A Parallel Algorithm for Game Tree Search Using GPGPU

Article *in* IEEE Transactions on Parallel and Distributed Systems · July 2014

DOI: 10.1109/TPDS.2014.2345054

Hao Wang
Chinese Academy of Sciences
1,000 PUBLICATIONS **13,713** CITATIONS
SEE PROFILE

Wei Li
Chinese Academy of Sciences
70 PUBLICATIONS **474** CITATIONS
SEE PROFILE

A Parallel Algorithm for Game Tree Search Using GPGPU

Liang Li, Hong Liu, Hao Wang, Taoying Liu, and Wei Li

Abstract—Game tree search is a classical problem in the field of game theory and artificial intelligence. Fast game tree search algorithm is critical for computer games asking for real-time responses. In this paper, we focus on how to leverage massive parallelism capabilities of GPU to accelerate the speed of game tree search algorithms and propose a concise and general parallel game tree search algorithm on GPU. The performance model of our algorithm is presented and analyzed theoretically. We implement the algorithm for two real computer games called Connect6 and Chess. We also use these two games to verify the effectiveness and efficiency of our algorithm. Experiments support our theoretical results and show good performance of our approach. Compared to classical CPU-based game tree search algorithms, our algorithm can achieve speedups of 89.95x for Connect6 and 11.43x for Chess, in case of no pruning. When pruning is considered, which means the practical performance of our algorithm, the speedup can reach about 10.58x for Connect6 and 7.26x for Chess. The insight of our work is that using GPU is a feasible way to improve the performance of game tree search algorithms.

Index Terms—Parallel computing, GPU, game tree search, alpha-beta pruning, Conect6, Chess

1 Introduction

Since its original application in graph processing, *General-Purpose computing on Graphics Processing Unit* (GPGPU or GPU) has been applied in many other fields that require massive parallel computing. In fact, lots of applications have get benefits from the massive parallelism capability of GPU [11] [17] [25]. In addition, researchers have also utilized GPU to solve some specific AI (Artificial Intelligence) problems successfully [1]. These practice shows that GPU becomes a promising way to solve compute-intensive AI problems due to its SIMD architecture specialized for parallel computing. Furthermore, considering the important role of *Game Tree Search* (GTS for short) in AI, it is worth to study the possibility and feasibility of utilizing GPU to improve the performance of GTS related algorithms that are widely used in computer games.

Basically, the aim of GTS algorithms is to search a game tree to find the best move in a computer game. Since GTS is a combinatorial game theory problem, it is hard to find an optimal solution for many computer games (i.e. Connect6 [31] and Chess [24]) due to their exponential time complexity. Therefore, two topics are studied mostly in the field of GTS research: one is to find better GTS algorithms to obtain near-optimal solutions; another one is to use advanced computing technologies to accelerate the speed of GTS algorithms. In practice, it is critical to reduce computing time of GTS algorithms for applications asking for real-time response, e.g., online computer game, decision tree, expert system and etc. To satisfy such a requirement, parallel computing technologies were introduced to improve the performance of GTS algorithms. Actually, CPU-based parallelism methods have been studied for decades [3] [12]. There are lots of important progress achieved that significantly expand the use of AI.

In this paper, our motivation is to investigate if GTS can get benefit from GPU and compare our proposed GPU-based approach with classical CPU-based approaches. In this respect, we first analyze the challenges of GPU-based GTS algorithms and give some insight on how to resolve them. Generally speaking, our study faces three challenging problems: *complexity of algorithm design on SIMD architecture*, *low performance of divergences on GPU for rule-based computer games* and *low pruning efficiency of parallel GTS algorithms*. In fact, solving the above problems depends on carefully exploiting parallelism potential of GPU.

Then, we propose following techniques to implement an effective and efficient GTS algorithm on GPU: In the first, we adopt *node-based parallel computing* for game tree search, which suites for GPU architecture. It makes the design of parallel GTS algorithms on GPU much easier than that of traditional *tree-based GTS algorithms*. In the second, we combine depth-first and breadth-first search methods to reduce the number of nodes to be calculated. This feature can eliminate the effect of low pruning efficiency and achieve good performance. In the third, we use a hybrid method to use both CPUs and GPU. This feature helps us fully utilize the advantage of both architectures. Based on the above ideas, we implement a GPU-based GTS algorithm, which can be used as a general framework for most of computer games. We also

- *Liang Li, Hong Liu, Taoying Liu and Wei Li are with Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. E-mail: {liliang, liuh, lty, liwei}@ict.ac.cn*
- *Hao Wang is with Department of Computer Science at Virginia Tech, VA, USA. E-mail: hwang121@cs.vt.edu.*

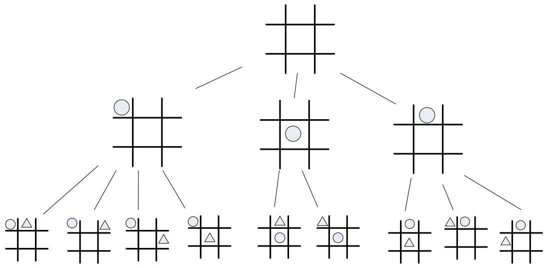


Fig. 1: The Two-Ply Game Tree of Tic-tac-toe

provide the performance model for this algorithm and analyze key factors that affect its performance.

We finally implement the proposed GPU-based algorithms for two real computer games *Connect6* [31] and *Chess* [24] to verify our design and analysis. Both the AI implementations of the two game in our GPU-based parallel algorithms derive from CPU-based parallel algorithms. Testing results show that our method can achieve satisfactory performance.

Our study has been organized as follows: Section 2 describes the problem space of GTS. Section 3 gives the previous work on parallel GTS algorithms and challenges for designing GPU-based approaches; Section 4 presents the details of our GPU-based method; Section 5 offers a performance model of our algorithm and its theoretical execution time; Section 6 uses two real games to verify our theoretical results; and our conclusions are presented in Section 7.

2 The problem of GTS

2.1 The Problem

In the field of AI, GTS is a classical method to find the optimal strategy for combinatorial games and has been studied for decades [3] [28]. Basically, the game tree in game theory is a directed graph in which nodes are *positions* in a game and edges are *moves*. Here, different positions indicate different game states and moves are choices for players in the current game position. In this subsection, we use a two-ply game tree of *tic-tac-toe* [30] as an example to introduce how GTS algorithm works.

Fig. 1 shows a two-level or two-ply game tree of tic-tac-toe [30]. The game board is a 3×3 grid. Two players take turns to occupy blanks of the grid. The player who can occupy three respective blanks in a horizontal, vertical, or diagonal row will win. Also as shown in Fig. 1, the process of the game can be represented by a game tree in a mathematical form. The tree starts at the initial position of the game as its root. Moves of two players are represented by triangles and circles. In Fig. 1, the player represented by the circle plays the game first and each node in this tree is a possible situation of the game. For the above game played by computers, various GTS algorithms are designed to find optimal moves. Commonly, GTS algorithms will dynamically generate the tree structure from the current position, traverse the tree, evaluate all possible moves, and find the best move.

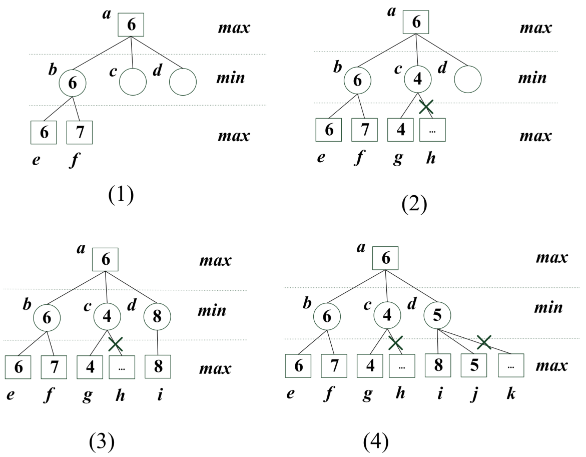
Basically, GTS algorithms contain two key functions: *tree search* and *node calculation*. The search function focuses on traversing the game tree to find optimal moves (i.e. searching the tree in Fig. 1). For computer games with high search complexity, technologies such as *historic knowledge* and *pruning* are used to avoid unnecessary visiting on certain nodes. The node calculation function consists of *leaf calculation* (i.e. *evaluation function*) and *branch calculation* (i.e. *move generation function*). The function will calculate each node (leaf or branch) and assign scores to nodes. The node with the highest score will be selected as the best move for players. Since the score calculation is rule-specific, different computer games may have different policies on node calculation.

