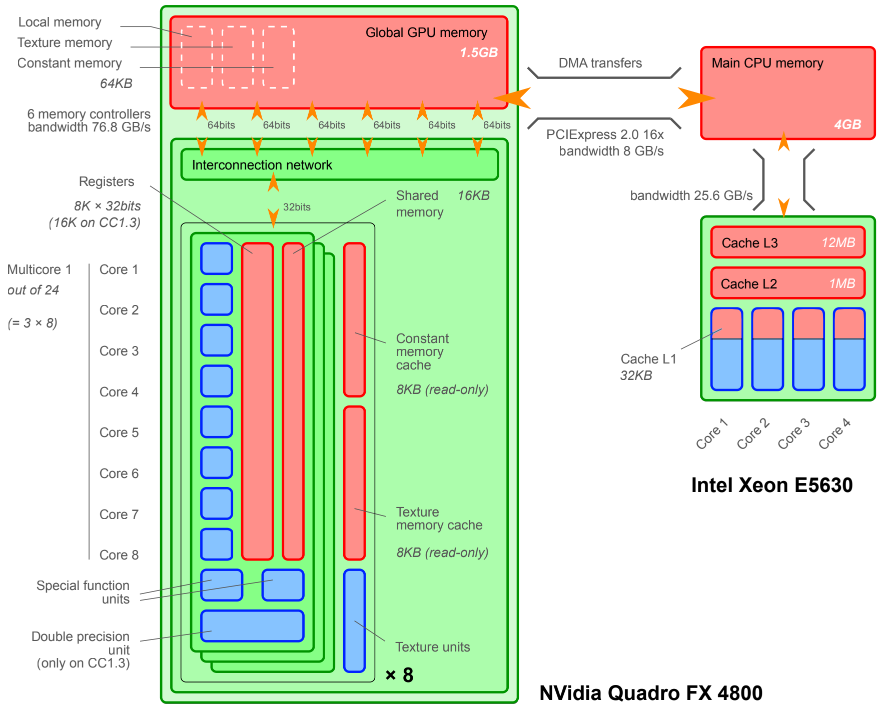


Figure 1.10: pre-Fermi GPU-CPU architecture[40]

is generally performed through three steps: data copy from host (CPU) to device (GPU) memory, execution on device, data copy from device to host. A GPU is connected to the host through a high-speed bus such as PCI-Express 16x. This bus is mainly used for DMA transfers between CPU and GPU memories since none of them is able to directly address the memory space of the other. When data and program has been loaded into GPU memory, execution can start since the number of stream multiprocessors and thread per SM is decided in advance. On the Tesla architecture, a SM can have 768 or 1024 threads simultaneously active according to the chip. The number of SM depends on the GPU: entry-level GPU has a few SM while high-end GPU has a lot of them. The GPU of the example has 24 multiprocessors therefore it can deal with 24576 active threads. Of course, all threads are not executed in the same time since a multiprocessor has only eight cores, so-called CUDA cores. Following the same example, 192 threads are executed at each clock cycle. In the worst case, 128 cycles are required to execute one instruction of all threads. The GPU can perform zero-cost hardware-based context switching and immediately switch to another thread to process using its scheduler. It is a very important thing to remember. We must manually take care of how blocks and threads are executed. It needs to be pre-configured by the programmer (Figures 1.15 and 1.14) show the hardware-software correspondence.

Memory hierarchy and its latency. A big difference between CPU and GPU is the memory hierarchy. GPU main (global) memory chips are actually found outside the GPU chip, therefore also called off-chip memory contrary to small local memory found inside SMs. Registers and shared memory are extremely fast while global memory is several hundred times slower. Each SM has a local high-speed internal memory. It can be seen as an L1 cache, but there are crucial differences: explicit instruction are required to move data from global memory to shared memory and there is no coherency among shared memories and global memory. When used as a cache memory, if the global memory changes, then shared memory's content will not be updated. Shared memory is considered as fast as register memory as long as there are no bank conflicts among threads. Global memory is linked to the GPU chip through a very large data path: up to 512-bits wide. Through a such bus width, sixteen consecutive 32-bits words can be fetched from global memory in a single cycle. In reality it is hard to achieve the peak bandwidth limit (over 100 GB/s). CUDA compiler allocates registers memory to threads. If the threads requires too many registers, local memory is used. Actually local memory does not exist in the hardware. Local memory is the name given to some global memory which was made private to a thread. This memory is extremely slow compared to register or shared memory, thus exceeding the maximum register memory lead to dramatically slow performances.

Table 1.3: Memory types in GPU [42]

Name	Addressable	Access	Cost (cycles)	Lifetime	Description
.reg	No	Read/Write	0	Thread	Registers
.sreg	No	Read-only	0	Thread	Special registers (platform-specific)
.const	Yes	Read-only	0[1]	Application	Constant memory
.global	Yes	Read/Write	>100	Application	Global memory
.local	Yes	Read/Write	>100	Thread	Local memory (private to each thread)
.param	Yes	Read-only	0		Parameters for a program
.shared	Yes	Read/Write	0	Block	Shared memory
.surf	Yes	Read/Write	>100	Application	Surface memory (global memory)
.tex	Yes	Read-only	>100	Application	Texture memory (global memory)

Both Tesla and Fermi architectures have several memory areas having different memory consistency models. CUDA toolkit is very system-centric (according to [43] definition) and memory management can quickly become a real nightmare for programmers. The communication between the threads is only possible within one block.

SIMD processing. Like processing units in hardware, threads are also divided into groups. The following scheme (Fig. 1.15) shows one grouping possibility. As for Grid and Block sizes, they are defined by developer at kernel execution. Warp size is fixed by hardware and is 32 for both Tesla and Fermi, but this size might change in future architectures according to NVIDIA CUDA documentation. Since the basic unit of execution flow in a multiprocessor is a warp of 32 threads, it is useless to execute less that 32 threads in a block. Warps execute in SIMD fashion. That brings the problem of warp divergence. A question about divergent threads rises from this execution strategy: What happens when threads do not execute the same code

[1]Cached, cost is amortized

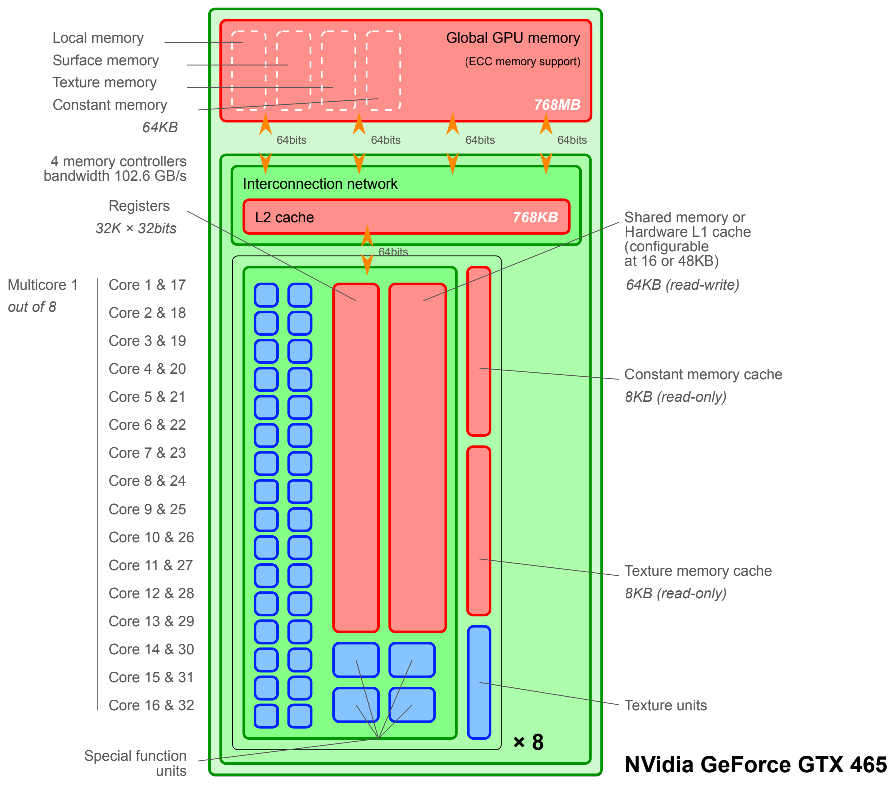


Figure 1.11: Fermi (2010-) GPU architecture, borrowed from [40]

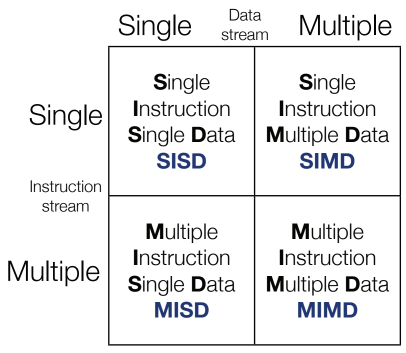

Single
Data stream
Multiple
Single
Instruction stream
Multiple
Single Instruction Single Data
SISD
Single Instruction Multiple Data
SIMD
Multiple Instruction Single Data
MISD
Multiple Instruction Multiple Data
MIMD

Figure 1.12: Flynn's Taxonomy (1966)

in a warp? On the Tesla architecture, each conditional branch is serialized. According to [41], ”else clause” is always executed first while other clauses are disabled, then ”if clause” is executed (and ”else clause” disabled). The problem of warp divergence is very important and conditional code pieces degrade the overall performance dramatically. It is one of the main problems which has to be tackled and be considered when implementing tree search. It is

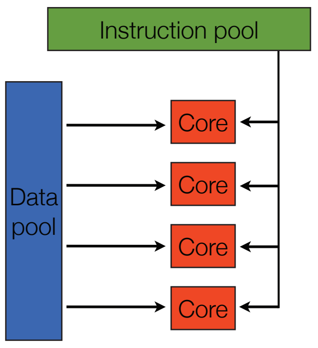


Figure 1.13: SIMD Processing

because not only the explicit if-else statements affect the parallelism but also loops where branching occurs and during tree search loop usage is unavoidable. Manual loop unrolling helps to improve the performance, but on the other hand, it cannot be done infinitely, since there are certain code length limitations.

1.7.1.2 Achieving high performance

Performance optimization revolves around three basic strategies[27]:

Maximize parallel execution to achieve maximum utilization. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called latency. In table 1.3 it is shown as cost. Each instruction takes 5 to 25 cycles to execute. The most common reason a warp is not ready to execute its next instruction is that the instruction's input operands are not yet available. As presented, global memory access takes hundreds of cycles, if some input operand resides in off-chip memory, the latency is 400 to 800 clock cycles. So higher level of parallelism is needed (more threads than the number of cores) to hide the latency. It is supported by the mentioned hardwired thread scheduler. In my case, the high instruction latency may result in shallow search depth.

To maximize utilization the application should be structured in a way that it exposes as much parallelism as possible and efficiently maps this parallelism to the various components of the system to keep them busy most of the time. GPU multiprocessor relies on thread-level parallelism to maximize utilization of its functional units. Utilization is therefore directly linked to the number of resident warps. At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction, if any, and issues the instruction to the active threads of the warp. Full utilization is achieved

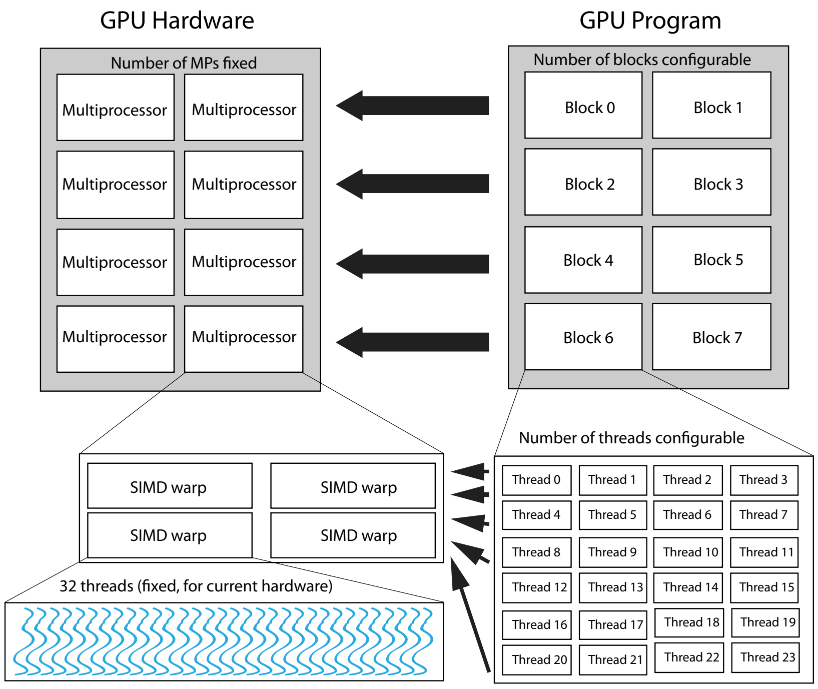


Figure 1.14: GPU programming scheme

when all warp schedulers always have some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when latency is completely *hidden*. The number of instructions required to hide a latency depends on the respective throughputs of these instructions.

The number of blocks and warps residing on each multiprocessor for a given kernel call depends on the execution configuration of the call. It should be decided by the programmer how many threads to run.

Optimize memory usage to achieve maximum memory throughput. The first step in maximizing overall memory throughput for the application is to minimize data transfers with low bandwidth. That also means minimizing

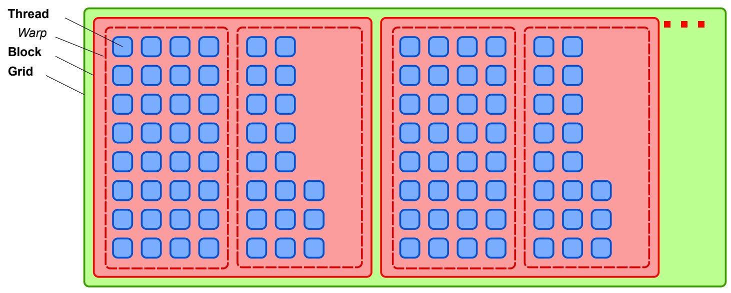


Figure 1.15: Software mapping - Example of 102 threads divided into 2 blocks, borrowed from [40]

data transfers between global memory and the device by maximizing use of on-chip memory: shared memory and caches (i.e. L1/L2 caches available on devices of compute capability 2.x, texture cache and constant cache available on all devices). Shared memory is equivalent to a user-managed cache. Applications should strive to minimize data transfer between the host and the device. One way to accomplish this is to move more code from the host to the device, even if that means running kernels with low parallelism computations. Intermediate data structures may be created in device memory, operated on by the device, and destroyed without ever being mapped by the host or copied to host memory. In real programs, the peak bandwidth is difficult to obtain since memory has to be aligned and threads accesses have to be coalesced. Coalescing allows merging all independent thread memory access into one big access. Several thread access patterns are recognized by coalescing hardware. It also means there is severe bandwidth degradation for stridden accesses. Also, because of the overhead associated with each transfer, batching many small transfers into a single large transfer always performs better than making each transfer separately. Because it is on-chip, the shared memory space is much faster than the local and global memory spaces. In fact, for all threads of a warp, accessing shared memory is fast as long as there are no bank conflicts between the threads, as detailed below. To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module. Typically, first the data should be loaded into the global memory from the CPU memory space, later copied to the shared memory. After that the computation part should appear. When it is finished, the data should be copied back from the shared memory to the global

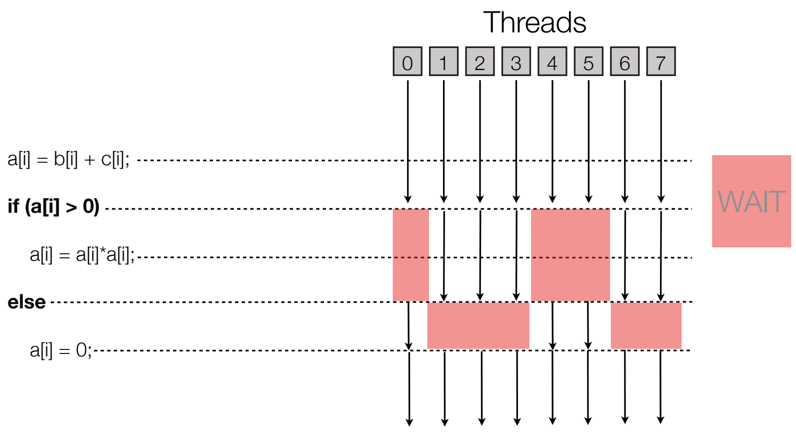


Figure 1.16: SIMD Processing - a problem with conditional code parts

memory and the host memory space. It is the right scheme to achieve good performance, though, it is not always possible due to the shared memory size limitations.

Optimize instruction usage to achieve maximum instruction throughput. To maximize instruction throughput the application should:

- Minimize the use of arithmetic instructions with low throughput. This includes trading precision for speed, single-precision instead of double-precision, or flushing denormalized numbers to zero.
- Minimize divergent warps caused by control flow instructions. Any flow control instruction (if, switch, do, for, while) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge (i.e. to follow different execution paths). If this happens, the different executions paths have to be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths are completed, the threads converge back to the same execution path. To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps.
- Reduce the number of instructions

Limitations. Blocks cannot communicate as MPI processes. Every block must have the same number of threads. The numbers of threads and blocks simultaneously running on an MP depends on several factors: limit of number of blocks (8), limit of number of warps, register usage, and shared memory usage. There are numerous difficulties regarding programming and debugging. The drivers, hardware and the software (CUDA) change very often. It requires constant adaptation and being aware of that. For the details please refer to the next subsection - CUDA. CUDA has certain limitations in regard to recursive calls, loop processing and the code length. Threads should be running in groups of at least 32 *(warps)* for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution model becomes a significant limitation for any inherently divergent task.