In fact, many computer games can be solved by the above procedure similarly. Representative methods include *minimax algorithm* [2] and its variant [6], and *negamax algorithm* with *alpha-beta pruning* [15]. Here, the negamax algorithm with alpha-beta pruning is shown in Algorithm 1. Note that the algorithm contains the depth-first tree search procedure and pruning procedure. In Algorithm 1, the input parameter $node$ indicates the current node of the game tree, $depth$ indicates the depth of *node*, α and β are threshold of maximal and minimal value of pruning. Based on the description in Algorithm 1, we take a small tree of three plies as an example shown in Fig. 2 to explain how the algorithm works. Here, the game tree is a *max-min tree*, which selects the maximum value of its children in odd plies and minimum value in even plies. The algorithm will start the search process from node a in Fig. 2 and traverse the game tree for the best solution.

Algorithm 1 Alpha Beta Search

```
Input:  $node, depth, \alpha, \beta$
Output:  $moveBest$
1:  if $node$ is leaf or $depth = 0$ then
2:      return evaluation($node$);
3:  end if
4:  $moveList$ = moveGeneration($node$);
5:  for $move$ in $moveList$ do
6:      $val$ = - AlphaBeta($move, depth - 1, -\beta, -\alpha$);
7:      if $val \geq \beta$ then
8:          return $val$;
9:      end if
10:     if $val \geq \alpha$ then
11:         $moveBest$ = $move$;
12:         $\alpha$ = $val$;
13:     end if
14: end for
15: return $\alpha$ and $moveBest$;
```

In Fig. 2(1), for node a, the move generation function in Step 4 of Algorithm 1 will obtain its child node b, c and d; then the search function will take node b as the current node, calculate it recursively for the next ply in Step 6 and pass the values of α and β to next



(1) (2)

(3) (4)

Fig. 2: Alpha-Beta Pruning

plies. We can see that the tree structure is generated dynamically through the recursive search process until the search function reaches to leaf e and f in Step 2, i.e., the children of node b. Then the evaluation for leaves is invoked in Step 2 and their evaluated values 6 and 7 are obtained respectively. According to the score returned by the evaluation function, the parent node b selects the minimum value of its children, which is 6 from node e, and returns it to node a.

In Fig. 2(2), node a continues to searching its next child, i.e., node c and passing the updated value 6 to it as the value of β; after node c traverses its child g, it gets 4, which is smaller than β. Then it cuts all right children (i.e. node h) at Step 8 and return its parent node a, because its evaluated value is 4 at most and its parent will never pick up this branch for a larger one. As shown in Fig. 2(3), the algorithm continues to traverse node d and it's child i, acquired 8; while 8 is not smaller than β, it can't prune the rest nodes. But, in Fig. 2(4), node k is pruned after evaluating node j, which is 5.

Note that in this algorithm, the evaluation function and the move generation function are invoked by leaves and branches respectively. In addition, pruning occurs after each child is traversed, according to the accumulated historical information of α and β. Actually, α and β are the lower and upper bounds respectively, and the node will be cut off if returned values from its children are larger than β.

2.2 Parallel GTS Algorithms

In fact, many parallel GTS algorithms have been proposed since 1980s, such as [3] [5] [12] [19]. The parallel GTS algorithm is an ideal solution to speed up computer games, especially for the tree search function. Generally speaking, the parallelism can be implemented at different levels of computer systems such as *multiple-processor level* and *multiple-computer level*. At the former level, techniques such as *multiple-thread* and *multiple-process* [4] [8] [9] [10] may be used; while at the later level, distributed programming models, such as *MPI* (*Message Passing Interface*), are commonly used. Implementations at different levels vary a lot for GTS algorithm design.

TABLE 1: A Classification of Parallel GTS Algorithms

System Structure	**Granularity of Parallelism**	
	Tree-based	Node-based
Multiple-Processor	I. [4] [10] [12] [13] [18] [22] [26] [33] [34]	II. Our method
Multiple-Computer	III. [8] [9] [14] [23] [29]	IV. [38]

In addition, according to the granularity of parallelism, we can divide the parallelism in parallel GTS algorithms into two categories: *tree-based parallelism* and *node-based parallelism*. For the former, the tree search function focuses on tree splitting and assigns many trees (or subtrees) to processors (cores) to calculate concurrently. For the later one, the search function assigns a set of nodes to processors (cores) directly and calculates them simultaneously. Table 1 shows a classification according to the above features. From Table 1, we can see that previous researches mainly use tree-based parallelism to maximize the usage of processors or computers. [4] provides an asynchronous parallel game tree search method on multiple-processor platform and [10] [13] consider the tree-based parallel game tree search on SIMD machines. In the meanwhile, [8] [9] [23] study the problem of how to implement parallel game tree search on multiple-computer and distributed systems. For node-based parallelism, the notable chess machine Deep Blue [38], which defeated the then-reigning World Chess Champion Garry Kasparov in a six-game match in 1997, adopts multi-computer structure which has 30 IBM RS/6000 SP computers. The machine uses a centralized control of parallel search, i.e., one SP computer as the master node controlling the other 29 nodes as slaves, and the early search iterations of the Deep Blue search are calculated on the master node, while the slaves search the last few plies of the chess tree in parallel. In some sense this could be viewed as a node-based algorithm. However, Deep Blue uses customized hardware but not general-purpose processors to calculate leaf and branch nodes.

As the technology of GPU developed, it provides another choice to accelerate GTS algorithms on multiple-processors or multiple-computer systems. Motivated by the high parallelism of many cores in GPU, we propose a node-based parallel search method on GPU as Class II in Table 1, which has fundamental differences from other approaches. More importantly, compared with exiting GPU-based algorithms, our algorithm uses node instead of tree as the basic unit of search.

3 Node-based Game Tree Search on GPU

3.1 Tree-based Parallel GTS Algorithms

Before we present our node-based approach,we briefly introduce classical tree-based algorithms to distinguish the merit of our approach.

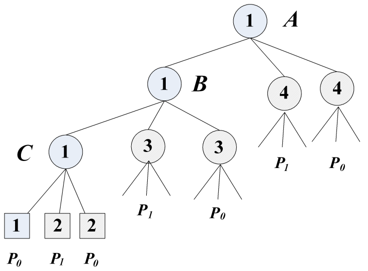


Fig. 3: PVS Search by Two Processors

3.1.1 PVS and EPVS Algorithms

Principal Variation Splitting (PVS) is a straightforward and efficient parallel GTS algorithm on CPU. Fig. 3 illustrates an example of the PVS algorithm with two processors. In PVS, the first branch of *principal nodes* [18] (i.e. nodes marked by 1) in the game tree should be searched serially first by a *processor* p_0 before parallel search of other nodes begins. Once the search of principal nodes is finished, according the PVS algorithm, all processors (i.e. p_0 and p_1) take efforts to begin parallel search by taking unvisited nodes of branch C (i.e. nodes marked by 2) to traverse. After these two nodes are calculated, the search on branch C is finished and processors will take brother nodes of C (i.e. the child nodes of B that marked by 3) to calculate in parallel. Similarly, when all nodes of branch B were calculated, new branches (i.e. the child nodes of A that marked by 4) are assigned to processors to calculate. The processor that finished the search task will inform its updated results to other processors and take the next unassigned branch to calculate. After getting the updated information from others, processors will use the information to help its own searching procedure including pruning. When there are no branches left and all processors finished their tasks, the PVS algorithm will halt and return the best move to the player. The limitation of the PVS algorithm is the overhead of synchronization among processors [18]. That is, a processor who has finished its task needs to wait for other processors' finish to start up another around of calculation. Therefore, *Enhanced PVS* (EPVS) is introduced, whose idea is assigning subtrees to idle processors from other busy processors. Obviously, the enhanced method will bring extra communication overhead. However, the performance improvement is significant for an unbalance tree and experiments prove its efficiency [18].

3.1.2 The DTS Algorithm

Dynamic Tree Splitting (DTS) [18] [33] [34] is another method for parallel game tree search, which uses a peer-to-peer model on multi-processor systems. This approach is based on a shared global list of active *split-points* (SP-LIST), by which all processors will find uncalculated nodes to process.

At the beginning of the parallel search, SP-LIST is cleared and one processor will take the root node of the game tree and other nodes remain in the idle state. Then an idle processor will look up SP-LIST to find a split point [18] in the game tree to traverse the subtree. If there is no points left in SP-LIST, the idle processor will broadcast a *HELP* message to all processors. Busy processors that receive the message will split and copy the state of the subtree to SP-LIST. The idle processor then consults SP-LIST again and obtains a split point and search the subtree from that point. The DTS algorithm completes after all processors stop in an idle state and no split-points left. Through the search process, the values of α and β for pruning are shared by these processors and will be updated as each processor's traversing. Due to the space limits, interested readers can refer to [18] [33] [34] for further understanding of this algorithm.

Compared with PVS and EPVS algorithms, the advantage of DTS algorithm is its usability and scalability. Unlike PVS and EPVS, DTS does not split the tree by specific nodes; therefore various game trees, no matter how wide or how deep they are, DTS algorithm can search them concurrently and efficiently. Also, it is oversimplified to add new processors for this algorithm because of its point-to-point protocol, which results in high scalability in the algorithm. In fact, many implementation of DTS algorithm for searching game tree contains hundreds of processors in their system [3] [6].

3.2 Challenges for GTS Algorithm Design on GPU

As discussed earlier, the motivation of our work is to use GPU to process thousands of game tree nodes simultaneously. However, as we know, GPU is very sensitive to the instruction divergence caused by the control flow in a warp. In fact, for GPU-based GTS algorithms, the kernel of GPU is the function to calculate the node, which involves lots of control flow instructions because of game rules. Due to the SIMD feature of GPU, the tree-based approach on CPUs cannot be easily adopted in GPU. That is, new approaches that can fully utilize the potential of GPU should be considered. More specifically, in order to develop efficient GPU-based GTS algorithms, we need to solve following challenges:

Complex control logic of parallel search on GPU. GPU-based parallel GTS algorithm design is much more complex than CPU-based algorithm design. On CPUs, we can search the tree simply according to the alpha-beta search. However, on GPUs, we have to unfold the recursive search and maintain the game tree in the memory. This significantly increases the system complexity, so we need to design algorithms carefully.

Low performance of divergent execution on GPU for rule-based computer games. As we mentioned above, GPU have low performance in divergent execution. Since most of GTS related calculation is rule-based, there are many divergences in node calculation functions, which will lead to a low speedup and affect the overall performance of GTS algorithms.

Low pruning efficiency for parallel tree search. As discussed in Section 2, pruning is very useful for serial

GTS algorithms since it can prevent duplicated calculations on game tree nodes. As a matter of facts, the pruning depends on the accumulated knowledge of all situations in the previous nodes, which implies that calculations of different nodes may depend on each other and it is hard to process them in parallel. For GPU-based parallel GTS algorithms, the problem still exists and we need to keep balance between pruning efficiency and parallelism to obtain best overall performance.

To solve the above challenges, we propose a node-based parallel method to utilize the potential of GPU.

3.3 A Node-based Approach on GPU

To solve the challenging problems in Section 3.2, we adopt following ideas to solve GTS problems on GPU:

Our first idea is *to adopt node-based parallel computing for game tree search.* As discussed earlier, the tree-based approach is not suite for GPU architecture. Different from the tree-based approach, our approach will assign a set of nodes from one or multiple subtrees to processors, while the tree-based approach will assign subtrees to processors. The benefit of our method is not only taking advantages of the high concurrency of GPU, but also avoiding the complexity of tree splitting.

Our second idea is *the combination of depth-first and breadth-first search.* That is, in our approach, we calculate many tree nodes in the same depth in the current game tree, which is the breadth-first search. In addition, each cycle in the search process will take in the deepest nodes of the current game tree, which is the depth-first search. In our approach, we not only calculate nodes on GPU, but also avoid the exponential growth of the space complexity through the parallel search process and prune nodes after calculation on GPU. Commonly, there are two methods to search the tree, the *depth-first* and the *breadth-first* search. The former will use less memory and quickly get the leaves, while the latter can traverse the sub-tree quickly and get more branches. Both of them are practicable. In our design on GPU-based GTS algorithm, we select the depth-first search on CPU due to memory limit and use breadth-first search on GPU since we schedule the nodes with the same depth to GPU in each cycle.

Our third idea is *to use CPUs and GPU in a hybrid way, i.e., hybrid programming on both CPUs and GPU* . From Algorithm 1, we know that there are many control logic in GTS algorithms. As we know, CPU architecture is good at complex execution of programs containing control statements such as *switch*, *condition* and *loops*, while GPU architecture specializes in intensive computation on data with the same type. Therefore, our third idea is to use both CPU and GPU architecture in our algorithm. That is, in our algorithm, CPU is responsible for maintaining the tree structure in the host memory and interacting with GPU, while GPU takes in tree nodes from CPU and calculates them simultaneously. Ideally, the approach can take advantage of the capability of GPU to compute massive nodes in parallel and CPU's flexibility to accelerate tree search and pruning.

4 Node-based GTS algorithm on GPU

4.1 The Algorithm

Based on the node-based approach discussed in Section 3.3, we provide our *Node-based GTS Algorithm on GPU* (NGTS-G for short) in Algorithm 2. In this algorithm, P_0 is the initial position for game tree search. M_{Best} is the best move calculated for P_0. L_{max} and L_{min} are thresholds of maximal and minimal number of leaf nodes fetched from a tree. B_{max} and B_{min} are thresholds for branch nodes. *leaf calculation function* and *branch calculation function* are game specific functions that are described in Algorithm 3 and 4 respectively.

Here we will explain some key steps of the algorithm. From Step 5 to Step 17 or from Step 28 to Step 36, leaves or branches are assigned to GPU to calculate concurrently. For branch nodes (Step 28 to Step 36), the algorithm will use GPU to generate child nodes for the next ply. For leaf nodes (Step 5 to Step 17), the algorithm uses GPU to evaluate leaves and prune the game tree. Compared with the process shown in Fig. 3, we can see that our node-based approach assigns nodes but not trees to processors for parallel processing.

From the algorithm, we can see that for node calculation executed by GPU, it will calculate nodes in same depth until no uncalculated nodes left in this layer (i.e. Step 5 to Step 17 or Step 28 to Step 36). This process is so-called *breadth-first search*. However, in respect of the whole search process, especially for node calculation performed by CPU (i.e. Step 18 to Step 27 and Step 37 to Step 42), the algorithm will use *depth-first search* to process the tree. Throughout the algorithm, we can see that CPU is responsible for execution control of the algorithm. That is, CPU is responsible for maintaining the game tree structure (e.g., Step 9, Step 32 and etc.), pruning and assigning tasks to GPU (e.g., Step 14, Step 25). Especially, when the branch and the leaf number are less than B_{min} and L_{min} respectively, our algorithm will use CPU to calculate nodes since CPU will execute faster than GPU in this situation. In Algorithm 2, only two steps are executed by GPU (i.e. Step 7 and Step 30). That is, GPU is used for calculating the branch and the leaf nodes in parallel. By this hybrid manner, our algorithm can fully utilize advantages of both architectures.