1.7.1.3 CUDA.

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia. CUDA is the computing engine in Nvidia graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest Nvidia GPUs become accessible for computation like CPUs. Unlike CPUs however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general purpose problems on GPUs is known as GPGPU.

There are certain distinguishing features of GPU programming, mostly limiting when compared to the CPU programming style[27].

Advantages of the hardware/software model when using CUDA.

- Scattered reads - code can read from arbitrary addresses in memory.
- Shared memory - CUDA exposes a fast shared memory region (up to 48KB per Multi-Processor) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is it possible when using texture lookups.
- Relatively fast downloads and readbacks to and from the GPU

Limitations of the hardware/software model when using CUDA.

- Unlike OpenCL, CUDA-enabled GPUs are only available from Nvidia

- Valid C/C++ may sometimes be flagged and prevent compilation due to optimization techniques the compiler is required to employ to use limited resources
- CUDA (with compute capability 1.x) uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments
- Copying between host and device memory may incur a performance hit due to system bus bandwidth and latency
- Maximum number of instructions per kernel is 2 million.

The current generation CUDA architecture (codename: ”Fermi”) which is standard on Nvidia's released GeForce 400 Series GPU is designed from the ground up to natively support more programming languages such as C++. It has eight times the peak double-precision floating-point performance compared to Nvidia's previous-generation Tesla GPU. It also introduced several new features[39] including:

- Up to 512 CUDA cores and 3.0 billion transistors
- ECC memory support
- Unified Virtual Addressing for CPU and GPU
- Thrust - a C++ standard library functions and classes implemented for CUDA
- CUDA C++ debugger
- C++ Virtual functions
- NPP - Nvidia Primitives Library
- The previous shared memory space became a configurable scratchpad/L1 cache. Fermi architecture has two configuration option: 48 KB of shared memory and 16 KB of L1 cache or 16 KB of shared memory and 48 KB of cache.
- The new architecture upgrades memory from GDDR3 to GDDR5. Despite its narrower memory bus width (256 vs 384-bit), the card of the figure has a theoretical memory bandwidth of 102.6 GB/s.

- Fermi architecture improves the issue of warp divergence because each multiprocessor features a dual warp scheduler. Each group of 16 cores can execute a different conditional branch. Our divergent thread example would be executed in parallel on Fermi architecture only. It works for half-warps only.

At the time of performing this research the newest CUDA 4.0 was not available for usage.

1.7.2 MPI

The ultimate goal is to find an algorithm to be used on modern supercomputers, which typically use the Message Passing Interface (MPI)[44] to provide inter-node communication. Message Passing Interface (MPI) is an API specification that allows processes to communicate with one another by sending and receiving messages. Besides many other applications, it is a de facto standard for parallel programs running on computer clusters and supercomputers, where the cost of accessing non-local memory is high.

Features:

- Processes have separate address spaces
- Processes communicate via sending and receiving messages
- MPI is designed mainly for SPMD/MIMD (or distributed memory parallel supercomputer)
- Each process is run on a separate node
- Communication is over high-performance switch
- MPI can support shared memory programming model

1.7.3 Problem with classic tree search on GPUs

CUDA does not allow recursion, therefore the GPU tree search implementation must rely on a modified iterative version of the algorithm [12]. Since GPU is SIMD hardware, each thread of every warp performs exactly the same operation at any given moment. Whenever a conditional operation within one warp is present, *warp divergence* occurs. Figure1.17 shows an example of such instructions (*break*). Each root node processed in a parallel manner represents different board state, therefore every searched tree may have a different structure. Iterative tree searching algorithm is based on one main outer loop and 2 inner loops (traversing up and down). Depending on the number of children for each node, the moment of the loop break occurrence varies, and therefore causes the divergence in each warp. The ideal case, when no divergence is present, was estimated by calculating average time of searching the same tree (starting with the same root node) by each thread. The result proved that reducing divergence, could significantly improve the search performance (reduce the time to find the solution needed). Different trees' structure forces some SIMD threads to wait for other threads which quit a loop (break instruction). Some threads quit the loop, some do not. Thread divergence occurs (Fig. 1.17). Figure 1.18 shows the effect, iteration numbers are marked. Here for

```
Outer Loop {

  Inner loop 1 {

    Go downwards();
    if (not possible to go
    downwards) break;
    .......
  }

  Inner loop 2 {


    Go upwards();
    if (possible to go downwards)
    break;
    ......
  }
}
```

Figure 1.17: Non-recursive generic tree traversal

2 given threads with 10 nodes each, the total time needed to complete the task extends to 15 iterations (assuming that one iteration corresponds to one node look-up). The reason for such a phenomenon is that each thread in a warp has to wait within an inner loop (as shown in Figure 1.17) until other threads have finished. Parallel node pruning (i.e. $\alpha\beta$) algorithm might be implemented as it is suggested in [9][50][49][48][47], however the batch manner of GPU processing implies serious difficulties in synchronizing the jobs processed in parallel. Moreover, the inter-thread communication is very limited (only within the same block). In some papers, SIMD $\alpha\beta$ implementations are presented, but they differ from this case. I.e. in [48] no particular problem is being solved, moreover the tree is synthetic with constant branching factor.

The second method of minimizing the warp divergence problem might be a subject to an automatic tuning, which should be considered as the manual tuning is very hard and time consuming. Considering the results obtained,

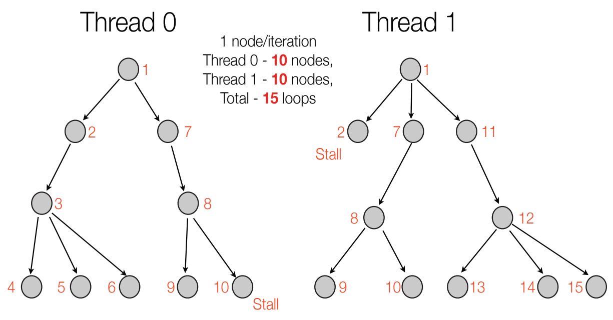


Figure 1.18: Warp divergence: different tree structures

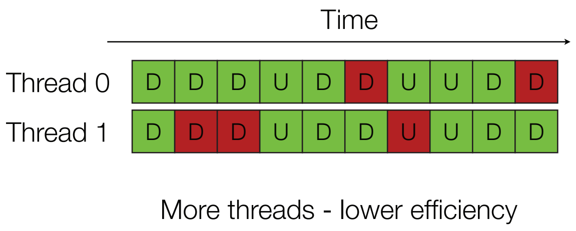


Figure 1.19: Impact on performance of warp divergence: green = instruction executed, red = wait

GPU as a group of SIMD processors performs well in the task of tree searching, where branching factor is constant. Otherwise, warp divergence becomes a serious problem, and should be minimized. Either by parallelizing the parts of algorithm that are processed in a SIMD way or by dividing main task into several smaller subtasks [46][11]. Direct implementation of the algorithm did not produce any significant improvement over the CPU sequential execution or effected in even worse performance. Direct implementation means just the most basic iterative minimax CUDA implementation. Given all this limitations and obstacles I conclude that this algorithm is hard to implement on GPU efficiently[15] and therefore I focus on MCTS GPU implementation instead.

Chapter 2

Related works

2.1 Classic approaches

One traditional AI technique for creating game playing software is to use a minimax tree search[2]. This involves playing out all hypothetical moves on the board up to a certain point, then using an evaluation function to estimate the value of that position for the current player. The move which leads to the best hypothetical board is selected, and the process is repeated each turn. While tree searches have been very effective in computer chess, they have seen less success in Computer Go programs. This is partly because it has traditionally been difficult to create an effective evaluation function for a Go board, and partly because the large number of possible moves each side can make each leads to a high branching factor. This makes this technique very computationally expensive. Because of this, many programs which use search trees extensively can only play on the smaller 9x9 board, rather than full 19x19 ones.

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[4][5] and Shared Hash Table[3].

SMP algorithms are classified by their scalability (trend in search speed as the number of processors becomes large) and their speedup (change in time to complete a search). Typically, programmers use scaling to mean change in nodes per second(NPS) rates, and speedup to mean change in time to depth. Scaling and scalability are thus two different concepts.

A Shared Hash Table is a Hash table or Transposition table which is accessed by various processes or threads simultaneously, running on multiple processors or processor cores. Shared hash tables are most often implemented as dynamically allocated memory treated as global array. Due to memory protection between processes, they require an Application programming interface provided by the operating system to allocate shared memory. Threads may share global memory from the process they are belonging to.

This technique is a very simple approach to SMP. The implementation requires little more than starting additional processors. Processors are simply fed the root position at the beginning of the search, and each searches the same tree with the only communication being the transposition table. The gains come from the effect of nondeterminism. Each processor will finish the various subtrees in varying amounts of time, and as the search continues, these effects grow making the search trees diverge. The speedup is then based on how many nodes the main processor is able to skip from transposition table entries.

Many programs use this if a 'quick and dirty' approach to SMP is needed. Unfortunately, this approach gives little speedup on a mere 2 processors, and scales quite badly after this. However, the NPS scaling is nearly perfect.

These algorithms divide the Alpha-Beta[12] tree, giving different subtrees to different processors. Because alpha-beta is a serial algorithm, this approach is much more complex. However, these techniques also provide for the greatest gains from additional processors.

The idea in Feldmann's Young Brothers Wait (YBW) as well in Kuszmaul's Jamboree Search[6][7][8], is to search the first sibling node first before spawning the remaining siblings in parallel. This is based on the observations that the first move is either going to produce a cutoff (in which case processing sibling nodes is wasted effort) or return much better bounds. If the first move does not produce a cut-off, then the remaining moves are searched in parallel. This process is recursive.

Many different approaches have been tried that do not directly split the search tree. These algorithms have not enjoyed popular success due to the fact that they are not scalable. Typical examples include one processor that evaluates positions fed to it by a searching processor, or a tactical search that confirms or refutes a positional search. The algorithm called DTS (Dynamic Tree Splitting)[10], invented by the Cray Blitz team (including Robert Hyatt), is the most complex, but gave the best results with speedup of around 11x for 16 threads. Though this gives the best known scalability for any SMP algorithm, there are very few programs using it because of its difficulty of implementation. In 2001 in [9] it is shown that a speedup of approximately 11 for DTS and 4 previous parallel alpha-beta can be achieved using 16 threads. R. Feldman[5] shows that a speedup for hundreds of processors can be achieved using his algorithm in regard to dynamic load distribution, choice of suitable processor network, implementation and distribution strategy. The mentioned algorithms are hard to implement even using parallel CPUs with shared memory, therefore GPU implementation may be very hard or ineffective. Regarding SIMD processing Hopp and Sanders[11] present a solution. The key point is to break down the complex MIMD algorithm into smaller pieces to avoid complicated control scheme and multiple inner loops which cause thread stalls. If such an algorithm can be transformed into a form where only one outer loop is present, then threads can remain more busy, since they don't have to wait for other ones within the inner loops.

2.2 Monte Carlo Tree Search

One major alternative to using hand-coded knowledge and searches is the use of Monte-Carlo methods. This is done by generating a list of potential moves, and for each move playing out thousands of games at random on the resulting board. The move which leads to the best set of random games for the current player is chosen as the best move. The advantage of this technique is that it requires very little domain knowledge or expert input, the trade-off being increased memory and processor requirements. In 2006 Remi Coulomb[16] and other researchers combined the ideas of classic tree search and Monte Carlo methods to provide a new approach to move planning in computer Go now known as MCTS. Kocsis and Szepesvari[17] formalised this approach into the UCT algorithm. It was the first algorithm to beat a human player in Go. Since that moment the algorithm has been evolving all the time and parallel schemes have been proposed[18]. Recent developments in Monte Carlo Tree Search and machine learning have brought the best programs to high dan level on the small 9x9 board. In 2009, the first such programs appeared which could reach and hold low dan-level ranks on the KGS Go Server also on the 19x19 board. Only a decade ago, very strong players were able to beat computer programs at handicaps of 25-30 stones, an enormous handicap that few human players would ever take. Nevertheless, the existing works test the parallel versions of this algorithm only for up to 512 cores. The open problem is the scalability of this approach for thousands or millions of threads.

No significant publications about efficient large scale GPU tree search are available at the moment of writing this thesis, however at the moment of finishing this work, in [13] Yoshizoe et al. parallelize MCTS in a distributed environment with a 700-fold speedup on 1200 cores. It utilizes Transposition table driven scheduling (TDS)[14] to efficiently share a tree among processors over the network. Therefore it might be an alternative for approaches presented in this thesis or a basis for GPU usage. Although the scalability of this algorithm has not yet been analyzed, it seems to be promising.

2.3 My contributions

To point out my research scope I define the following problems:

- How to utilize thousands or millions of GPU threads to perform an effective tree search?
- Can we implement the idea?
- How does the best solution found perform?
- What is the benefit of the GPU implementation?
- What are the limitations of analyzed algorithms?
- What kind of problems can be solved afterwards?

In this thesis I am presenting an efficient parallel GPU MCTS implementation based on the introduced *block-parallelism* scheme which combines GPU SIMD thread groups and performs independent searches without any need of intra-GPU or inter-GPU communication. I compare it with a simple leaf parallel scheme which implies certain performance limitations. The obtained results show that using my GPU MCTS implementation on the TSUBAME 2.0 system one GPU's performance can be compared to 50-100 CPU threads depending on factors such as the search time and other MCTS parameters. The block-parallel algorithm provides better results than the naive leaf-parallel scheme which fail to scale well beyond 1000 threads on a single GPU. The block-parallel algorithm is approximately 4 times more efficient in terms of the number of CPU threads' results comparable with the GPU implementation. In order not to generate random numbers on GPU I introduce an algorithm, where the numbers are transferred from the CPU for each GPU block accessible as a look-up table. This approach makes the time needed for random-sequence generation insignificantly small. Additionally I am proposing two improvements of the first GPU version of the algorithm to increase its strength. The first one is simultaneous CPU/GPU processing used to decrease the negative effect of GPU's latency. Another modification is applied to the decision step of the MCTS algorithm and utilizes big sample set size to calculate possible error based on the set's variance. In this thesis for the first time I discuss scalability of the algorithm for millions of threads. The program is designed in the way that it can be run on many nodes using Message Passing Interface (MPI) standard. As a method of evaluating my results I compared the results of multiple CPU cores and GPUs playing against the standard sequential CPU implementation. Therefore the algorithm's scalability is analyzed for multiple CPUs and GPUs. My results show that this algorithm implies almost no inter-node communication overhead and it scales linearly in terms of the number of simulation performed in a given time period. However, beyond a certain number of running threads, a lack of performance improvement was observed. This problem is analyzed and some reasons for this have been given by me.

Table 2.1: My contributions

Problem	Solution
GPU architecture/CUDA limitations	**Block parallelism**
	Fast random sequences
GPU Latency	**Simultaneous CPU-GPU processing**
Low efficient leaf parallel part of block parallelism	**Variance based error estimation for decision making**
Unknown scalability	**MPI based multi-CPU and multi-GPU parallel programs**
	SameGame implementation for scalability analysis

Chapter 3

Monte Carlo Tree Search

Contents

3.1 Introduction

Monte Carlo Tree Search (MCTS)[51][17] is a method for making optimal decisions in artificial intelligence (AI) problems, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search.

Research interest in MCTS has risen sharply due to its spectacular success with computer Go and its potential application to a number of other difficult problems. Its application extends beyond games, and MCTS can theoretically be applied to any domain that can be described in terms of state, action pairs and simulation used to forecast outcomes such as decision support, control, delayed reward problems or complex optimization[52][53][54][22]. The main advantages of the MCTS algorithm are that it does not require any strategic or tactical knowledge about the given domain to make reasonable decisions and algorithm can be halted at any time to return the current best estimate. Another advantage of this approach is that the longer the algorithm runs the better the solution and the time limit can be specified allowing to control the quality of the decisions made. This means that this algorithm guarantees getting a solution (although not necessarily the best one). It provides relatively good results in games like Go or Chess where standard algorithms fail. So far, previous research has shown that the algorithm can be parallelized on multiple CPUs. The importance of this research is that if the MCTS algorithm can be efficiently parallelized on GPU(s) it can also be applied to other similar problems on modern multi-CPU/GPU systems such as the TSUBAME 2.0 supercomputer.

In this chapter I am presenting an efficient parallel GPU MCTS implementation based on the introduced *block-parallelism* scheme which combines GPU SIMD thread groups and performs independent searches without any need of intra-GPU or inter-GPU communication. Then I compare it with a simple leaf parallel scheme which implies certain performance limitations. The obtained results show that using my GPU MCTS implementation on the TSUBAME 2.0 system one GPU's performance can be compared to 50-100 CPU threads depending on factors such as the search time and other MCTS parameters using root-parallelism. The block-parallel algorithm provides better results than the simple leaf-parallel scheme which fail to scale well beyond 1000 threads on a single GPU. The block-parallel algorithm is approximately 4 times more efficient in terms of the number of CPU threads needed to obtain results comparable with the GPU implementation.

3.2 MCTS algorithm

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 S_i. First the algorithm starts only with the root node. The general MCTS algorithm comprises 4 steps (Figure 3.1) which are repeated until a particular condition is met (i.e. no possible move or time limit is reached)

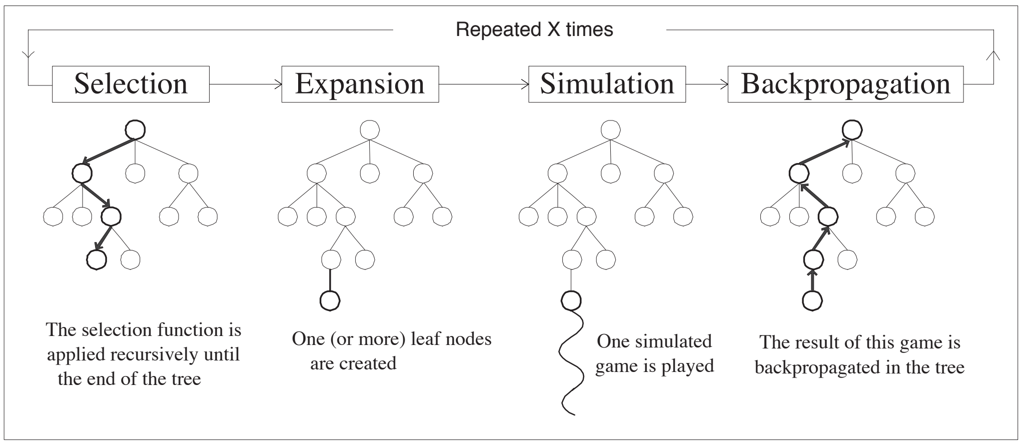

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until the end of the tree
One (or more) leaf nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 3.1: A single MCTS algorithm iteration's steps (from [18])

3.2.1 MCTS iteration steps

3.2.1.1 Selection

- a node from the game tree is chosen based on the following formula. The value of each node is calculated and the best one is selected. In this paper, the formula used to calculate the node value is the Upper Confidence bound applied to Trees (UCT)[17]. The node with the highest value is selected.

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

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

Supposed 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 (Figure 3.2). 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. The C parameter adjusts the exploitation/exploration ratio. In the described implementation the parameter C is fixed at the value 0.2.

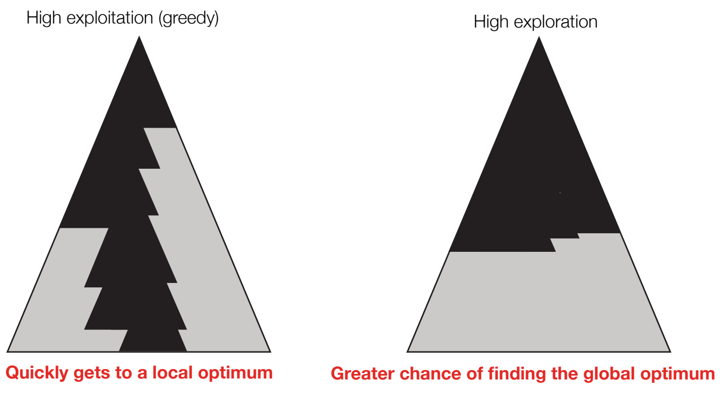

High exploitation (greedy)
High exploration
Quickly gets to a local optimum
Greater chance of finding the global optimum

Figure 3.2: Monte Carlo Tree Search - Exploitation/Exploration comparison

3.2.1.2 Expansion

One or more successors of the selected node are added to the tree depending on the strategy. This point is not strict, in my implementation I add one node per iteration, so this number can be different.

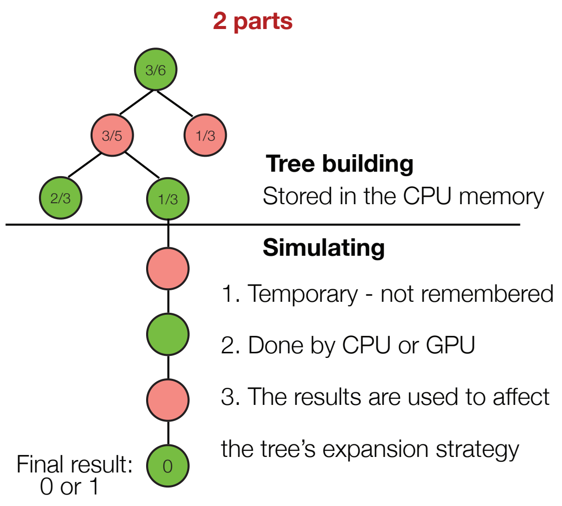


Figure 3.3: Monte Carlo Tree Search - 2 parts of the algorithm

3.2.1.3 Simulation

For each of the added node(s), simulation(s) are performed and the node(s) values (successes, total simulations) are updated. 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 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.

3.2.1.4 Backpropagation

It recursively 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. After the iterations have finished, the decision step occurs. A node with the

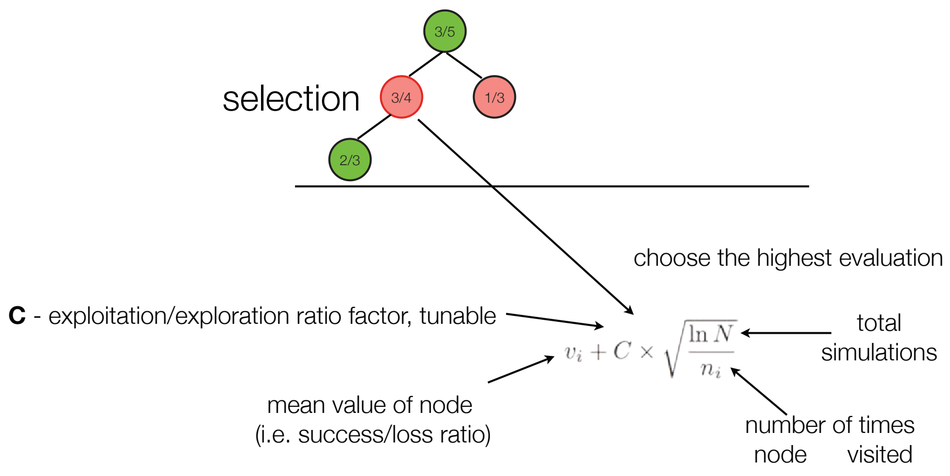

3/5
selection
3/4
1/3
2/3
choose the highest evaluation
C - exploitation/exploration ratio factor, tunable
$v_i + C \times \sqrt{\frac{\ln N}{n_i}}$
total simulations
mean value of node (i.e. success/loss ratio)
number of times node visited

Figure 3.4: Monte Carlo Tree Search - the selection phase

highest UCT value (typically an average - success/simulation ratio) is selected - Figure 3.5.

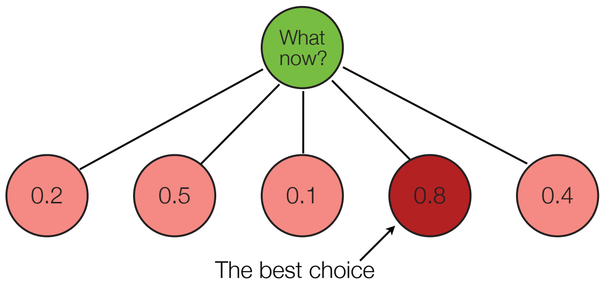
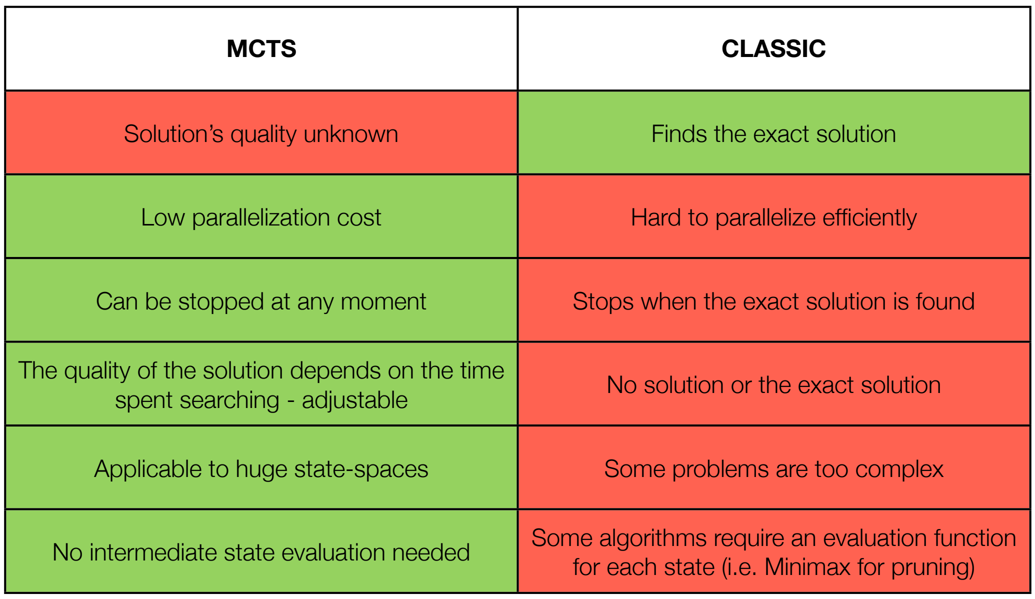


Figure 3.5: MCTS - decision step

3.3 MCTS vs classic tree search



MCTS	CLASSIC
Solution's quality unknown	Finds the exact solution
Low parallelization cost	Hard to parallelize efficiently
Can be stopped at any moment	Stops when the exact solution is found
The quality of the solution depends on the time spent searching - adjustable	No solution or the exact solution
Applicable to huge state-spaces	Some problems are too complex
No intermediate state evaluation needed	Some algorithms require an evaluation function for each state (i.e. Minimax for pruning)

Figure 3.6: MCTS vs classic tree search

Monte Carlo Tree Search has many advantages compared to classic, exhaustive search (Figure 3.6). The main is that no knowledge about the intermediate state is needed. It means that the only important fact is the final outcome. Therefore, this algorithm could be applied to problems like Go or Chess, where the intermediate state estimation is difficult. However the main disadvantage iof MCTS the uncertainty of the solution's quality.

3.4 Parallelization of Monte-Carlo Tree Search

3.4.1 Existing parallel approaches

In 2007 Cazenave and Jouandeau[55] proposed 2 methods of parallelization of MCTS and later in 2008 Chaslot et al.[18] proposed another one and analyzed 3 approaches (Figure 3.7):

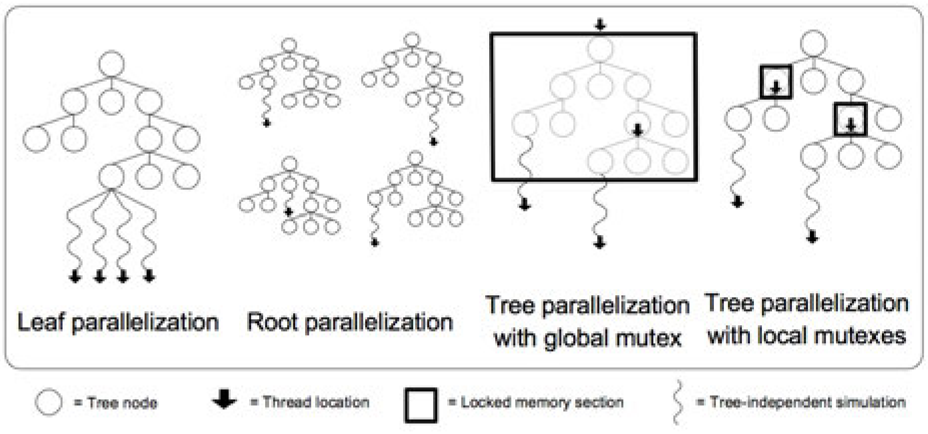

Leaf parallelization
Root parallelization
Tree parallelization with global mutex
Tree parallelization with local mutexes
= Tree node
= Thread location
= Locked memory section
= Tree-independent simulation

Figure 3.7: Parallel MCTS as in [18])

- **Leaf Parallelization** It is one of the easiest ways to parallelize MCTS. Only one thread traverses the tree and adds one or more nodes to the tree when a leaf node is reached (*Selection* and *Expansion* phases). Next, starting from the leaf node, independent simulated games are played for each available thread (*Simulation* phase). When all games are finished, the result of all these simulated games is propagated backwards through the tree by one single thread (*Backpropagation* phase). Leaf parallelization is presented in Figure 3.7a. Its implementation is easy and does not require any synchronization. However there are some problems. The time required for a simulated game is unpredictable. Playing n games using n different threads takes more time in average than playing one single game using one thread, since the program needs to wait for the longest simulated game. Another problem is the possibility of reaching a local maximum/minimum and being stuck there. Instead of searching different parts of a tree, the algorithm's computational power is lost. Leaf parallelization can be easily implemented inside an SMP environment, or on a cluster using MPI (Message Passing Interface) communication.

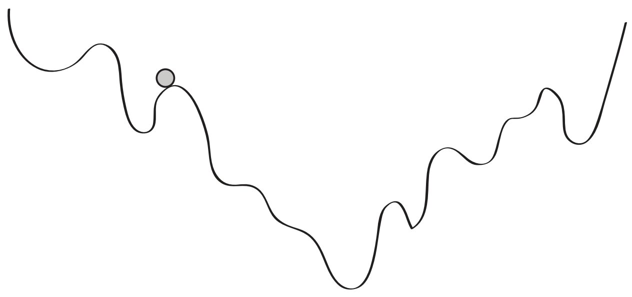


Figure 3.8: Leaf parallelism - seen as optimization using one starting point - likely to get stuck in a local minimum

- **Root Parallelization** Cazenave[55] proposed a second parallelization called *single-run* parallelization. It is also referred to as *root parallelization*[18]. The method works as follows. It consists of building multiple MCTS trees in parallel, with one thread per tree. Similar to *leaf parallelization*, the threads do not share information with each other. When the available time is spent, all the root children of the separate MCTS trees are merged with their corresponding clones. For each group of clones, the scores of all games played are added. The best move is selected based on this grand total. This parallelization method only requires a minimal amount of communication between threads, so the parallelization is easy, even on a cluster. Root parallelization is presented in Figure 3.7b.

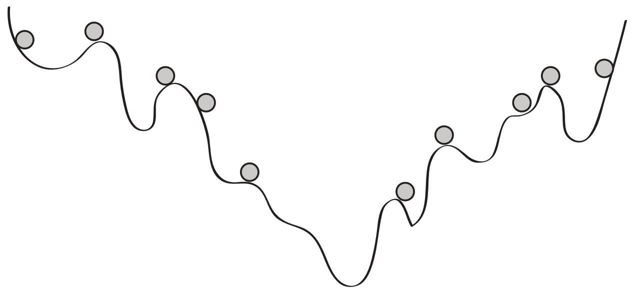


Figure 3.9: Root parallelism - seen as optimization using many starting points - greater chance of 'falling' into the global minimum)

- **Tree Parallelization** Chaslot et al. [18] proposed a method called *tree parallelization*. This method uses one shared tree from which several simultaneous games are played. Each thread can modify the information contained in the tree; therefore mutexes are used to lock from time to time certain parts of the tree to prevent data corruption. There are two methods to improve the performance of tree parallelization:
  - **1. Mutex location** Based on the location of the mutexes in the tree, we distinguish two mutex location methods: (1) using a global mutex and (2) using several local mutexes.
  - **2. Virtual loss** If several threads start from the root at the same time, it is possible that they traverse the tree for a large part in the same way. Simulated games might start from leaf nodes, which are in the neighborhood of each other. It can even happen that simulated games begin from the same leaf node. Because a search tree typically has millions of nodes, it may be redundant to explore a rather small part of the tree several times. Once a tread visits a node the value of this node will be decreased. The next thread will only select the same node if its value remains better than its siblings' values.

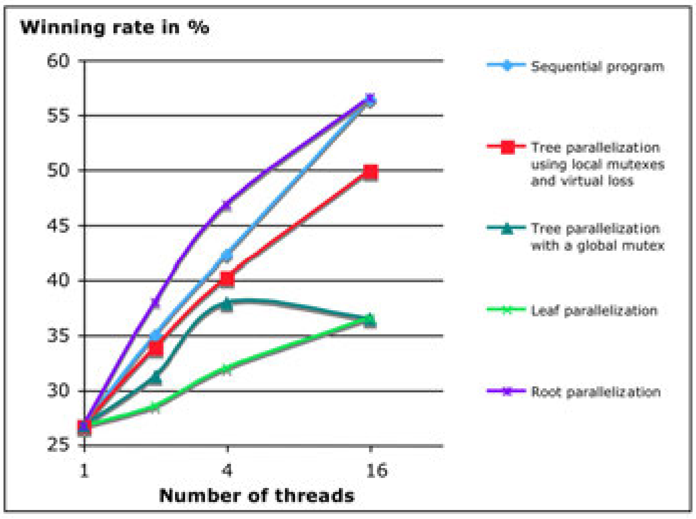


Figure 3.10: Performance of the different parallelization methods as in [18])

In this paper I will consider only the first 2 approaches as the limitations of the GPU make it difficult to implement an efficient communication between the running threads and as shown in [18] the *Root parallel algorithm* performs the best among those 3 (Figure 3.10).

3.4.2 Proposed algorithm: Block parallel MCTS for GPUs

This is the key part of this thesis. In the GPU implementation 2 approaches are considered and discussed. The first one (Figure 3.12a) 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 3.12c), is the proposed in this thesis block parallelization method. It comprises both leaf and root parallel schemes. Root parallelism (Figure 3.12b) is an efficient method of parallelization MCTS on CPUs.

(a) Leaf parallelism + (b) Root parallelism = **(c) Block parallelism**

It is more efficient than simple leaf parallelization[18], because building more trees diminishes the effect of being stuck in a local extremum and 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. In Figure 3.13 I present my analysis of such an approach. Given that a problem can have many local maximas, starting from one point and performing a search might not be very accurate. Then we should spend more time on the search to build a tree - that is the sequential MCTS approach, the most basic case. The second one, leaf parallelism should diminish this effect by having more samples from a given point while keeping the depth at the same level. In theory, it should make the search broader. 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. We can consider those trees as starting points in optimization. The last, my proposed algorithm, combines those two, so each search should be more accurate and less local at the same time.

The main advantage. Works well with SIMD hardware, improves the overall result on 2 levels of parallelization.

The main weakness. CPU sequential tree management part (proportional to the number of trees - Figures 3.14 and 3.16).