The pruning procedure is another key component of our node-based algorithm. In Algorithm 2, Step 14 is to prune redundant nodes. That is, after calculating scores for all leaves, our algorithm will update the parent node of P_0 and check its brother nodes to cut off some nodes according to the pruning procedure discussed in Algorithm 1. Therefore, Step 6 in our algorithm is critical for pruning efficiency since the order of picking up leaf nodes has a great impact on pruning efficiency. That is, dedicated arrangement (e.g. sorting according to the numerical order of leave scores) of leaf nodes can greatly

Algorithm 2 Node-based GTS Algorithm on GPU

```
Input: $P_0$
Output: $M_{Best}$
 1: Set $P_0$ as the $root$ of the game tree $T$;
 2: if $T$ is $NULL$ then
 3:     return $M_{Best}$;
 4: end if
 5: if the number of remaining leaves is more than $L_{min}$
    then
 6:     Get $l$ leaves from tree $T$, $l \leq L_{max}$;
 7:     Call $leafCalculationFunction$ on GPU;
 8:     Get $evaluatedValueList$ by GPU;
 9:     for $each$ in $evaluatedValueList$ do
10:         Update its $parentNode$ by $each$ in $T$;
11:         if $parentNode$ is $root$ then
12:             $M_{Best}$ = $each$;
13:         end if
14:         Prune $T$ from the updated nodes;
15:     end for
16:     Go to Step 2;
17: end if
18: if the number of remaining leaves is more than 0
    then
19:     Get one leaf $node$ from tree $T$;
20:     Call $leafCalculationFunction$ on CPU;
21:     Update the tree $T$ by the evaluated leaf $node$;
22:     if $parentNode$ is $root$ then
23:         $M_{Best}$ = $each$;
24:     end if
25:     Prune $T$ from the updated nodes;
26:     Go to Step 2;
27: end if
28: if the number of remaining branches is more than
    $B_{min}$ then
29:     Get $b$ branches from tree $T$, $b \leq B_{max}$;
30:     Call $branchCalculationFunction$ on GPU;
31:     Get $childNodeList$ by GPU;
32:     for $each$ in $childNodeList$ do
33:         Update $T$ by generated child nodes from $each$;
34:     end for
35:     Go to Step 2;
36: end if
37: if the number of remaining branches is more than 0
    then
38:     Get one leaf $node$ from tree $T$;
39:     Call $branchCalculationFunction$ on CPU;
40:     Update $T$ by new child nodes from $node$;
41:     Go to Step 2;
42: end if
```

improve pruning efficiency. In fact, in our experiments, our test program uses sorting of leaf nodes to achieve good pruning efficiency.

4.2 An Example

Next, we use a simple game tree in Fig. 4 to explain how Algorithm 2 works. For simplicity, we assume that the depth of the tree is 4 and the branch factor is 3. We also assume $L_{min} = L_{max} = B_{min} = B_{max} = 3$. In Fig. 4, circle nodes represent branches and square nodes represent leaves. In addition, shaded nodes mean that they need to be calculated.

After initialization, the algorithm first enters Step 37 because there is only a root node in the game tree (Fig. 4(1)). Since it is better to use CPU when there are few nodes, our algorithm uses CPU (i.e. the classical CPU-based GTS algorithm) to calculate node a and generates node b, c and d. Then the state of the tree changes from Fig. 4(1) to Fig. 4(2).

When there are more leaves or branches to calculate (i.e. $l \geq L_{min}$ or $b \geq B_{min}$), our algorithm enters Step 5 or Step 28 respectively. For the game tree in Fig. 4(2), it will use GPU to calculate node a, b and c as branches simultaneously. Then new branch nodes e to n are generated to form the tree shown in Fig. 4(3). According to our algorithm, for the tree in Fig. 4(3), it will take branch node e, f and g to calculate on GPU. These nodes will generate 9 leaf nodes o to w which are shown in Fig. 4(4). Now, because leaf nodes emerge in the game tree and our algorithm will execute Step 5 to process leaves[2]. That is, our algorithm will take nodes from leaf o to leaf q to calculate their scores. This step involves the process of pruning redundant nodes.

After calculate three leaves (i.e. leaf node o, p and q), branch node e will be updated according to the scores of the above leaf nodes (i.e. updating the value of α and β in Algorithm 1). When our algorithm begin to calculate leaf nodes r, s and t in the next loop (shown in Fig. 4(5)), it will compare the score of leaf nodes r, s and t and update the branch node f. Similar operations may repeat on branch node h. After all child nodes of branch node b are processed, our algorithm will calculate branch node c and d similarly. When all child nodes of node a are processed, our algorithm will return a best move for this game tree.

4.3 Branch and Leaf Calculations

From Algorithm 2, we can see that our idea is to take massive nodes (branches or leaves) to GPU and calculate them concurrently. Note that the algorithm does not include the detail of node calculation (i.e. *leaf calculation function* and *branch calculation function* in Algorithm 2). In fact, the performance of node calculation on GPU is critical for overall performance of our algorithm. That is, if the performance of node calculation on GPU is much slower than that on CPU, our algorithm may be inferior to CPU-based approaches. In this subsection, we will introduce the detail of above two GPU-based functions.

[2] If both leaves and branches are more than their minimal thresholds (i.e. $l \geq L_{max}$ or $b \geq B_{max}$) at the same time.

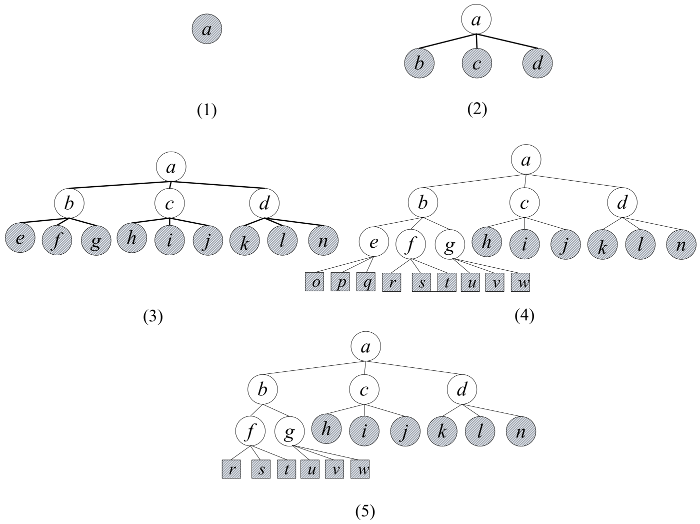


Fig. 4: An Example of Node-based Game Tree Search on GPU

Algorithm 3 Branch Calculation on GPU

```
Input: $Pl$
Output: $Ml$
1: Get $id$ of the current thread;
2: Restore position $P_{id}$ for the current thread, according to $id$ and $Pl$;
3: for each legal point $p$ in $P_{id}$ do
4:     Recognize the shapes of the point $p$ according to game rules;
5:     Set the score for the shape;
6:     Accumulate the score for $p$;
7: end for
8: Assemble the valuable moves as the move list $Ml$;
9: return $Ml$;
```

Algorithm 4 Leaf Calculation on GPU

```
Input: $Pl$
Output: $V$
1: Get $id$ of the current thread;
2: Restore position $P_{id}$ for the current thread, according to $id$ and $Pl$;
3: for each non-empty point $p$ in $P_{id}$ do
4:     Recognize the shapes of the point $p$ according to game rules;
5:     Set the score for the shape;
6:     Accumulate the score for $p$;
7: end for
8: Add up all the values as evaluated value $V$;
9: return $V$;
```

As mentioned earlier, node calculation is game-specific, i.e., different computer games have different rules to calculate the score of branches and leaves. However, different games may share a same framework except the formulas to get the scores of branches and leaves. We use Algorithm 3 and 4 to show the process of *branch calculation function* and *leaf calculation function*.

Both Algorithm 3 and 4 take a list of game positions (i.e. parameter Pl) from CPU as input to GPU. For branch calculation, it returns candidate moves (i.e. parameter Ml) as output. For leaf calculation, it returns an evaluation value (i.e. parameter V). In addition, both algorithms use one thread to calculate a branch or a leaf.

Next, we will explain how Algorithm 3 and 4 work. When executing the algorithms, we need to send the current position from CPU to GPU. In fact, the parameter Pl contains all moves since the beginning of the game. This parameter is used to restore the game tree structure for GPU in Algorithm 3 and 4. After receiving the parameter, GPU begins to execute Algorithm 3 and 4 to find optimal moves for the current players.