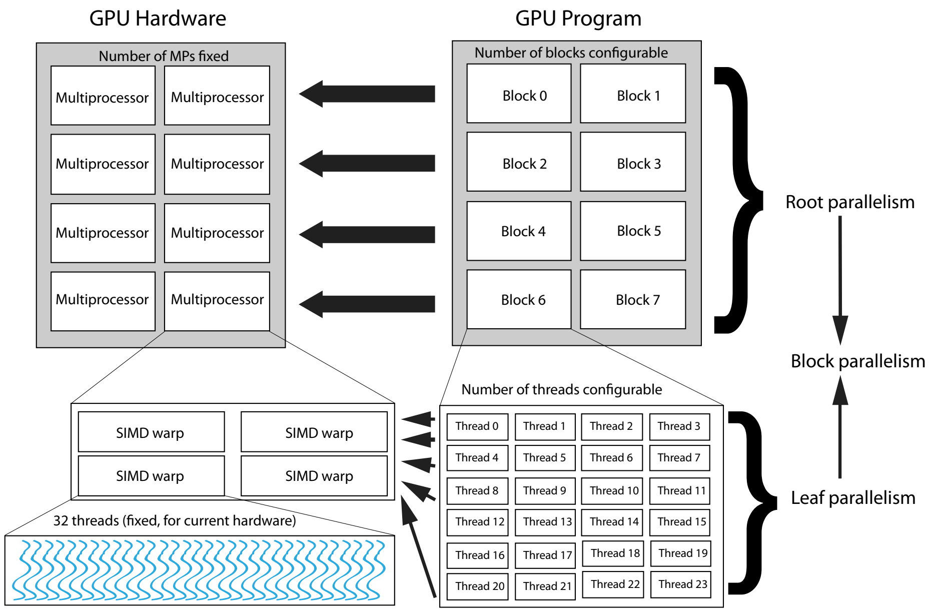


Figure 3.11: Correspondence of the algorithm to hardware.

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(Figure 3.11). 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, therefore the more blocks exist, the more time without actually simulating is spent. On the other hand the deeper the tree, the better the performance. Naturally, a compromise has to be established. In our experiments the smallest number of threads used is 32 which corresponds to the warp size. Figure 3.14 presents this computing scheme and explains why a small block size might decrease the performance when the total number of threads is large. Summarizing, 2 approaches are discussed regarding the GPU:

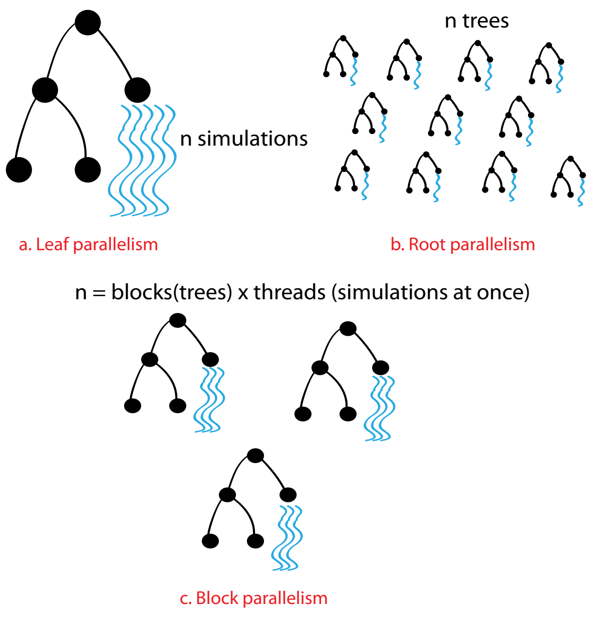


Figure 3.12: An illustration of considered schemes

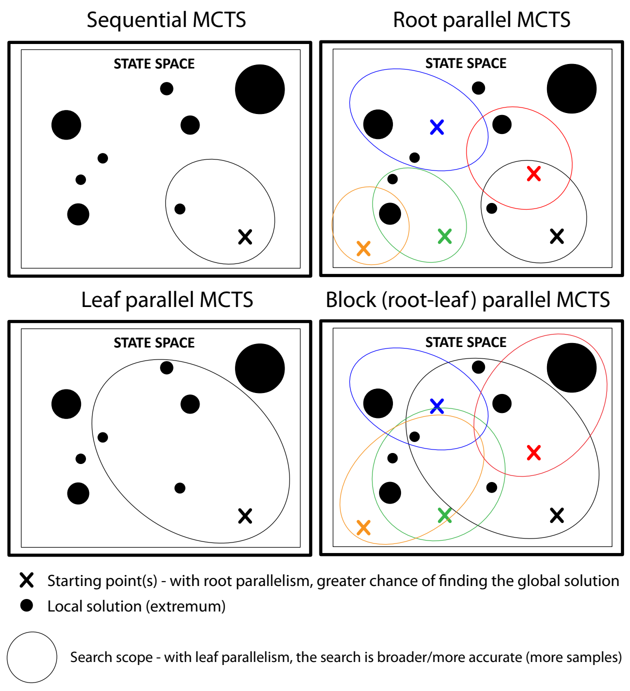

Sequential MCTS
Root parallel MCTS
STATE SPACE
Leaf parallel MCTS
Block (root-leaf) parallel MCTS
Starting point(s) - with root parallelism, greater chance of finding the global solution
Local solution (extremum)
Search scope - with leaf parallelism, the search is broader/more accurate (more samples)

Figure 3.13: Block parallelism's theoretical impact on the performance

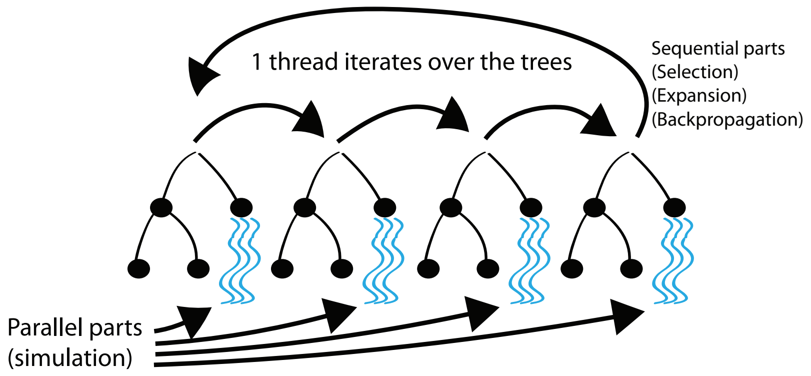


Figure 3.14: Block parallelism - sequential overhead

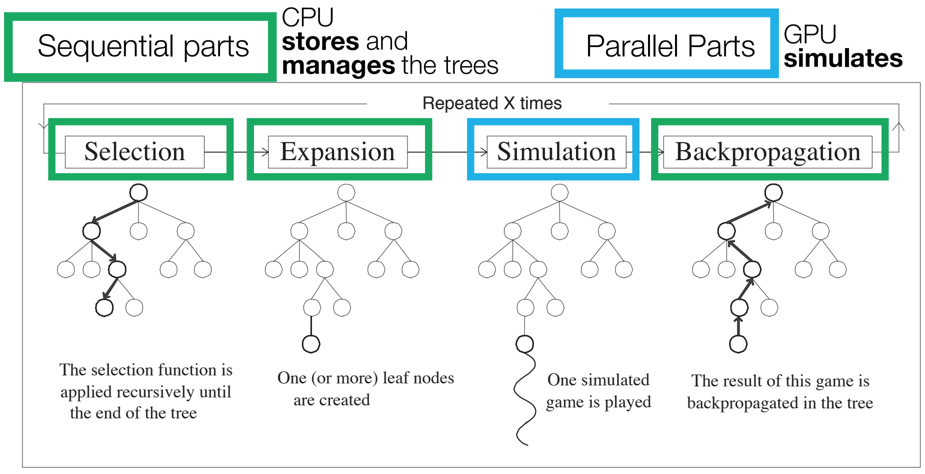

Sequential parts
CPU
stores and
manages the trees
Parallel Parts
GPU
simulates
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until the end of the tree
One (or more) leaf nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 3.15: Block parallelism - sequential overhead, 4 trees)

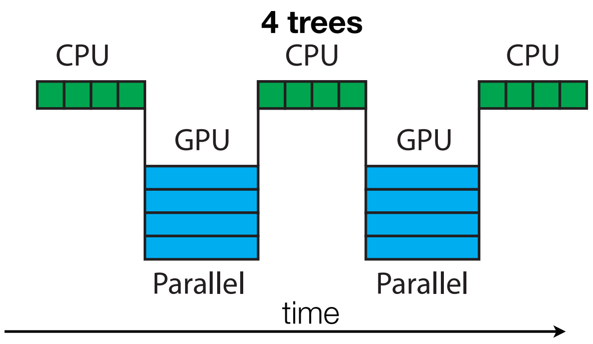


Figure 3.16: Block parallelism - sequential overhead (2)

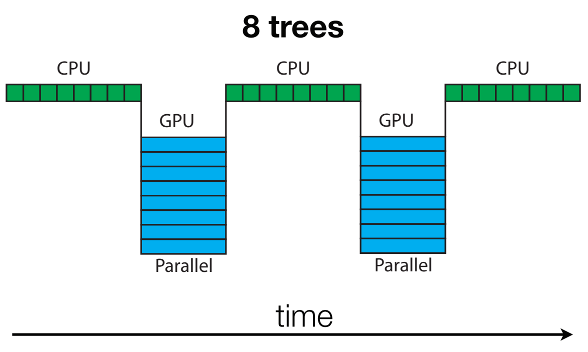


Figure 3.17: Block parallelism - sequential overhead, 8 trees, more trees take more CPU time)

- The simplest leaf parallelization
- The block parallelization method - the one comprising both leaf and root parallel schemes.

Some of the factors which might affect the overall MCTS performance.

- Search time - the longer the run, the deeper the tree. More nodes are expanded, therefore a better decision evaluation should be obtained.
- Parallelism - number of trees and/or increased simulation speed
- Exploitation/exploration ratio adjustment
- Domain knowledge - preprogrammed, fast/complex playouts
- Implementation dependent factors - data structures, algorithms

One of the goals of this thesis is to determine which parameters affect the scalability of the algorithm when running on multiple CPUs/GPUs.

3.5 MPI usage

In order to run the simulations on more machines the application has been modified in the way that communication through MPI is possible. This allows us to take advantage of systems such as the TSUBAME supercomputer or smaller heterogenous clusters. The implemented scheme (Figures 3.19 and 3.18) defines one master process (with id 0) which controls the game and

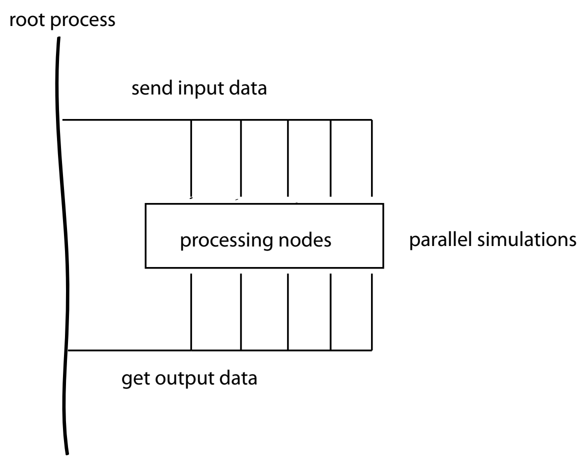


Figure 3.18: Parallel simulation scheme

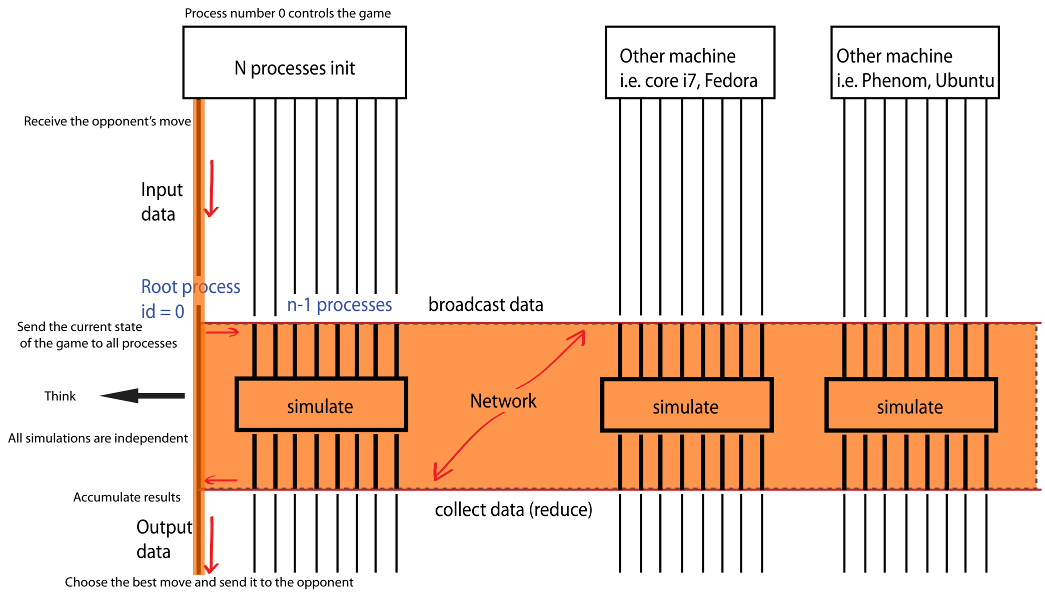


Figure 3.19: MPI Processing scheme

I/O operations, whereas other processes are active only during the MCTS phases. The master process broadcast the input data (current state/node) to the other processes. Then each process performs an independent Monte Carlo search and stores the result. After this phase the master process collects the data (through the *reduce* MPI operation) and sums the results. To ensure that all the processes end in the specified amount of time, the start time and time limit values are included in the input data. None of the processes can exceed the specified time limit. The other important aspect of the multi-node implementation is whether it should be based entirely on MPI even within one node. The possible approaches include MPI, MPI/OpenMP, MPI/pthreads schemes.

3.6 Hybrid CPU-GPU approach

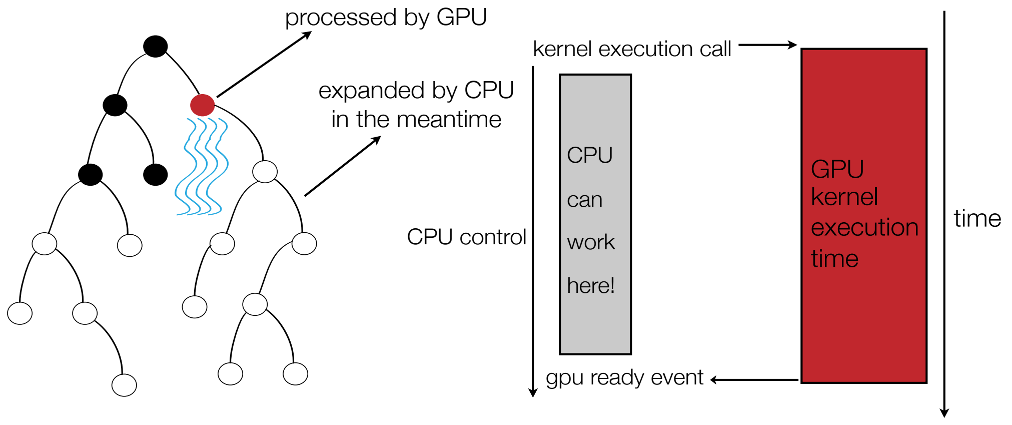


Figure 3.20: Hybrid CPU-GPU processing scheme

I observed that the trees formed by our algorithm using GPUs are not as deep as in the case of root parallelism implemented on CPU. This results from the time spent on each GPU kernel's 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 we experimented on using hybrid CPU-GPU algorithm (Figure 3.20). 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.

Pseudocode:

```
GPUkernelCall<<<>>>......

        while (cudaEventQuery(stop) == cudaErrorNotReady) {

                CPU expands the tree in the meanwhile

        }
```

3.7 Single player MCTS version (SP-MCTS)

In this section I am presenting a new idea of exploitation/exploration ratio selection strategy for problems where the final score limit is not known beforehand, as in the SameGame puzzle problem. It is based on estimating the upper confidence bound by calculation of the standard error of the mean. The motivation regarding SP-MCTS has been inspired by the the paper by Schadd et al.[22] about addressing NP-complete puzzles with MCTS (SP-MCTS). One of the problems they have experienced was that in the standard UCT formula it is assumed that the values fall into specific range for the algorithm to work properly. I am presenting an approach which changes this formula in the way that such a value's range tuning is not necessary and the search holds the main properties of following the best solution.

3.7.1 Original SP-MCTS

In that approach the selection formula has been modified from the original one to this form:

$$\bar{X} + C * \sqrt{\frac{logT(N)}{t(N_i)}} + \sqrt{\frac{\sum x^2 - t(N_i) * \bar{X}^2 + D}{t(N_i)}}$$

The first two terms corresponds to the original UCT formula and the latter one takes the variance into account. The disadvantage of this approach is that there are even more parameters to be tuned, not mentioning the necessity of adjusting it to the expected range of the possible final score. As long as it can be estimated the is no major problem. In this approach a method called 'scaling' is used. It might take some considerable amount of time to tune the C and D constants.

3.7.2 The new proposed selection formula

I keep the first term representing the average:

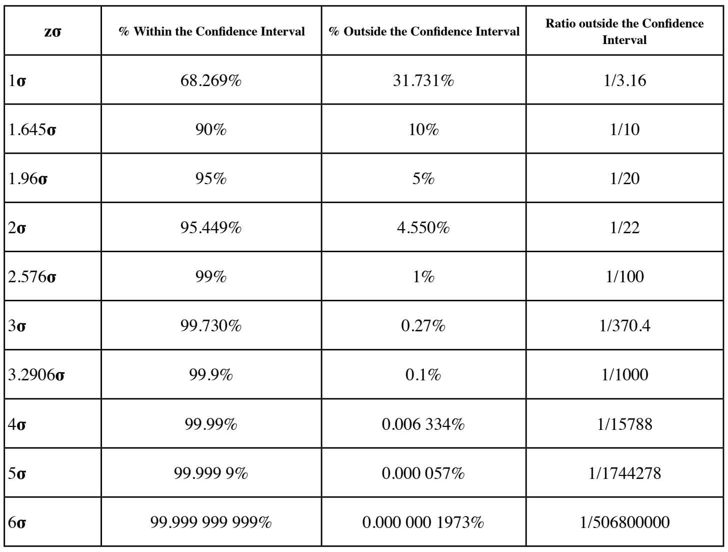
$$\bar{X}$$

But instead of the other two terms I put so called *Standard error of the mean (***SEM)**[56] multiplied by a constant σ.

$$SEM = \frac{Stdev}{\sqrt(t(N_i))}$$

Therefore the final formula is:

$$\bar{X} + \sigma * \frac{Stdev}{\sqrt(t(N_i))} = \bar{X} + \sigma * \sqrt{\frac{\sum(\bar{X} - x_i)^2}{(t(N_i))}}$$



zσ	% Within the Confidence Interval	% Outside the Confidence Interval	Ratio outside the Confidence Interval
1σ	68.269%	31.731%	1/3.16
1.645σ	90%	10%	1/10
1.96σ	95%	5%	1/20
2σ	95.449%	4.550%	1/22
2.576σ	99%	1%	1/100
3σ	99.730%	0.27%	1/370.4
3.2906σ	99.9%	0.1%	1/1000
4σ	99.99%	0.006 334%	1/15788
5σ	99.999 9%	0.000 057%	1/1744278
6σ	99.999 999 999%	0.000 000 1973%	1/506800000

Figure 3.21: Confidence Intervals

SEM is a way of estimating the confidence bounds and the σ parameter defines the range, therefore while we increase the σ value, the exploration importance increases. Therefore not only have we simplified the formula by removing constants needing tuning, but additionally this approach does not require a priori knowledge about the expected score range. During the selection process we consider the Upper Error Bound for a node. The nodes which are less explored have higher error, therefore their value is higher and they have priority during the selection process. Eventually when n becomes infinite, it converges to the average value.

A standard error uses the standard deviation in relation to the sample size. That is the greater the sample, the smaller the standard error. If the data are assumed to be normally distributed, quantiles of the normal distribution, sample mean and standard error can be used to calculate approximate confidence intervals for the mean. In statistics, a Z-score (standard score) indicates how many standard deviations an observation or datum are above or below the mean.

Given the value $SEM(\bar{x})$ the expected percentage of values which are above or below given confidence interval(CI) can be calculated. It is associated with the Z-score. The confidence limits of the sample mean are defined. I.e. a 99.9% CI can be defined by the range

$$\{\bar{x} - 3.2906\sigma, \bar{x} + 3.2906\sigma\}$$

which in application to the mean gives the range

$$\{\bar{x} - 3.2906 * SEM(\bar{x}), \bar{x} + 3.2906 * SEM(\bar{x})\}$$

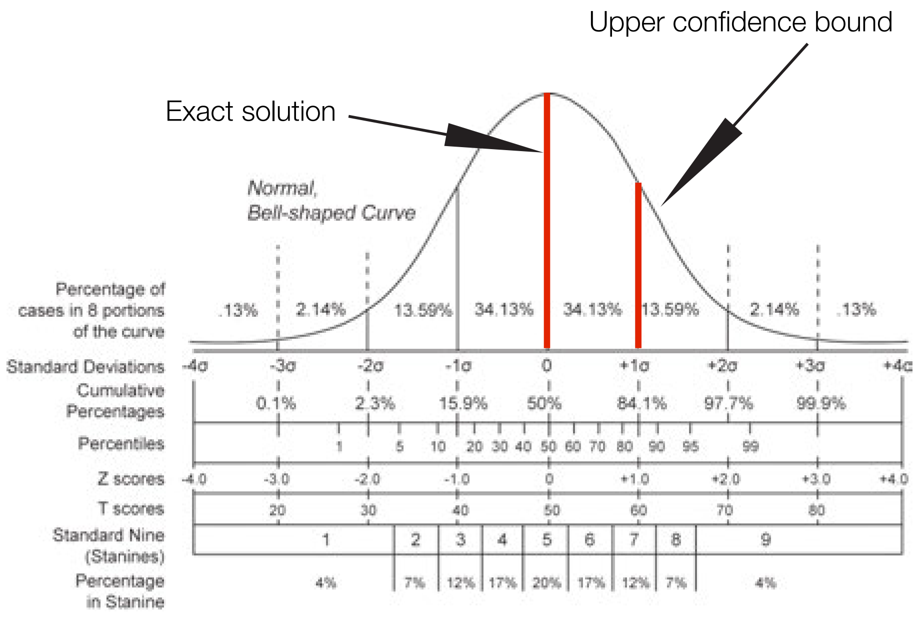


Figure 3.22: SEM illustration

Expansion strategy. In our implementation we chose to expand all of the selected node's children with a predefined number of initial simulations to provide the base value. The number of simulations while adding a node can affect the overall performance, more simulations will make the process slower, but the search will be more accurate. In our experiments, this value was usually set within the range of 2 to 6. Adjusting the σ value affects the search process, when σ value is low, the search becomes exploitative.

3.8 Variance based error estimation for node selection in 2-player version

Similarly as in the previous section, I use error analysis to improve the estimation of nodes values during the final decision step. In my implementation I correct our result using the lower value, which means that the more the average value uncertain is, the lower its confidence value (Example - Figure 3.23). I also consider two types of error calculation. One, where I calculate the error only for the root's children based on the simulation number and successes. Then the approximation is made to choose the next best possible move. In the other case I consider calculating the error at all times during the MCTS selection process for all the nodes. The latter method can sustain the performance improvement for longer search times, when the number of nodes in the trees is greater. A standard error uses the standard deviation in relation to the sample size. That is the greater the sample, the smaller the standard error. If the data are assumed to be normally distributed, quantiles of the normal distribution and the sample mean and standard error can be used to calculate approximate confidence intervals for the mean. In statistics, a Z-score (standard score) indicates how many standard deviations an observation or datum is above or below the mean. Based on this I alter the original decision making step by adding the variance based information to the pure average value of a node.

Therefore, having a range:

$$\{\bar{x} - y\sigma, \bar{x} + y\sigma\}$$

I search for the highest value within a specified confidence bound.

In summary:

- Calculation is based on all gathered samples
- It works better for larger number number of samples utilizes leaf parallelism

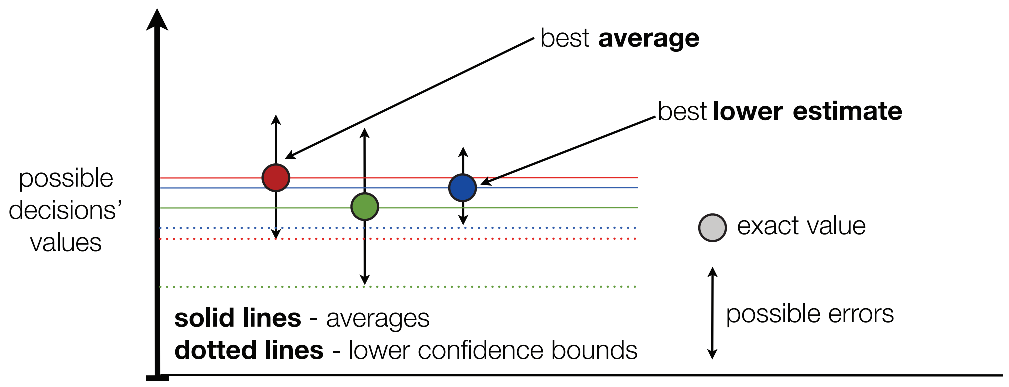

best **average**
best **lower estimate**
possible decisions' values
exact value
possible errors
solid lines - averages
dotted lines - lower confidence bounds

Figure 3.23: Variance based error estimation - decision making step

- It is applied to the decision making step (final selection)

3.9 Random sequence generation

In the current CUDA implentation (3.2) there is no easy and fast method of generating random number on GPU, like the CPU corresponding algorithms provide. I use the Mersenne Twister[57] which is a pseudorandom number generator on GPU, because it of its fast generation of very high-quality pseudorandom number generation. It has a very long period of $2^{19937} - 1$. Then I am sending the random numbers from the CPU memory to the GPU memory. Assuming that there are maximally 64k GPU threads running simultaneously and that the depth of the trees can reach 62 levels in Reversi and around 100 in Samegame, in order to obtain unique sequences for each thread, we would need $64 * 1024 * 100 \sim 6.5M$ numbers. That is a big amount of data to send constantly and definitely only to be stored in and accessed from the global memory. Therefore in my approach I generate only as many random numbers as there are threads, meaning in this example 64k, 100 hundred times fewer numbers to be generated and sent. In order to obtain unique sequences my algorithm work as follows(Figure 3.24):

```
const unsigned long tid = (blockDim.x * blockIdx.x + threadIdx.x); //thread id
const unsigned long total = gridDim.x * blockDim.x; // total number of threads
__shared__ short localrands[128]; // shared, fast local memory

__syncthreads();

localrands[threadIdx.x] = randoms[tid]; //load the numbers into the shared memory

__syncthreads();

int temp_rand;
```

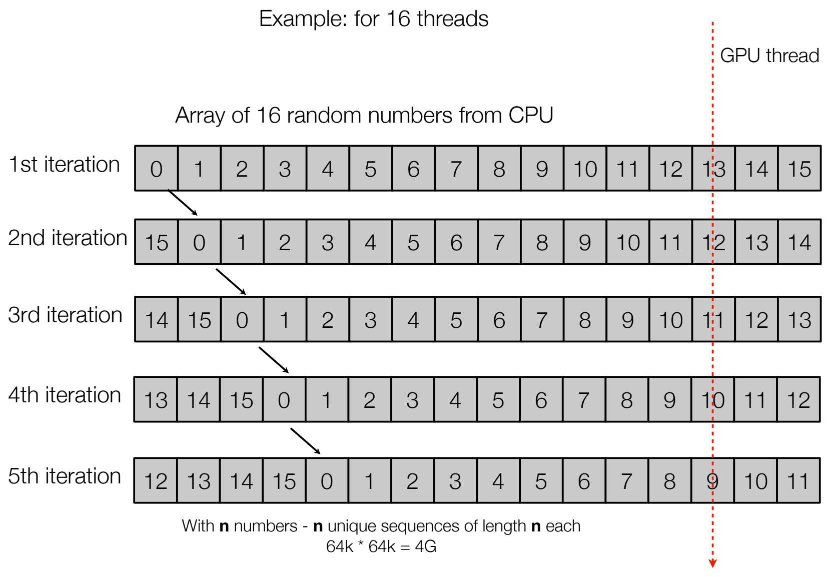


Figure 3.24: Fast 'pseudorandom' sequences generation on GPU

```
for (int i = 0; i < maxDepth; i++) {

        // i - current depth
        rand = localrands[(threadIdx.x + i);

        //work
}
```

In this example, since the shared memory buffer's size (for each GPU block) is 128, I am able to generate up to 128 sequences of depth 128. If global memory is used and all the numbers are accessible it increased of course. The method of getting random sequences relies on shifting the array at each depth level by one. I tested this approach experimentally and compared to the 'fully' randomly filled sequences. It performs as well quality-wise, but it is faster. The way I measured that was by calculating the number of duplicate sequences in a set[58]. This fraction remained the same for my approach and the slower one.

Chapter 4

MCTS: Results and analysis

Contents

4.1 Reversi

In general there are some basic features of graphs presenting the results which need to be explained, it is important to understand how to read them as they may seem to be complicated.

- **Number of simulation per second** in regard to the number of CPU threads, GPU threads or other factors - (i.e. Figure 4.1 or 4.4). By this I mean the average number of random playouts performed (MCTS algorithm - step 3) during the game.

- **Score** (or average score) in regard to the number of CPU threads or GPU threads - (i.e. Figure 4.2). Score means the point difference between player A and player B who play against each other. The higher score, the better. If score is greater than 0, it means a winning situation, 0 means a draw, otherwise a loss.

- **A game step** is a particular game stage starting from the initial Reversi position until the moment when no move is possible. As it was mentioned before, there can be up to 60 game steps in Reversi. As the game progresses, the average complexity of the search tree changes (Chapter 1, Figure 1.8, the way the algorithm behaves changes as well. Instead of showing the score or the speed of the algorithm in regard to the number of cores, I also show the performance considering the game stage.

- **Win ratio** - Another type of measuring the strength of an algorithm. It means the proportion of the games won to the total number of games played. The higher, the better.

- **Different game types** - There are different combinations of playing agents considered. The first one is a randomly playing one, thus having no *intelligence*. We define a randomly playing computer as a player which at all times chooses a move at random (from a given group of possible moves). The reason for using such an agent is to provide an opponent which could be easily used for comparison of different algorithms. The other ones are the sequential MCTS CPU algorithm, parallel MCTS CPU (root parallelism) algorithm and GPU (or multi-GPU) algorithms. Therefore the possible combinations include CPU-random, Multi-CPU-CPU, CPU-GPU, Multi-CPU-GPU etc.

4.1.1 CPU MPI Implementation

Figure 4.1 shows the average number of simulations per second in total depending on the number of cpu cores. The very little overhead is observed during the MPI communication. The number of simulations per second increases almost linearly and for 1024 threads the speedup is around 1020-fold (around 8 million simulations/sec).

Fig. 4.2 shows the average score of CPU MCTS parallel algorithm playing against randomly acting computer depending on the number of cpu cores. Here, a high increase in the algorithm's strength is observed when the thread number is increased up to 32, later the strength increase is not as significant, which can lead to a conclusion that the root parallel MCTS algorithm can have some limitations regarding the strength scalability given the constant increase in the speed shown in Figure 4.1. Actually this has been studied ([55][18][59]) and some conclusions have been formed that such a limit exists and that the root-parallel algorithm performs well only up to several cores. I will form my own thoughts on this matter in the later part of this chapter after presenting all the results.

Figure 4.3 shows the average score and win ratio of MCTS parallel algorithm playing against sequential MCTS agent depending on the number of cpu cores. In this case the opponent is much stronger, since it is actually making reasonable decisions. From this graph it can be seen that obviously when 1 root-parallel MCTS thread plays against the sequential MCTS, they

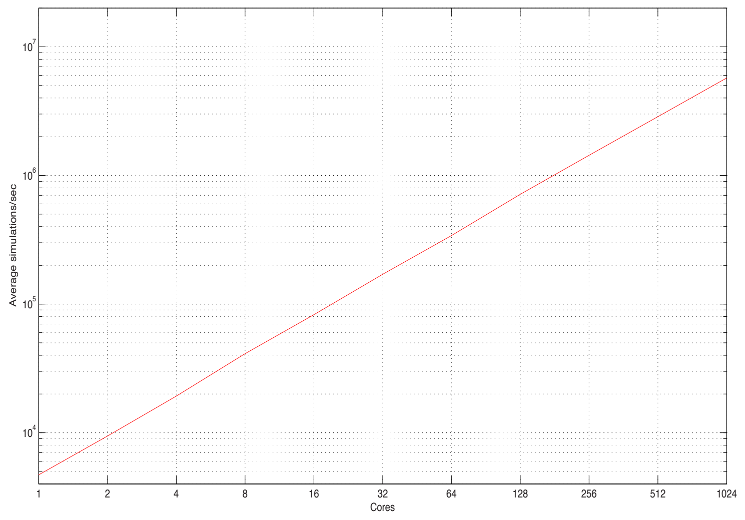


Figure 4.1: Average number of simulations per second in total depending on the number of cpu cores (multi-node) - TSUBAME 1.2

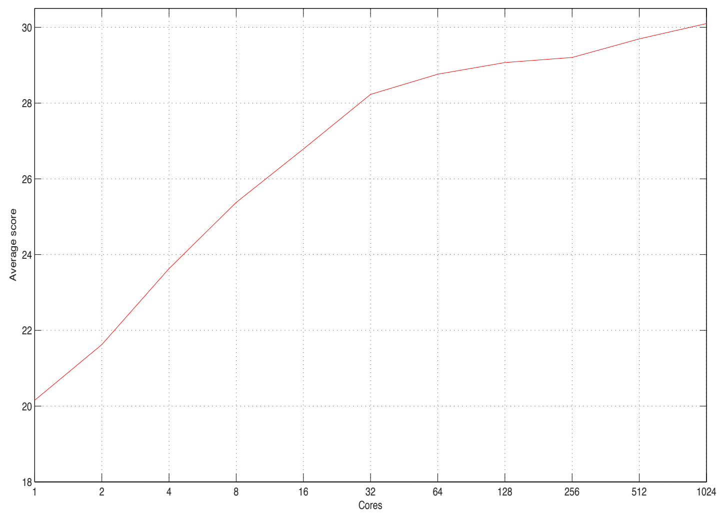


Figure 4.2: Average score of MCTS parallel algorithm playing against randomly acting computer depending on the number of cpu cores (multi-node) - TSUBAME 1.2

are equal (the winning percentage of around 48% for each of them - the missing 4% are the draws). In this graph I present the absolute score (not the score difference between players, so it is not so tightly associated with the winning ratio, but still the more, the stronger the algorithm). When the number of cores doubles, the parallel algorithm wins in more than 60% of the cases, and when the number of threads equals 16, it reaches 90%, to get to the level of around 98% for 64 threads. Then again the improvement is not clearly visible as the number of threads is further increased up to 1024 and the score stops increasing as well. It is another sign that there might be a problem when scaling the algorithm for more threads.