Firstly, as shown in Step 1 and 2 of both algorithms, each core on GPU reads the positions in device memory by its thread identity, which is imported by GPU, and restores its position in local or shared memory on GPU. The parameter Pl is optimized to minimize data transmission between GPU and CPU and maximize the coalescence access when computing on GPU.

Secondly, as shown in Step 3 of both algorithms, the algorithms will set scores for restored positions according to the prior knowledge based on game rules, which is laid in shared memory on GPU and shared by all threads. For example, for the game Connect6, they are shapes of the game and their corresponding scores. For branch calculation in Algorithm 3, this part will recognize the shapes of the position and set the scores for empty points in order to find next move, while leaf calculation in Algorithm 4 will set the scores for the shapes and non-empty points to further evaluate. The calculation of scores depends on the gathering of AI information, which is closely related to game rules.

Thirdly, as shown in Step 8 of both algorithms, the algorithms will assemble the scores above and compute the returned results into the device memory on GPU, which will be sent to CPU as the results for the cycle in Algorithm 2. In Algorithm 4, the leaf calculation is simply adding up all evaluated points and shapes, and returning the results. Algorithm 3 is much complicated, where branch calculation should filter the unvalued moves and accumulate specific moves. In fact, different games and programs have their own strategy to get better selections. Particularly, as the game Connect6 needs two points in one move, therefore the algorithm needs to determine which one should be selected as the first.

Basically, the framework in Algorithm 3 and 4 can be used for most of games. Developers need to replace AI related part by their own functions.

5 Performance analysis

5.1 The Performance Model

The aim of the performance model is to analyze the effectiveness and efficiency of our algorithm. Our algorithm will generate a game tree T during its execution. We assume that the depth of T is d and the width is w. In our algorithm, for each cycle (Step 3 to Step 23 in Algorithm 2), we need to calculate l leaves or b branches in parallel. We denote the average execution time of calculating l leaves on GPU as $t_{GPU}(l)$. Similarly, we denote the average execution time of calculating b branches as $t_{GPU}(b)$. Then the total time of Algorithm 2 can be expressed as

$$T_{GPU} = t_O + m_l \times t_{GPU}(l) + m_b \times t_{GPU}(b) \quad (1)$$

where t_O is the overhead of parallel search algorithm including maintaining the game tree, m_l and m_b are loop numbers of calculating leaves and branches respectively. Obviously, we have following relationship

$$m_l \times l + m_b \times b \approx N_{GPU} \quad (2)$$

where N_{GPU} is the total number of nodes calculated by our algorithm. Meanwhile, for game tree T, we have

$$m_l \times l \approx m_b \times b \times w \quad (3)$$

which means that most of nodes calculated in our algorithm are leaves. From Equation 1, 2 and 3, we get

$$m_b \approx \frac{N_{GPU}}{(1+w) \times b} \quad (4)$$

$$m_l \approx \frac{N_{GPU}}{(1+\frac{1}{w}) \times l} \quad (5)$$

Then taking Equation 4 and 5 into Equation 1, we have

$$T_{GPU} = t_O + \frac{N_{GPU}}{(1+\frac{1}{w})l} \times t_{GPU}(l) + \frac{N_{GPU}}{(1+w)b} \times t_{GPU}(b) \quad (6)$$

Equation 6 gives the formal expression of execution time of Algorithm 2. In fact, the following holds for N_{GPU}:

$$N_{GPU} \leq N_{total} \quad (7)$$

where N_{total} is the total number of nodes in T. That is

$$N_{total} = \sum_{i=0}^{d} w^i = \frac{w^{d+1}-1}{w-1} \quad (8)$$

Since it is not easy to obtain the exact formal expression of N_{GPU}, we choose to analyze different cases of N_{total} to exploit properties of our algorithm. Next, we will analyze our algorithm in case of $N_{GPU} = N_{total}$.

5.2 Performance Analysis in Case of No Pruning

In case of $N_{GPU} = N_{total}$, all nodes of the game tree must be calculated and pruning technology has no effects on performance. Therefore, we can get the following relation by Equation 6

$$T^{NP}_{GPU} = t_O + \frac{N_{total}}{(1+\frac{1}{w})l} \times t_{GPU}(l) + \frac{N_{total}}{(1+w)b} \times t_{GPU}(b) \quad (9)$$

Replacing N_{total} in Equation 9 by Equation 8, we get

$$T^{NP}_{GPU} = t_O + \frac{\frac{w^{d+1}-1}{w-1}}{(1+\frac{1}{w})l} t_{GPU}(l) + \frac{\frac{w^{d+1}-1}{w-1}}{(1+w)b} t_{GPU}(b) \quad (10)$$

In practice, w is much larger than 1 (w is 30 to 50 in most algorithms), so we get a simple form for Equation 10 as

$$T^{NP}_{GPU} = t_O + \frac{w^d}{l} \times t_{GPU}(l) + \frac{w^{d-1}}{b} \times t_{GPU}(b) \quad (11)$$

From Equation 11, we can get following results:

Theorem *5.1:* If T, l and b are determined, in case of no pruning, the execution time of our algorithm is directly proportional to $t_{GPU}(l)$ and $t_{GPU}(b)$.

In addition, since w^d is much larger than w^{d-1}, we also have the following result:

Theorem *5.2:* In case of no pruning, the change of $t_{GPU}(l)$ affects the performance of our algorithm much more than $t_{GPU}(b)$. This implies that the change of l may affect the performance of our algorithm more than b.

5.3 Performance Analysis Considering Pruning

In fact, the case of $N_{GPU} = N_{total}$ occurs rarely in most games since the pruning technology can be used. That is, in most cases, we have $N_{GPU} < N_{total}$. Recalling Equation 6

$$T^{P}_{GPU} = t_O + \frac{N_{GPU}}{(1+\frac{1}{w})l} t_{GPU}(l) + \frac{N_{GPU}}{(1+w)b} t_{GPU}(b) \quad (12)$$

If all parameters other than N_{GPU} in Equation 12 are determined, we can simply Equation 12 as

$$T^{P}_{GPU} = t_O + k \times N_{GPU} \quad (13)$$

Since t_O can be omitted when the size of T is large enough, we can further simplify Equation 13 as

$$T^{P}_{GPU} \approx k \times N_{GPU} \quad (14)$$

From Equation 14, we can get following result:

Theorem *5.3:* In case of pruning, the execution time of our algorithm is directly proportional to N_{GPU}.

Note. The size of N_{GPU} depends on the design of GTS algorithms. It is well known that parallel search algorithms will calculate more nodes than needed. That is, some nodes in N_{GPU} nodes need not to calculate. In ideal conditions, there are a minimal number N_{MIN} of nodes needed to calculate to find a solution. The closer N_{GPU} to N_{MIN}, the better the performance of our algorithm is, and vice versa. However, it is hard to determine the theoretical bound of N_{MIN}. In practice, the number of nodes calculated by a serial GTS algorithm N_{serial} will be treated as a referee of N_{MIN}.

5.4 Effects of Game Complexities

In previous subsections, we analyzed how GPU architecture and our algorithm affect the performance of GTS. However, given the same architecture and algorithms, different games have different performance due to different values on N_{total}, w and d, namely the game tree complexity. In fact, differences on these parameters determine the game complexity of different games. Basically, game complexity can be measured by several factors such as *tree complexity*, *branching factor*, *average depth*,

TABLE 2: A Classification of Game Trees for Some Games

Class	Game	Depth	Width	Tree Complexity
Solved	Tic-tac-toe	9	4	10^5
	Sim	14	3.7	10^8
	Connect Four	36	4	10^{21}
Partially Solved	Chess	80	35	10^{123}
	Connect6	30	46000	10^{140}
	Havannah	66	240	10^{157}
Unsolved	Arimaa	92	17281	10^{402}
	Go(19×19)	150	250	10^{360}

computational complexity and etc. Also, even in the same level of game complexity, different games distinct from each other for their different rules and implementations, i.e., the branch and leaf calculations. Basically, according to their difficulty to solve and the AI level of its computer program, two-player computer games can be classified into three classes [39], as shown in Table 2.

Table 2 lists the complexities of some common games. In fact, as analyzed before, our node-based GTS algorithm focuses on *partially solved games*, which may have a reasonable solution via AI and parallel computing. Therefore we implement algorithms for two partially solved games Connect6 and Chess to verify their performance. Both these computer games have similar level of complexity of game rule and related high tree complexity with large branch factors or average depths.

6 Experiments and verifications

6.1 Experiment Settings

6.1.1 Machine Configurations

All experiments are performed on a machine equipped with two quad-cores CPUs Intel (R) Xeon (R) E5620 processors, and 24G host memory. Operating system is Scientific Linux release 6.0 (Carbon). Two GPU in our machine are NVIDIA Tesla C2050 with 15 multiprocessors, 480 cores and 3GB global memory. Each processor can simultaneously schedule at most 48 warps. The actual number of warps executed depends on the usage of resources such as registers and shared memory. We use CUDA toolkit 4.0 version to implement algorithms. When we use *nvcc* command provided by CUDA to compile our programs, we use the option *-DPROFILING* to do profiling to get the runtime usage of kernel resources.

6.1.2 Configurations of Connect6 and Chess

We verify our parallel GTS algorithm by redesigning two computer games, *Connect6* [31] [32] and *Chess* [24].

Connect6 is a two-player game similar to Gomoku, which is widely played around the world. For the game, two players, Black and White, alternately place two stones of their color, black and white respectively, on a 19×19 Go-liked board[3]. The winner is the first one who gets six or more stones in a row, horizontally, vertically or diagonally. We implement the Connect6 game based on our *Cloudict.Connect6* [36] program running on CPU, which won the gold medal of 16th Computer Olympiad hosted by the International Computer Games Association (ICGA) in 2011. Chess is a one of the most popular games played by people all over the world. It is also a two-player strategy board game played on a game board with 64 squares arranged in a 8×8 grid. The Chess program in our experiments is based on an open-source version called *Chessmate* [37], which is a pure CPU-based implementation.

In our experiments, we use two configurations for both games. The first one adopts the GPU-based GTS algorithm without the tree pruning. With this configuration, we can first verify the stand-alone capability of GPU to process massive nodes calculations in parallel. This case also helps us to investigate the possibility of utilizing GPU for computer games with low pruning efficiency. The second configuration will use classical pruning technologies as well as the parallel search algorithm on GPU to evaluate the practical performance.

The parameters used in our GTS algorithm are listed as follows. Generally, a game tree is determined by three parameters: the width w, the depth d and the initial position p_0. In the Connect6 experiments, for both configurations (i.e. with and without the tree pruning), we set $w = 48$, to limit the width of the game tree; for the configuration without the pruning, we set $d = 4$; otherwise we set $d = 5$. In the Chess experiments, we do not fix the width of the game tree; for the configuration without the pruning, we set $d = 5$; otherwise we set $d = 6$. For parameter p_0, we select the normal initial position in each game. That means, we select one black stone in the middle of the game board for Connect6 in our tests[4]; for Chess, it's just the initial board with each piece in its initial position.

6.2 Evaluations of Parallel Node Calculation on GPU

As discussed in Section 2, the node calculation includes the calculation for the leaf and the branch. We evaluate both programs to illustrate the performance benefits of the node calculation on GPU. As discussed in Section 5, the performance of the node calculation depends on the leaf number l and the branch number b. Before we present performance results, we give the performance metric for the node calculation as follows:

$$f_l = \frac{T_S^l}{T_{GPU}^l} \tag{15}$$

and

$$f_b = \frac{T_S^b}{T_{GPU}^b} \tag{16}$$

[3] Different with the game Go, in Connect6, the first Black player places just one stone for the first move.

[4] For Connect6, through experiments, we find that different initial positions except for extremely cases do not make differences to experiments results.

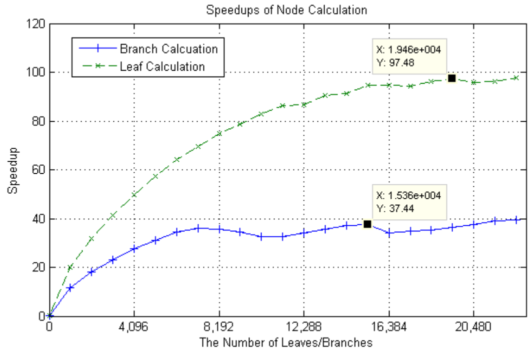


Fig. 5: Normalized Speedup when Changing Leaf/Branch Number for Connect6

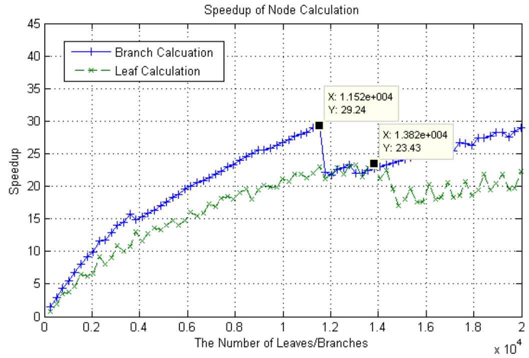


Fig. 6: Normalized Speedup when Changing Leaf/Branch Number for Chess

Here, T_S^l and T_S^b are the time of the leaf and the branch calculation in CPU using the serial algorithms, T_{GPU}^l and T_{GPU}^b are the corresponding time in GPU.

For the evaluation experiments here, we get the various numbers of nodes from one game tree, and calculate them on CPU serially and on GPU concurrently, respectively, to obtain the T_S and T_{GPU}. The node calculations include branch calculation and leaf calculation in the experiments. For Connect6 and Chess, we select the both initial positions as the two roots of the two game trees.

Fig. 5 and Fig. 6 illustrate the speedup over CPU algorithms of the leaf calculation and the branch calculation for Connect6 and Chess respectively, when changing the number of leaf nodes l or branch nodes b. We have observed three phenomena:

1) When the number of l or b increases, the normalized speedup also increases. For Connect6, we observe up to 37x and 97x speedup over CPU for the branch calculation and the leaf calculation, respectively. For Chess, the speedup is up to 29x and 23x for the branch and the leaf, respectively.
2) When the number of l or b reaches to a certain value, the normalized speedup will come to the threshold.
3) In Connect6, the speedup of the leaf calculation is much higher than that of the branch calculation; while, in Chess, the speedup of the leaf calculation is lower than that of the branch calculation.

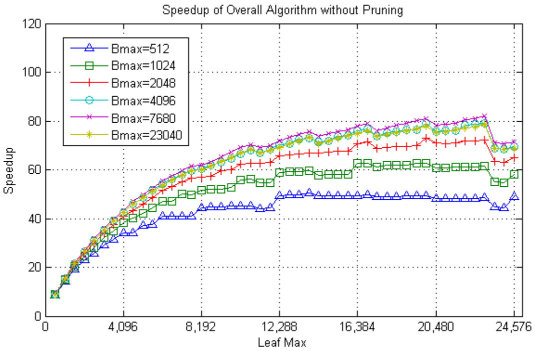


Fig. 7: Normalized Speedup When Changing Maximum Leaf Number for Connect6

It is easy to explain the phenomenon 1) and 2). When the number of l or b increases, more computation can be offloaded to GPU and accelerated by our algorithms. Therefore, no matter for the branch or the leaf calculation, the speedup will increase. When the number l or b reach to a threshold, most of the GPU cores have the task to compute in parallel. At this value, increasing l or b cannot improve parallelism of GPU cores. Therefore, f_l and f_b will not change significantly after that threshold. For the phenomenon 3), the reason is that in our algorithms, the results of the branches are sorted and then we select the first w moves during the calculation, i.e., Step 30 in Algorithm 2. The sorting brings more time cost for the branch calculation, leading to the lower speedup than that of the leaf calculation for Connect6.[5]

These experiments illustrate that the branch calculation and the leaf calculation can be accelerated by GPU. For the node calculation, there are a lot of branch divergence due to the complex rules of computer games. Although the branch divergence may reduce the performance of massive parallel computing on GPU, our experiments illustrate the enhanced performance, since the positions in a same game tree are very similar. For example, in 10-plies Connect6 game tree, at least more than 94% of positions are similar, which may significantly reduce divergences and improve the performance. The experiment also verifies the result in Theorem 5.1, that is, the high performance of node calculation does bring good overall performance in case of no pruning.