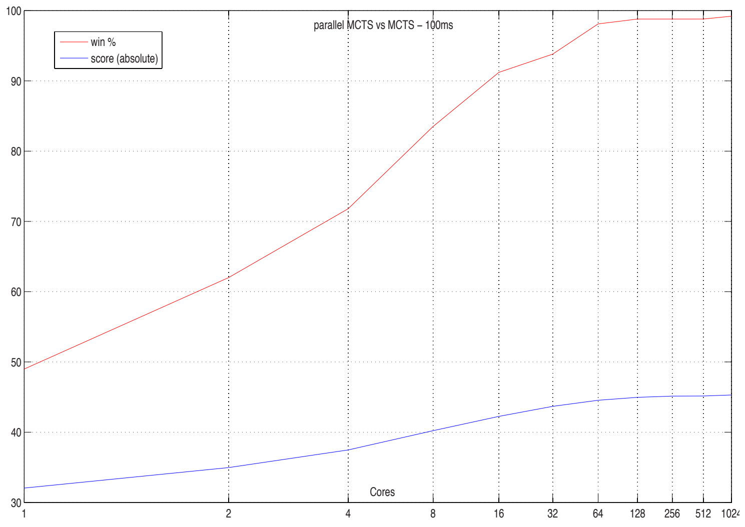


Figure 4.3: Average score and win ratio of MCTS parallel algorithm playing against sequential MCTS agent depending on the number of cpu cores - TSUBAME 1.2

The first and main conclusion of the results obtained is that the root parallelization method is very scalable in terms of multi-CPU communication and number of simulations performed in given time increases significantly. Another one is that there is a point when raising the number of trees in the root parallel MCTS does not give a significant strength improvement.

4.1.2 Single GPU Implementation

In this part I will present the results of the single GPU MCTS implementation. They include leaf parallelization and block parallelization as well as compare the speed of GPU and CPU processing and the strength of both implementations. If not specified otherwise, the MCTS search time = 500 ms, and GPU block size = 128. In all cases the warp size is 32, so is the minimal block size.

4.1.2.1 Block Parallelism

Figure 4.4 is a simple comparison of the peak processing speed of 4 i7 CPU cores and one Nvidia GTX 280 GPU as a typical desktop computer configuration. Here the GPU reaches 2 million simulations per second while 4 CPU cores around 300 thousand, which makes the GPU more than 6 times faster.

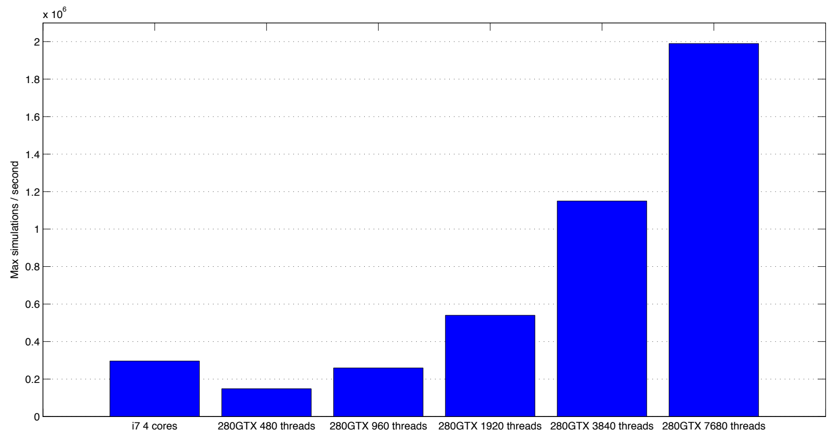


Figure 4.4: Comparison of number of simulations per second - single CPU/single GPU

Figures 4.5 and 4.6 present a more complex comparison where different Nvidia Tesla C2050 GPU thread configurations (leaf parallel) play against 1 Core i7?980X core (sequential MCTS). One thing is that the characteristics are slightly different, CPU's speed increases much faster when the game is in the final phase and the state-space size decreases (game step 25 and higher). Is is an important observation and shows that GPU is actually slower when the kernel has to executed very often (when the state-space is small, the processing time is short) and then the advantage of parallel GPU processing diminishes. One of the other things which are a consequence of that is that if the simulations part takes shorter time, the MCTS algorithm iterations are shorter, therefore, there are more nodes in the tree and its deeper. The other, that around 16 Tesla threads are as fast as one CPU core on average.

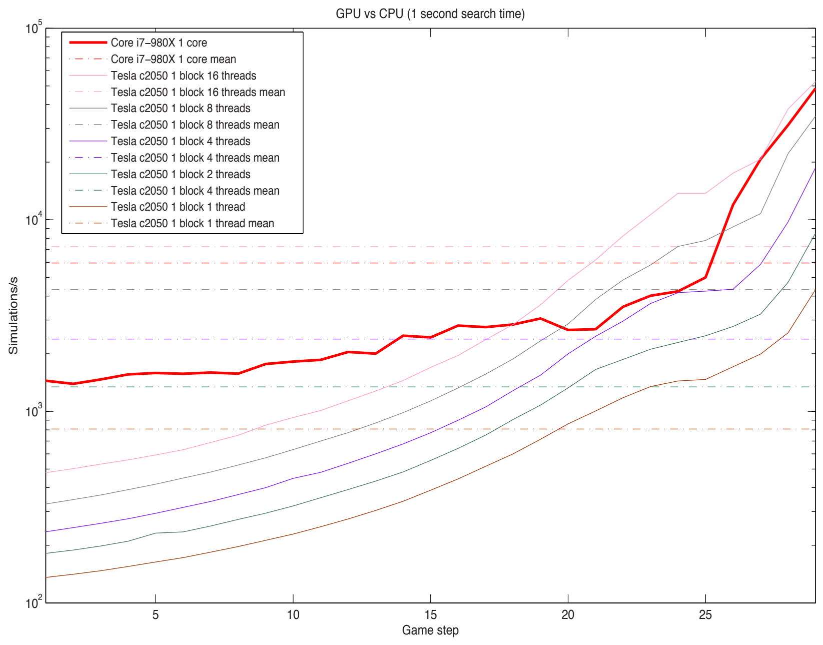


Figure 4.5: Comparison of number of simulations per second - multiple CPU cores/single GPU

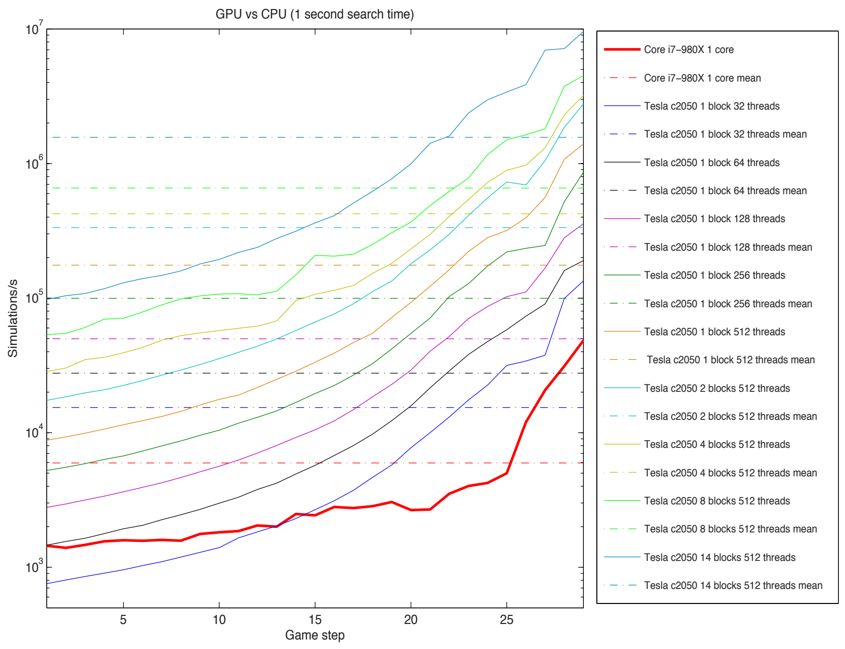


Figure 4.6: Comparison of number of simulations per second - multiple CPU cores/single GPU - continued

Testing platform - TSUBAME 2.0 Supercomputer

- CPUs - Intel(R) Xeon(R) CPU X5670 @ 2.93GHz
- GPUs - NVIDIA Tesla C2050 - 14 (MP) x 32 (Cores/MP) = 448 (Cores) @ 1.15 GHz

Leaf parallelism vs block parallelism vs root parallelism Now, there are 2 figures which present the results of the block-parallel GPU implementation and compare them to the simplest leaf parallel one. First, Figure 4.7 shows the average processing speed of these 2 algorithms in regard to the number of GPU threads used and the block size. Clearly the leaf parallel approach is faster, since there is no tree management cost (there is only one tree), and the GPU is busier. In case of the block-parallel implementation, the algorithm gets slower as the block size decreases (the number of trees increases).

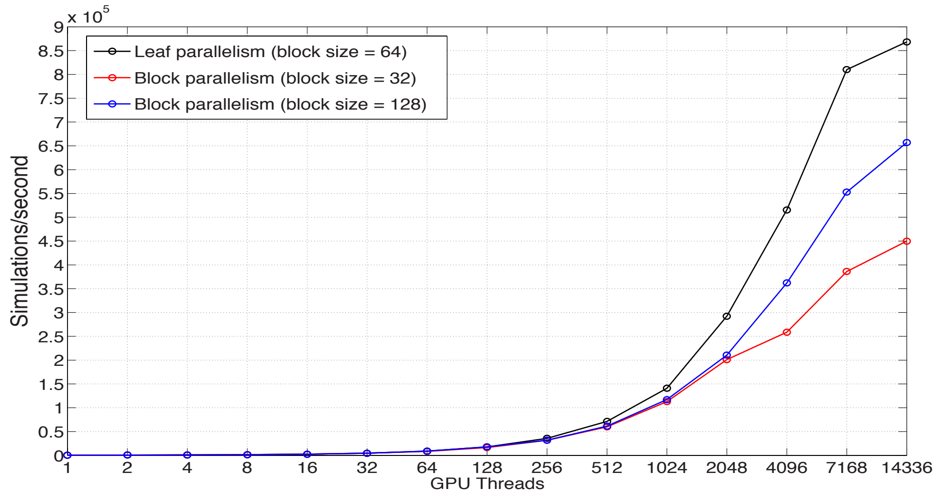


Figure 4.7: Block parallelism vs Leaf parallelism, speed (sim/s)

However, Figure 4.8 shows that in case of leaf parallelism the peak winning ratio equals approximately 0.78 and it is reached for 1024 threads, after that points the algorithm gets even weaker. It can be explained by the fact that the strength increase of the leaf parallel approach is lower by the strength decrease causes by the longer processing time (more GPU threads, more time needed for the kernel to finish processing). In case of the block parallelism, it is always stronger than the leaf-parallel version. When the block size equals 32 the results are improving until the number of trees reaches 224 (7168 threads), then for 448 trees the winning ratio decreases. When the block size is was increases to 128, the algorithm is slightly weaker until that point than the 32-block-size version, but then it continues to get stronger.

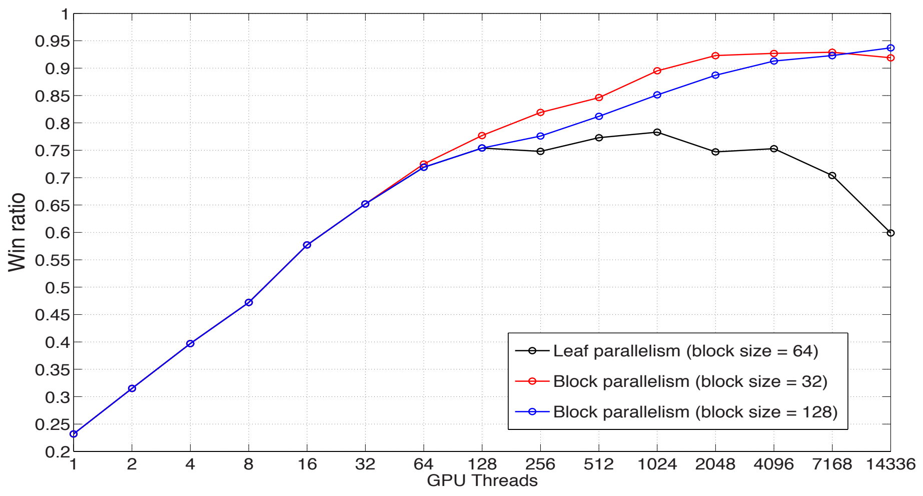


Figure 4.8: Block parallelism vs Leaf parallelism, final result (points) - 1 GPU vs 1 CPU

More simulations/s = better score? Not always. Clearly it was presented that it is not necessarily true as the leaf parallel algorithm is the fastest, but the weakest one. In case of block parallelism, it varies, depends on the block size and thread number.

Block parallelism: number of trees vs their management cost There are basically 4 factors which can in theory improve the strength of a block-parallel algorithm:

- The speed (more simulations) - more threads (trees x block size), increases the GPU load, but causes longer processing time
- The number of trees (increased root parallelism) - increases the strength, but decreases the speed
- Reduced sequential part of the algorithm (fewer trees), decreases the strength, but increases the frequency GPU kernel is executed

The last 2 points are contradicting (and all 3 points involve block size and the number of trees) and a clear rule how to choose the proper block size and the number of trees at the same time is not easy. It can vary from application to application, implementation and it might depend on the hardware as well. The dependencies are complex. It can be observed that for such a small number of threads, the performance decreases while more blocks/trees have to be managed. In my case a number of 128 threads per block was a reasonable choice, providing quite good results for both small and large number of threads.

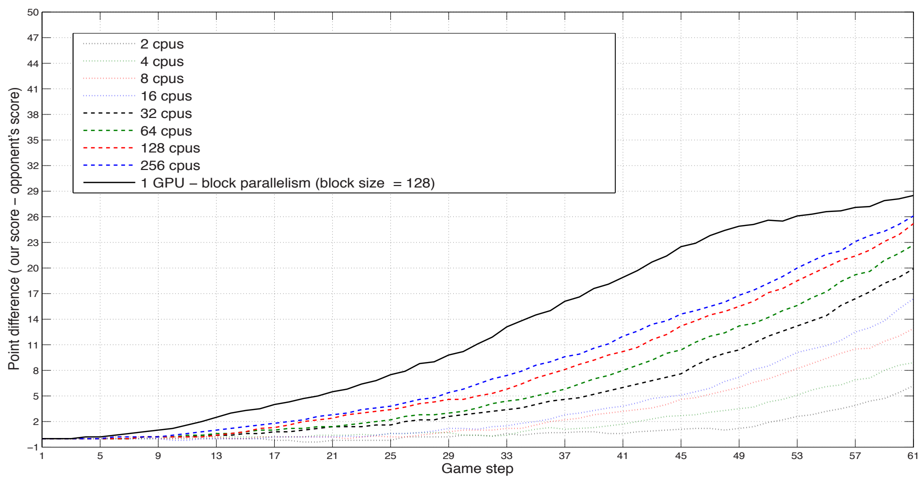


Figure 4.9: GPU vs root-parallel CPUs

In Figure 4.9 the strength of a block-parallel MCTS GPU player is presented compared to the strength of multiple CPU cores (root parallelization). In all cases the agents were playing against a basic sequential MCTS CPU version. There are 2 important things about this graph. The first one is that one Tesla GPU outperforms even 256 cpus throughout all the game in terms of score and the final result (last game step, right side of the X axis). The other is that the characteristics of the intermediate score are different for CPU and GPU. The GPU is stronger in the middle part of the game and in the last stages of the game it gets weaker, while CPU algorithm gains advantage gradually. It can be explained by figures 4.5 and 4.5 that the GPU gets slower in the later stages of the game and the search tree becomes shallower. Since the CPU produces deeper tree, the result of the search might be more accurate. It can be seen in Figure 4.10. Also the next section will discuss this problem and provide a possible improvement.

4.1.2.2 Hybrid CPU-GPU

As it has been mentioned, due to the long kernel processing time, GPU MCTS implementation suffers from significantly shallower search trees compared to the CPU version (Figure 4.10). One of the possible solutions is to use both CPU and GPU in order to build the tree as described in previous chapter in section 3.6. The results show that using this modification, the tree depth increases significantly and also this improves the overall strength of the algorithm.

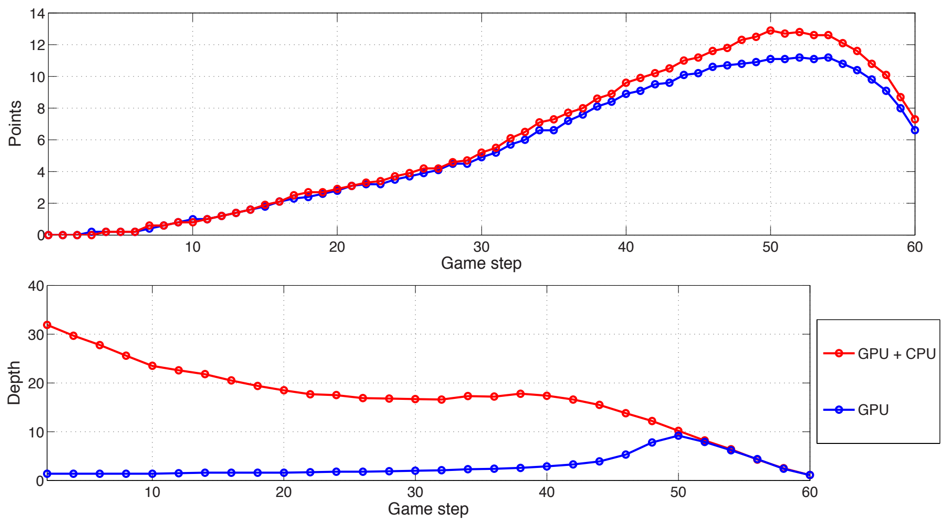


Figure 4.10: Hybrid CPU/GPU vs GPU-only processing

4.1.2.3 Variance based error estimation for node selection

Figure **??** presents the improvement in the algorithm's strength when the variance error estimation is applied to the decision making step. Since Standard Error of the Mean (SEM) calculation utilizes the large number of samples (leaf parallel part of the algorithm), certain gains in the overall and intermediate score are observed for both leaf and block parallel algorithms. In fact the relative gain in performance is much higher for the leaf parallelism, where the algorithm becomes as strong as the basic block parallel version without the error estimation. The sigma value has too be manually tuned to get good results, if it is too small, then there will not be any significant change. If it is too big, then the results may become even worse than in the case when sigma equals 0.