6.3 Overall Performance Evaluation Without Pruning

In this subsection, we will present the overall performance of Connect6 and Chess without the tree pruning, so as to verify the effectiveness of Theorem 5.1 and 5.2.

We first check how L_{MAX} and B_{MAX} affect overall performance of our algorithms without the tree pruning. Fig. 7 and Fig. 8 illustrate the speedup of Connect6 when we change the maximum leaf number and the maximum

[5] We always observe the speedup of the branch calculation and the leaf calculation, when the total number of nodes is greater than 39. As a result, for the later experiments, we set $B_{min} = L_{min} = 40$.

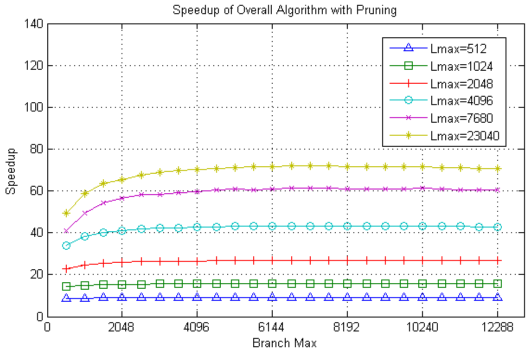


Fig. 8: Normalized Speedup When Changing Maximum Branch Number for Connect6

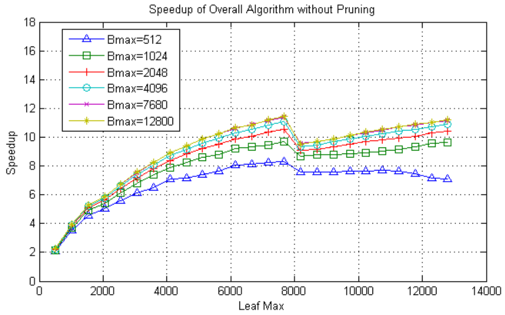


Fig. 9: Normalized Speedup When Changing Maximum Leaf Number for Chess

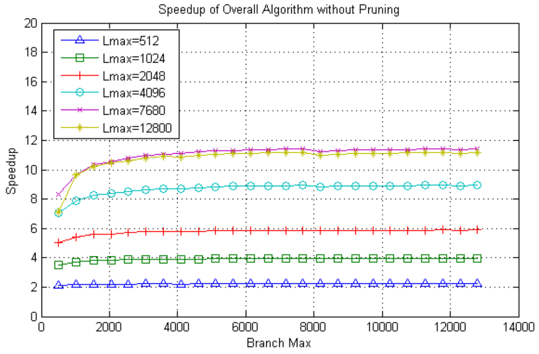


Fig. 10: Normalized Speedup When Changing Maximum Branch Number for Chess

branch number, respectively. In Fig. 7, different lines represent the speedups of different B_{MAX} values. We observe the obvious speedup for all B_{MAX} values. When L_{MAX} reaches a threshold, e.g., larger than 16384, the speedup will not change significantly in all configurations of B_{MAX}. In Fig. 8, we choose several L_{MAX} values and change the value of B_{MAX}. After the B_{MAX} reaches 4096, the speedup keeps almost same.

Fig. 9 and Fig. 10 illustrate the similar experiments for Chess. We also observe the similar tendency as those for Connect6. We observe up to 11.8x speedup for Chess, while the speedup for Connect6 is 70.8x. The different speedup in Connect6 and Chess comes from the different performance of branch and leaf calculations for these two games, as shown in Fig. 5 and Fig. 6. The speedup results indicate that when our node-based GTS algorithm is not configured with the tree pruning, GPU can be used to achieve better performance.

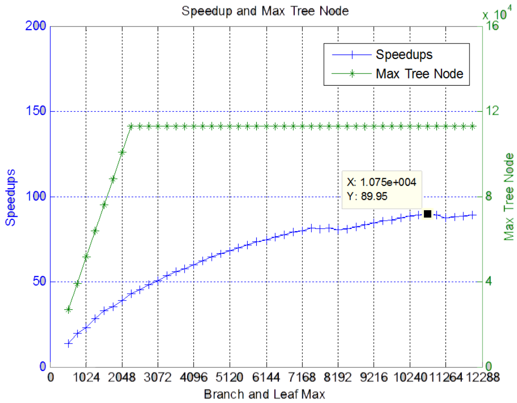


Fig. 11: Normalized Speedup and Max Maintained Tree Nodes in Connect6

As illustrated by Theorem 5.2, the performance of the leaf calculation affects overall performance more than the branch calculation. Fig. 8 and Fig. 10 have verified our analysis, since we observe more obvious differences between the lines representing numbers of leaves than those in a same line with different numbers of branches.

In our parallel search algorithms, there are $l = L_{MAX}$ and $b = B_{MAX}$ at most time because of the exponential growth of the branch and leaf nodes. The number of leaves is around w times of that of branches because of exponential growth of the tree. As presented in the previous paragraph, the performance of the leaf calculation affects the overall performance more than the branch calculation. That means L_{MAX} can determine the overall performance mostly. Therefore, in the following experiments, we set $B_{MAX} = L_{MAX}$ and vary L_{MAX}.

For Connect6, compared with the serial search on CPU, the normalized speedup is shown in Fig 11. The speedup of the overall performance is increasing with the increased number of L_{MAX}, and will meet the threshold when most of CPU and GPU are used. When $B_{MAX} = L_{MAX} = 10750$, we get the maximal speedup which is 89.95x over the serial algorithm on CPU. For Chess, compared with the serial search on CPU, the normalized speedup is shown in Fig 12. The speedup of the overall performance is also increasing with the increased number of L_{MAX} as that in Connect6. When $B_{MAX} = L_{MAX} = 7680$, we get the maximal speedup which is 11.43x over the serial execution on CPU.

In this experiment, we also evaluate the maximum number of tree nodes maintained in the memory during the execution of applications, denoted as the green line shown in Fig 11 and Fig 12. Since our node-based GTS algorithm maintains the tree structure in the memory during the tree search, higher parallelism (i.e., larger B_{MAX} and L_{MAX}) will lead to more nodes of the game tree put into the memory. As illustrated in Fig. 4(4), during the execution of parallel search, there are 22 nodes

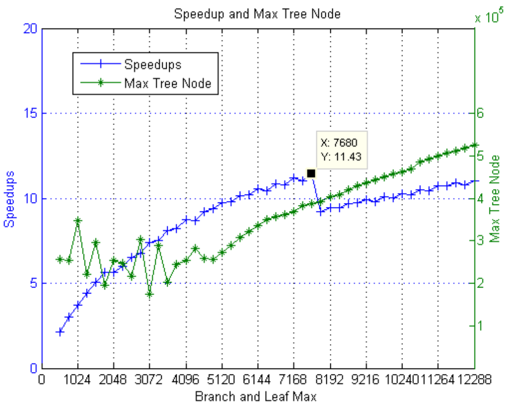


Fig. 12: Normalized Speedup and Max Maintained Tree Nodes in Chess

maintained in the memory. For Connect6, as shown in Fig 11, the number of nodes in the memory is linear to B_{MAX} and L_{MAX}. The reason is that we set the width of the game tree fixed $w = 48$ in our experiments, which represents the common case of the game. In addition, the number of the nodes reaches to the threshold when $B_{MAX} \geq 2304$. Because the depth of the tree set to $d = 4$ in the experiment is small enough, the algorithm easily reaches the leaf ply and its maximum value, leading to all possible situations of the game can be hold in the tree structure. For Chess, as shown in Fig 12, the number of the nodes in the memory is variable when the B_{MAX} and L_{MAX} are smaller than 5000. It is because the rule of the Chess permits much less branches (i.e., b in the algorithm) than that of Connect6. The figure also illustrates the number of the nodes will become linear to B_{MAX} and L_{MAX} after a threshold, but not level off a peak value. It is because the branches in the Chess program are much deeper than those in Connect6, leading to the number of the nodes kept in the memory increasing with the B_{MAX} and L_{MAX}.