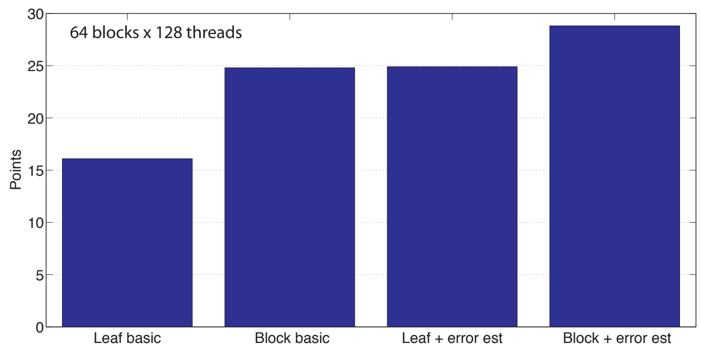


Figure 4.11: Variance based error estimation - score improvement

4.1.3 Multi GPU Implementation

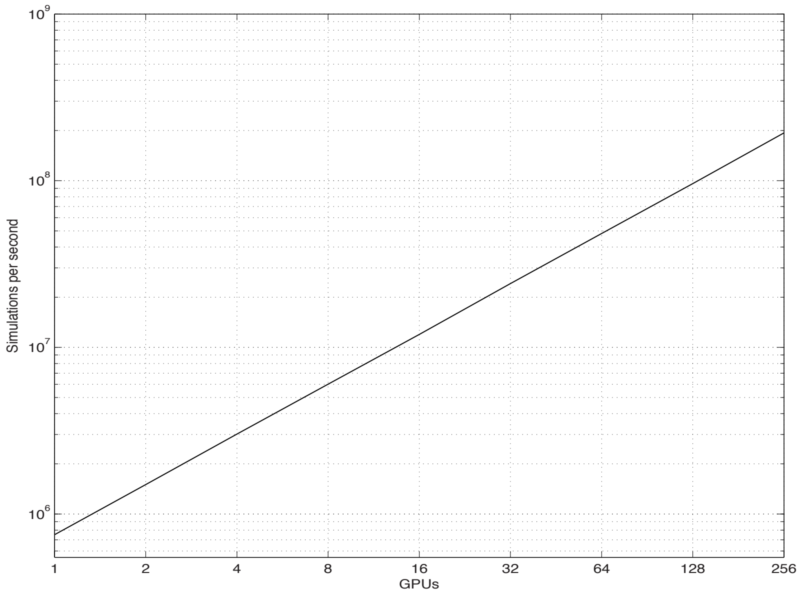


Figure 4.12: Number of simulations per second depending on the GPU number, 56x112 threads per GPU (TSUBAME 2.0)

Multi-GPU implementation follows the same pattern as the multi-CPU scheme. Each TSUBAME 2.0 node has 3 GPUs and using more than 3 GPUs requires MPI utilization. In Figure 4.12 we see that just like with the multiple CPUs, there is no inter-node communication bottleneck and the raw simulation speed increases linearly reaching approximately $2*10^8$ simulations/second when 256 GPUs (128 nodes, 2 GPUs per node) are used. Another graph (Figure 4.13 shows that just like in the case of multiple CPUs, even when the simulation speed increases linearly for multiple devices, the strength of the algorithm does not. Here it is presented for up to 32 GPUs. Figure 4.14 shows the same occurrence for different GPU configurations playing against root-parallel 16 CPUs, but in regard to the game progress. The results get better significantly when the number of GPUs is increased from 1 to 2,3 or 6, but the difference between the strength of this algorithm when 6 and 12 GPUs are used is not so big. It is another reason to think that there is a problem limiting the scalability of the algorithm. The next illustration (Figure 4.15) presents the average score during the game for two AI agents playing against a sequential MCTS agent - the first one uses 2048 CPUs in the root-parallel configuration and the other one takes advantage of 256 GPUs' computing power using block-parallel MCTS. It can be said that the results are quite similar. Therefore, if one GPU was as strong as 256 CPUs and 256 GPUs are as strong as 2048 CPUs, then it means that either the block-parallel GPU approach scales worse than the root-parallel CPU one or that above certain number of threads used, there is no significant improvement in the algorithm's strength. Also we see that the characteristics of the scores during the game stages are slightly different.

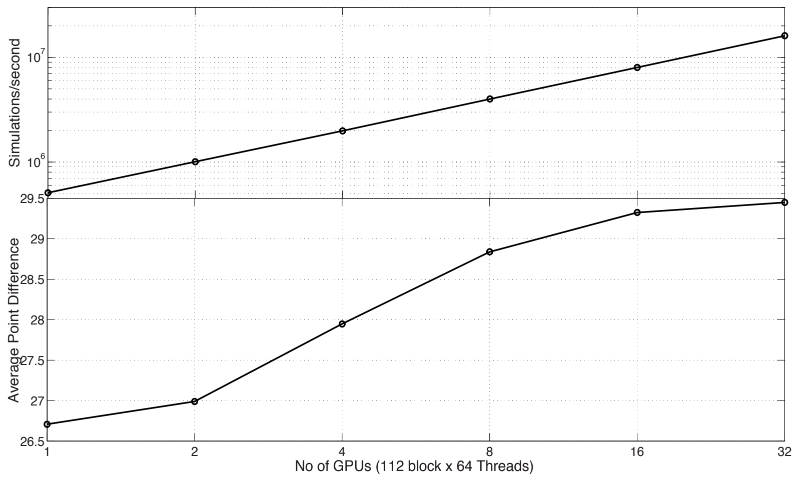


Figure 4.13: 1 CPU core vs multiple GPUs (TSUBAME 2.0), raw speed vs results (leaf parallelism)

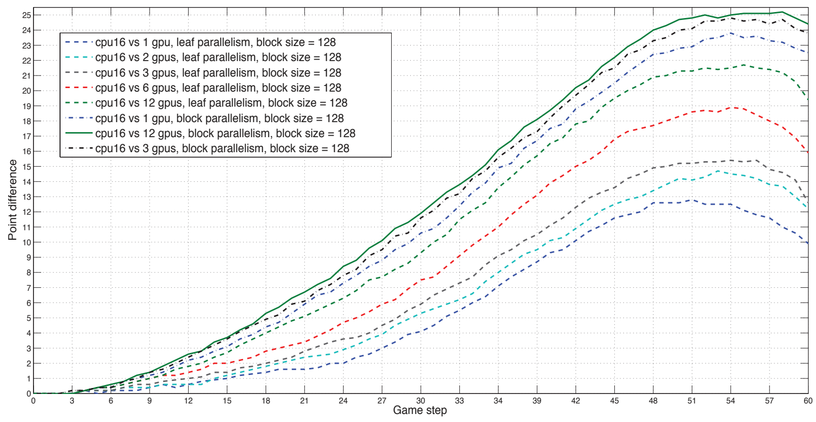


Figure 4.14: 16 CPU cores vs different GPU configurations on TSUBAME 2.0 at particular game stages

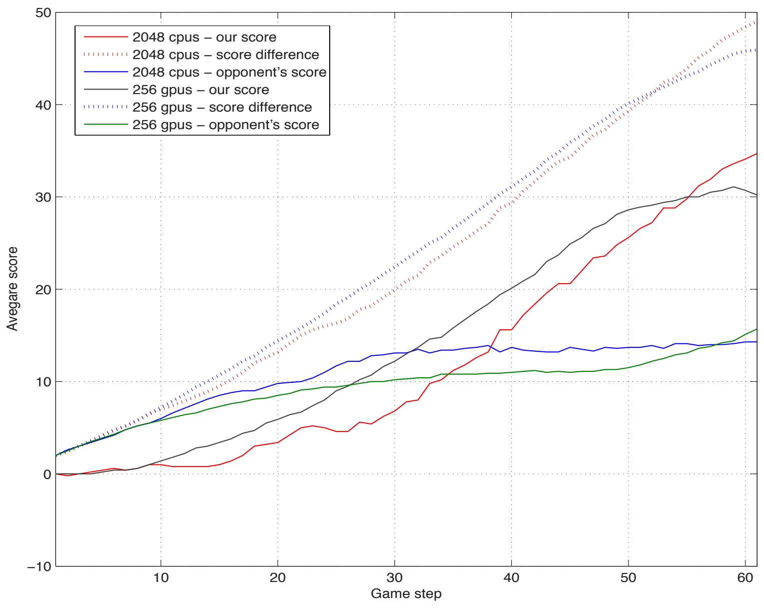


Figure 4.15: 2048 CPU cores vs 128 GPUs on TSUBAME 2.0 at particular game stages

4.2 Samegame

I modified the algorithm as described in Section 3.7 to be able to solve SameGame puzzle in order to get as many points as possible. The first reason to do this was to test if MCTS algorithm can be easily applied to other domains. The second one to check if block parallelism follows the same patterns as in Reversi. And the last one to check the scalability limitations, since it is easier to adjust fewer parameters in SameGame rather than having a 2-player game and to see if there are any similarities in 2 problems. Before applying MCTS to SameGame it was not sure if the problem itself may cause limitations in scalability. The first graph (Figure 4.16 shows results of 3 schemes: root parallelism, block parallelism and leaf parallelism. Points signify particular results of multiple runs, so that the variance of the obtained scores can be observed. What can be read from this figure is that root parallelism performs the best on average, block parallelism is slightly weaker and in the case on leaf parallelism there was no improvement as the number of threads is increased. Although when all the results are taken under consideration, not only the average, we see that in fact the worst results obtained in the whole sets tend to become better in all cases and block parallelism is better than leaf parallelism, but weaker than root parallelism.

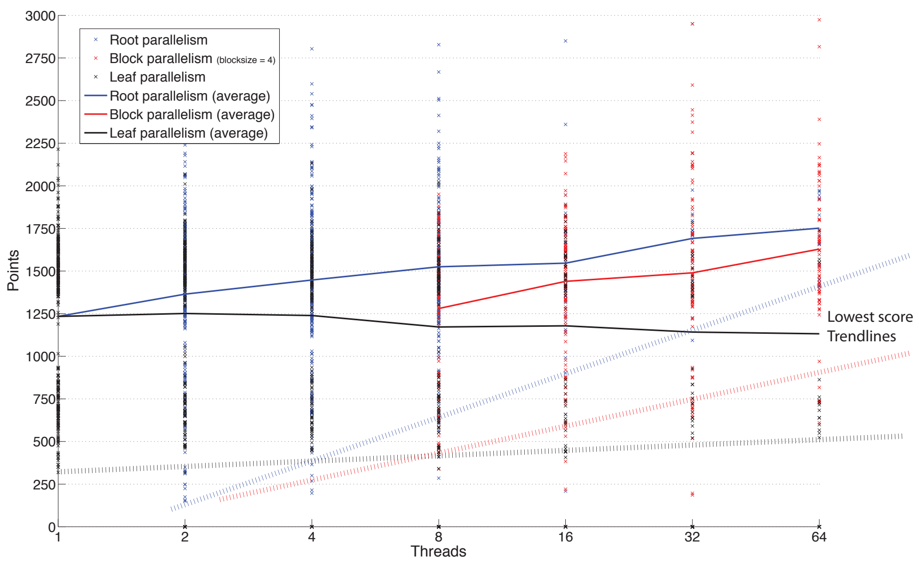


Figure 4.16: Average points in SameGame solving using parallelization and block parallelism - Sigma = 3, Max nodes = 10k (CPU implementation)

4.3 Scalability analysis

The next illustrations (Figures 4.18 and 4.17) show results of changing the MCTS environment to analyze the scalability affecting parameters. First in 4.18 we see how changing the sigma (exploitation/exploration ratio) affects it and Figure 4.17 presents how changing the problem size impacts the strength increase of the parallel approach. There are 2 things to be observed which are

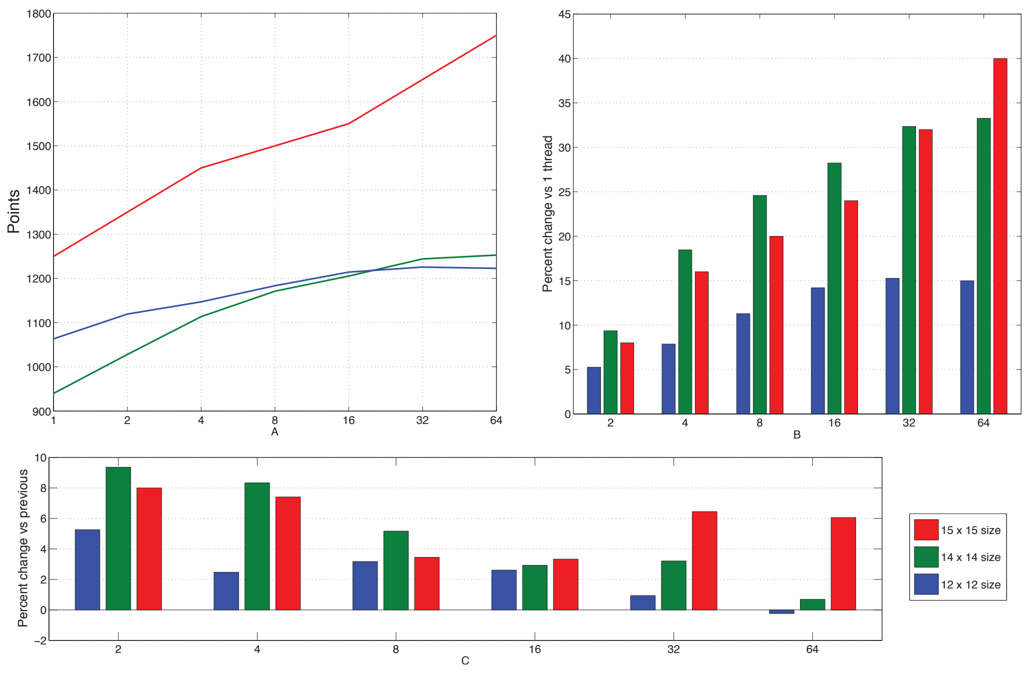


Figure 4.17: Scalability of the algorithm when the problem size changes, (A) - absolute score, (B) - relative change in regard to 1 thread score, (C) - relative change when number of threads is doubled, sigma = 3, Root parallelism, Max nodes = 10k

important. First, the sigma parameter affects the scalability in the way, that when the exploitation of a single tree is promoted, the overall strength of the algorithm improves better with the thread increase. In my opinion, this is due to the deeper tree search for each of the separate trees. When the exploration ratio is higher, then the trees basically form similar trees and reducing it propagates more diversified tree structures among the threads. Then in the next figure we see that as the problem size decreases, the improvement while parallelizing the algorithm also diminishes. It means that parallel MCTS algorithm presents *weak parallelism*, so it would also mean as long as we increase the problem size, the number of threads can grow and the results improve.

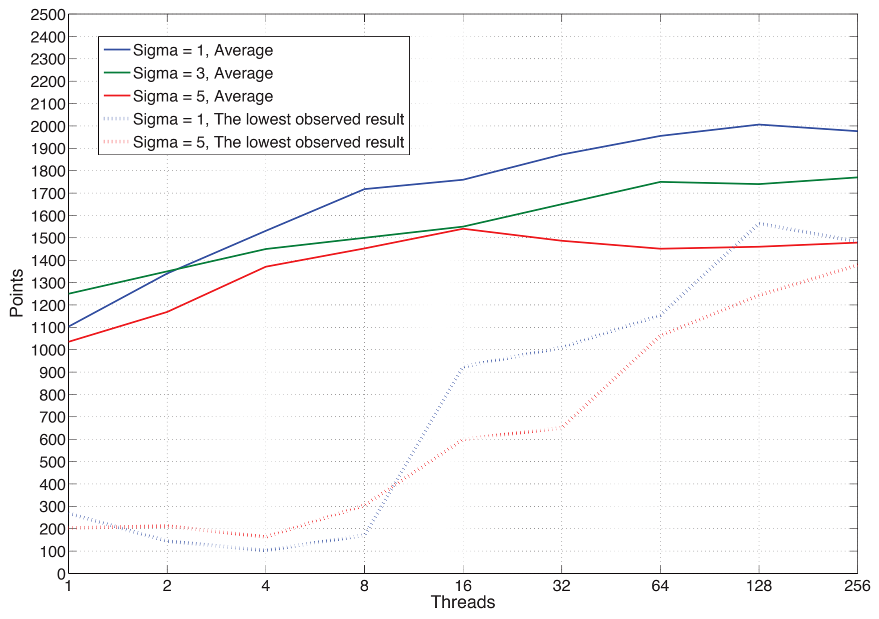


Figure 4.18: Differences in scores during increasing number of threads and changing sigma constant, Root parallelism, Max nodes = 10k

Random sampling with replacement This paragraph is an analysis of possible cause of scalability limit in regard to the problem size. What is interesting in this case, when large number of threads is simulating at random is to obtain the probability of having exactly x distinct simulations after n simulations. Then, assuming that m is the total number of possible combinations (without order), $D(m, n)$ can be calculated, which is the expected number of distinct samples.

$$P(m, n, x) = \frac{\binom{m}{x}\binom{n-1}{n-x}}{\binom{m+n-1}{n}}$$

$P(m, n, x)$ is the probability of having exactly x distinct simulations after n simulations, where m is the total number of possible combinations (according the theorems of combinatorics, sampling with replacement). Then:

$$D(m, n) = \sum_{x=1}^{n} x * P(m, n, x)$$

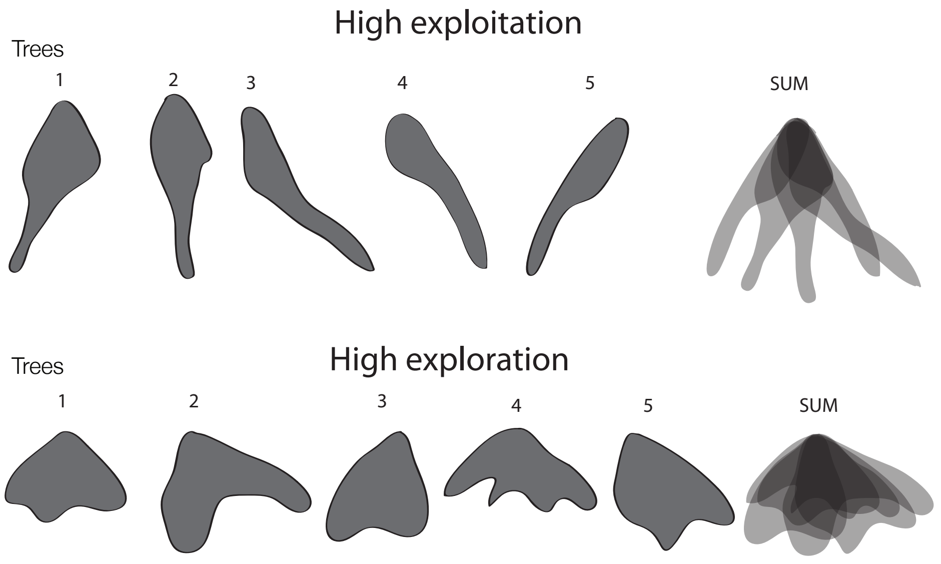


Figure 4.19: Explanation of performance change caused by sigma constant adjustment

It is hard to calculate D(m,n) for big numbers because of the factorials, but according to [58] an approximation (slightly underestimating) can be used. Let $r = \frac{n}{m}$. Then:

$$D(m,n) \sim m(1-e^{-r}) + \frac{r}{2}e^{-r} - \frac{O(r(1+r)e^{-r}}{n}$$

The first term is the most important. Having this I was able to analyze the impact of having large number of samples in regard to the state space size and check how many of those samples would repeat (theoretically). For an instance if:

$m = 10^8$, $n_1 = 10^4$, $n_2 = 10^7$

In the first case (n_1): $\frac{D(m,n_1)}{n_1}$ $\sim$ 99.5% - almost no repeating samples

Then if I consider (n_2): $\frac{D(m,n_2)}{n_2}$ $\sim$ 95.1% - around 4.9% samples are repeated

This means that the scalability clearly depends on the space state size/number of samples relation. As the number of samples approaches the number of possible paths in a tree, the algorithm will lose its parallel properties and even finding the exact solution is not guaranteed, since we would have to consider infinite number of samples. Figure 4.20 presents another example close to Reversi, when the maximum depth of the tree is 60 and the branching factor is fixed here and equals 4. It can be observed that as the tree gets smaller (the solution is closer) the number of repeated samples increases (the higher the line the more repeated samples there are). When the tree is shallow enough (depth is lower than 10) it is very significant.

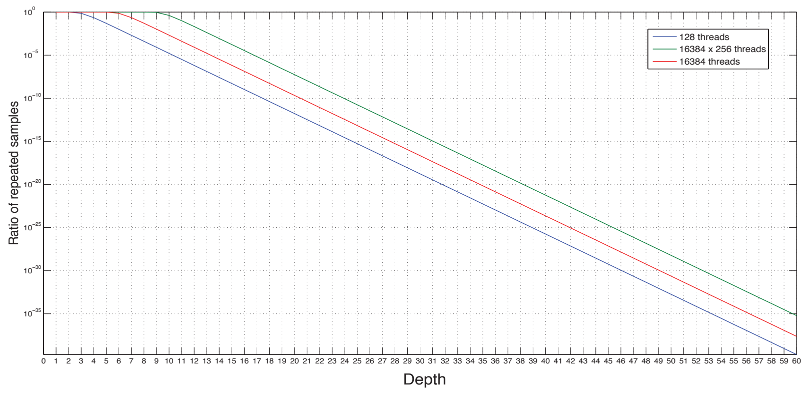


Figure 4.20: Ratio of repeated samples (Y axis), Tree size depth (X axis) for different number of threads

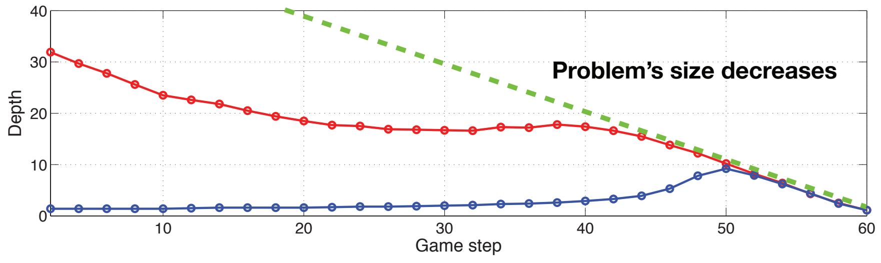


Figure 4.21: Explanation of performance change caused by the problem size change

If the state-space is small, the impact of the parallelism will be diminished (Figure 4.21). Low problem complexity may be caused by the problem is simple itself (i.e. small instances of SameGame, TicTacToe) or in the ending phases of games/puzzles (only few steps ahead left).

Chapter 5

Conclusions and open problems

5.1 Summary

GPU Minimax implementation analysis. I showed limitations of the GPU Minimax implementation, improvements and what can be done in the future, explained why it is not a good approach.

GPU MCTS Implementation. I introduced and described my GPU MCTS implementation.

MCTS Block Parallelism SIMD hardware limitation. I solved the problem using the block parallel scheme.

Hybrid GPU/CPU processing. Since GPU needs more time, the MCTS trees are smaller. Therefore I proposed a modification which partially diminishes this negative effect.

Variance-based error estimation. I presented a modification of the algorithm utilizing leaf parallelism and improving the strength of the algorithm

Single player MCTS modification. I showed a modified, simpler formula for one-player games/puzzles.

Scalability analysis. In the last part of the thesis I analyzed possible scalability limitations which include:

- The problem complexity
- The tree depth
- Random number generation, bounds for the number of unique random sequences
- The implementation

I showed that for both problems (Reversi and SameGame) a similar scalability problem occurs and it is not caused by any communication issues. It is most likely that this an algorithm demonstrating weak-parallelism and the scalability problems are affected by the problem size. The importance of this

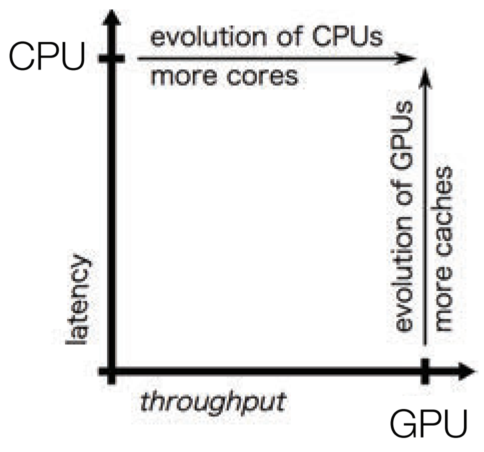


Figure 5.1: CPU-GPU convergence

research and the block-parallel algorithm is caused by its possible application on future hardware. The GPUs will most likely keep working in SIMD way, whereas CPUs will contain more cores and probably utilize SIMD operations (like PowerPC). Programming CPU may become harder, GPU programming can be simplified. In both cases this algorithm could be applied.

5.2 Future work

Power usage analysis. It should be investigated what is the power consumption in regard to the GPU configuration and implementation.

Further investigation on large scale tree search.

New applications of GPU MCTS.

Compare SP-MCTS algorithm and the new proposed in this thesis in terms of performance/strength

Open problems.

- Can we find a better algorithm than parallel MCTS?
- Will there always be a scalability problem?
- Can strong-scaling be achieved with GPUs?

Effective parallelization of alfabeta pruning on GPU It is possible that with the progressing improvement of GPUs and changes in the architecture it will be easier to achieve this task.

Bibliography

[1] www.top500.org (Cited on page 2.)

[2] Stuart Russell and Peter Norvig: Artificial Intelligence: A Modern Approach: 2nd Edition, Chapter 3, Prentice Hall, 2003 (Cited on pages 5, 7, 12 and 29.)

[3] Henri Bal: The shared data-object model as a paradigm for programming distributed systems. Ph.D. thesis, Vrije Universiteit, 1989 (Cited on page 29.)

[4] Rainer Feldmann, Peter Mysliwietz, Burkhard Monien: A Fully Distributed Chess Program. Advances in Computer Chess 6, 1993 (Cited on page 29.)

[5] Rainer Feldmann: Game Tree Search on Massively Parallel Systems. Ph.D. Thesis, 1993 (Cited on pages 29 and 30.)

[6] Chris Joerg and Bradley C. Kuszmaul: Massively Parallel Chess, 1994 (Cited on page 30.)

[7] Bradley C. Kuszmaul : Synchronized MIMD Computing. Ph. D. Thesis, Massachusetts Institute of Technology, 1994 (Cited on page 30.)

[8] Bradley C. Kuszmaul: The StarTech Massively Parallel Chess Program, 1995 (Cited on page 30.)

[9] Valavan Manohararajah: Parallel Alpha-Beta Search on Shared Memory Multiprocessors. Masters Thesis, 2001 (Cited on pages 26 and 30.)

[10] Robert Hyatt: The DTS high-performance parallel tree search algorithm, 1994 (Cited on page 30.)

[11] Hopp H., Sanders P.: Parallel Game Tree Search on SIMD Machines, Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems, 1995 (Cited on pages 27 and 30.)

[12] Knuth, D. E., Moore, R. W.: An analysis of alpha-beta pruning. Artificial Intelligence, 6, 1975 (Cited on pages 25 and 30.)

[13] Kazuki Yoshizoe, Akihiro Kishimoto, Tomoyuki Kaneko, Haruhiro Yoshimoto and Yutaka Ishikawa: Scalable Distributed Monte-Carlo Tree Search: SoCS 2011, Symposium on Combinatorial Search, Castell de Cardona, Barcelona, Spain, July 15-16, 2011 (Cited on page 31.)

[14] Romein, J. W.; Plaat, A.; Bal, H. E.; and Schaeffer, J.: Transposition table driven work scheduling in distributed search. In Proc of AAAI, 1999 (Cited on page 31.)

[15] Kamil Rocki, Reiji Suda: Parallel Minimax Tree Searching on GPU, PPAM Conference 2009, Wroclaw, Poland (Cited on page 27.)

[16] R. Coulom: Efficient selectivity and backup operators in Monte-Carlo tree search, in Proceedings of the 5th International Conference on Computers and Games, 2006 (Cited on page 31.)

[17] L. Kocsis and C. Szepesvari: Bandit based Monte-Carlo Planning, Proceedings of the EMCL 2006, volume 4212 of Lecture Notes in Computer Science (LNCS), pages 282-293, Berlin, 2006 (Cited on pages 31, 35 and 37.)

[18] 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 (Cited on pages 9, 10, 31, 37, 41, 42, 43, 44 and 58.)

[19] http://www.site-constructor.com/othello/othellorules.html (Cited on page 8.)

[20] Victor Allis, "Searching for Solutions in Games and Artificial Intelligence". Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands. ISBN 9090074880, 1994 (Cited on page 10.)

[21] Shigeki Iwata, Takumi Kasai, The Othello game on an n x n board is PSPACE-complete, Theoretical Computer Science, Volume 123, Issue 2, 31 January 1994 (Cited on page 10.)

[22] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, and H. Aldewereld: Addressing NP-Complete Puzzles with Monte-Carlo Methods. In Volume 9: Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning, pages 55-61, Brighton, UK, 2008 (Cited on pages 10, 36 and 51.)

[23] http://samegame.sourceforge.net/ (Cited on page 10.)

[24] Thomas J. Schaefer, On the complexity of some two-person perfect-information games, Journal of Computer and System Sciences, Volume 16, Issue 2, April 1978 (Cited on page 10.)

[25] J. M. Robson: ”N by N checkers is Exptime complete”. SIAM Journal on Computing, 13 (2): 252-267, 1984 (Cited on page 10.)

[26] Hiroyuki Adachi, Hiroyuki Kamekawa, and Shigeki Iwata: Shogi on n x n board is complete in exponential time. Transactions of the IEICE, J70-D(10):1843-1852, October 1987 (Cited on page 12.)

[27] http://developer.download.nvidia.com/compute/ cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf (Cited on pages 14, 19 and 23.)

[28] J. M. Robson: ”The complexity of Go”. Information Processing; Proceedings of IFIP Congress. pp. 413-417 (Cited on page 12.)

[29] Christos Papadimitriou: Computational Complexity. Addison-Wesley, 1994 (Cited on page 12.)

[30] Michael Sipser: Introduction to the Theory of Computation. PWS Publishing. ISBN 0-534-94728-X. Section 8.3: PSPACE-completeness, pp. 283-94, 1997 (Cited on page 10.)

[31] Erik D. Demaine: Playing Games with Algorithms: Algorithmic Combinatorial Game Theory, in Proceedings of the 26th Symposium on Mathematical Foundations in Computer Science (MFCS 2001), Lecture Notes in Computer Science, volume 2136, Marianske Lazne, Czech Republic, August 27-31, 2001 (Cited on pages 10 and 12.)

[32] Victor Allis: Searching for Solutions in Games and Artificial Intelligence. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands, 1994 (Cited on page 10.)

[33] Claude Shannon: ”Programming a Computer for Playing Chess”. Philosophical Magazine 41 (314), 1950 (Cited on page 10.)

[34] Shi-Jim Yen, Jr-Chang Chen, Tai-Ning Yang, and Shun-Chin Hsu: ”Computer Chinese Chess”. International Computer Games Association Journal 27, March 2004 (Cited on page 10.)

[35] David S. Johnson: The NP-completeness column: an ongoing guide, Journal of Algorithms , 1985 (Cited on page 12.)

[36] Maarten P. Schadd, Mark H. Winands, H. Jaap Herik, Guillaume M. Chaslot, and Jos W. Uiterwijk: Single-Player Monte-Carlo Tree Search. In Proceedings of the 6th international conference on Computers and Games (CG ’08), Springer-Verlag, Berlin, 2008 (Cited on page 11.)

[37] P.E. Hart, N.J. Nilsson, and B. Raphael: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernatics, SSC-4(2):100-107, 1968 (Cited on page 13.)

[38] R.E. Korf. Depth-first iterative deepening: An optimal admissable tree search. Artificial Intelligence, 27(1): 97-109, 1985 (Cited on page 13.)

[39] Fermi Compute Architecture Whitepaper. Nvidia corporation, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf (Cited on page 24.)

[40] http://www.kde.cs.tsukuba.ac.jp/∼vjordan/docs/master-thesis/nvidia_gpu_archi/ (Cited on pages 15, 18 and 21.)

[41] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, Andreas Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. Department of Electrical and Computer Engineering, University of Toronto (Cited on page 19.)

[42] CUDA PTX 1.4 official documentation. Nvidia corporation (Cited on pages 11 and 17.)

[43] Kourosh Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. WRL Research Report 95/9, Western Research Laboratory (Cited on page 17.)

[44] Snir Marc, Otto Steve, Huss-Lederman Steven, Walker David, Dongarra Jack, "MPI: The Complete Reference". MIT Press Cambridge, MA, USA, 1995 (Cited on page 25.)

[45] Warren H.S.: Hacker's Delight, Addison Wesley, 2002 (Not cited.)

[46] Hopp H., Sanders P.: Parallel Game Tree Search on SIMD Machines, Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems, 1995 (Cited on page 27.)

[47] Hewett R., Ganesan K.: Consistent Linear speedup in Parallel Alpha-beta Search, Proceedings of the ICCI'92, Fourth International Conference on Computing and Information, 1992 (Cited on page 26.)

[48] Schaeffer, J., Brockington M. G.: The APHID Parallel $\alpha\beta$ algorithm, Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing, 1996 (Cited on page 26.)

[49] Borovska P., Lazarova M.: Efficiency of Parallel Minimax Algorithm for Game Tree Search, Proceedings of the International Conference on Computer Systems and Technologies, 2007 (Cited on page 26.)

[50] Schaeffer J.: Improved Parallel Alpha Beta Search, Proceedings of 1986 ACM Fall joint computer conference, 1986 (Cited on page 26.)

[51] Monte Carlo Tree Search (MCTS) research hub, http://www.mcts-hub.net/index.html (Cited on page 35.)

[52] Romaric Gaudel, Michele Sebag - Feature Selection as a one-player game, 2010 (Cited on page 36.)

[53] Guillaume Chaslot , Steven Jong , Jahn-takeshi Saito , Jos Uiterwijk - Monte-Carlo Tree Search in Production Management Problems, 2006 (Cited on page 36.)

[54] O. Teytaud et. al, High-dimensional planning with Monte-Carlo Tree Search, 2008 (Cited on page 36.)

[55] T. Cazenave and N. Jouandeau: On the parallelization of UCT. In H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M.P.D. Schadd, editors, Proceedings of the Computer Games Workshop 2007 (CGW 2007), pages 93-101. Universiteit Maastricht, Maastricht, The Netherlands, 2007 (Cited on pages 41, 42 and 58.)

[56] Snedecor, George W. and Cochran, William G., ”Statistical Methods”, Eighth Edition, Iowa State University Press, 1989 (Cited on page 52.)

[57] Matsumoto, M.; Nishimura, T: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation, 1998 (Cited on page 55.)

[58] V. F. Kolchin, B. A. Sevastyanov, and V.P. Chistyakov. Random Allocations, 1976 (Cited on pages 56 and 73.)

[59] Segal, R: On the scalability of parallel UCT. In Proc. Computers and Games (CG ’2010), 2010 (Cited on page 58.)