6.4 Overall Performance Evaluation With Pruning

The experiments without pruning are ideal to verify the optimal performance of GTS using our method. However, in practice, the pruning technique is used to accelerate GTS by reducing the amount of nodes to be calculated. In this subsection, we evaluate our method with pruning. Based on the idea of the alpha-beta pruning discussed in Section 2, we implement the pruning function for both Connect6 and Chess. Fig. 13 and Fig. 14 show the overall performance of our algorithm for Connect6 and Chess with pruning, respectively.

For Connect6, compared with the serial search on CPU with pruning, the speedup is up to 10.58x when $B_{MAX} = L_{MAX} = 7424$. The corresponding maximal number of tree nodes maintained in the memory, shown in the green line, is approximately linear to the B_{MAX} and L_{MAX}. For Chess, compared with the serial search on CPU with pruning, the speedup is up to 7.26x when $B_{MAX} = L_{MAX} = 7680$. The maximal number of tree nodes in the memory is also approximately linear to the B_{MAX} and L_{MAX} when they are larger than 5120.

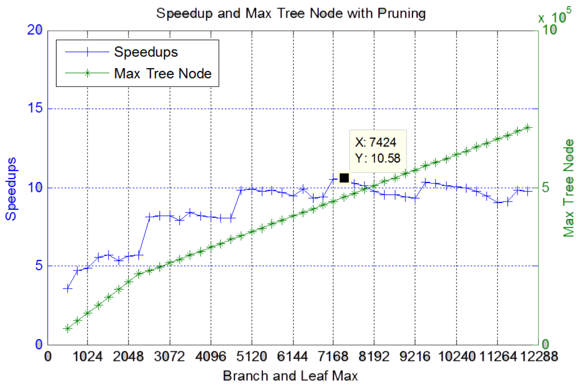


Fig. 13: Normalized Speedup and Max Maintained Tree Nodes in Connect6 (with Pruning)

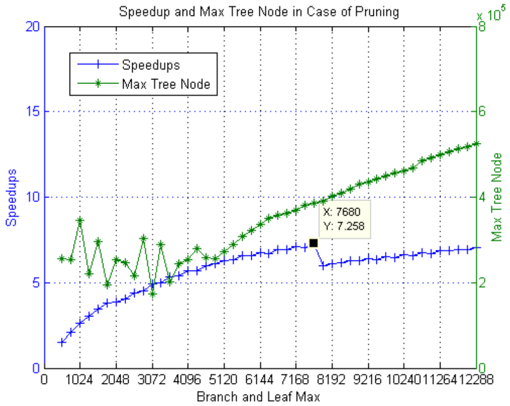


Fig. 14: Normalized Speedup and Max Maintained Tree Nodes in Chess (with Pruning)

These two figures illustrate when the parameters of applications change, our method can also achieve satisfactory performance. Compared with the performance without pruning, the speedup of both applications with pruning decreases significantly. This phenomenon is related to Theorem 5.3, which illustrates the performance of the game tree search depends on the number of the nodes calculated. In Table 3, we list out the number of the nodes calculated in our method with pruning, the corresponding number in the serial implementation on CPU with pruning, and the total number of the game tree for these two applications. For Connect6, N_{serial} is 2,653,180, while N_{GPU} is 24,503,857. This illustrates that the same pruning algorithm will cut down much more branches and leaves in the serial implementation on CPU than those in the parallel version on GPU. This is related to the algorithm of the alpha-beta pruning illustrated in Fig. 2, which includes four steps to cut down the branches and leaves by the rule of minimax. They are totally same with the execution flow in the serial implementation on CPU. However, in the parallel implementation on GPU, we issue the calculation of

TABLE 3: The Number Of Nodes

Parameters	Connect6	Chess
N_{GPU}	24,503,857	5,104,063
$N_{serial}(N_{MIN})$	2,653,180	3,694,481
N_{total}	260,225,329	5,104,063

nodes in a same layer in parallel. For example, in Fig. 2, the calculation of node b, c, d will be executed in parallel; and their values will be returned back until the max depth is reached, e.g., the depth in our experiments is set to 5 for the pruning as described in Section 6.1.2. Due to the loss of the knowledge of other nodes, the pruning algorithm in our parallel method cannot cut down some branches and leaves, e.g., the node h cannot be cut down because the pruning algorithm doesn't know the return value of node b is 6, even the value of node g is obtained. As a result, the algorithm of the alpha-beta pruning can cut down much more nodes in the serial game tree search than the parallel version, which leads to the decreased speedup of our method when the pruning is enabled. For Chess, the speedup decreases from 11.43x without pruning to 7.26x with pruning, which is better than that of Connect6, from 89.95x without pruning to 10.58x with pruning. The numbers of nodes calculated in the serial CPU version and our parallel GPU version are also illustrated in Table 3. For Chess, N_{serial} is 3,694,481, while N_{GPU} is 5,104,063. The number of N_{GPU} is same with the total number, which means the pruning algorithm cannot cut down any node for Chess in the configurations we used, i.e, the search depth d and the search width w. Due to much more complicated rule of Chess than Connect6, even the serial pruning algorithm with the priori knowledge cannot cut down as many nodes as those in Connect6, leading to the speedup decreased less than Connect6.

6.5 Analysis on Performance Fluctuation

From the observation on Fig. 6, 9, 12 and 14, we see that the speedup of our GPU-based Chess algorithm has fluctuation in the middle area. Here, the speedup drop comes from the task scheduling mechanism of GPU architecture. As introduced in Section 6.1.1, in our experiments, the GPU chip has 15 processors, each of which can run 32 threads (such a set of threads is called a *warp*) concurrently. Each GPU processor can schedule at most 48 warps (i.e. 1536 threads) simultaneously, while the actual number of warps scheduled depends on how they exclusively use limited kernel resources such as shared memory and registers at runtime. For instance, in Fig. 6, for branch calculation, each GPU processor can only schedule 24 warps because each thread (i.e. the branch calculation function shown in Algorithm 3) exclusively occupies 30 registers during its execution. In fact, if the number of registers occupied by a thread is less than 20, the GPU processor can schedule the maximal number of warps (i.e. 48 warps) to execute. Therefore, in our experiments, at runtime the GPU chip with 15 processors can schedule 11520 (i.e., 32*15*24) branch nodes simultaneously. When the number of branch nodes submitted to the GPU exceeds 11520, the branch nodes other than the first 11520 scheduled nodes have to wait for the next round of scheduling of 11520 nodes. Due to the waiting time of these branch nodes, the speedup of our algorithm has significant drop when the number of branch nodes exceeds 11520. This figure also illustrates after the significant drop, if we continue increasing the number of the branch nodes, the normalized speedup increases. This is because the warps to handle the nodes larger than 11520 can be executed in parallel after waiting for the finish of the warps for previous nodes. However, for Connect6, since the branch and leaf node calculation uses less than 20 registers at runtime, the GPU processor can always schedule the maximal number of warps. Therefore, the performance of Connect6 has very small fluctuation.

The results shown in Fig. 9, 12 and 14 can also be explained by the above reason. That is, if the number of the nodes exceeds the processing capacity of the GPU due to their usage of kernel resources, the speedup will have some fluctuation. For example, for experiments shown in Fig. 12 and 14, the GPU processor can only schedule 16 warps since the branch and leaf node calculation will occupy 30 and 33 registers respectively. Therefore, the speedup drop happens when both of the branch and leaf nodes reaches to 7680 (i.e., 32*15*16).

7 Conclusions and Future Work

In this paper, we provide a node-based GTS algorithm on the system with GPU. The performance model of this algorithm is presented and the relationships between the performance and the key parameters are revealed. With our parallel GTS algorithm, we redesign two computer games, Connect6 and Chess, to evaluate the effects of our method. In the future, we will continue our research in the following directions. 1) Improve the GTS algorithm and extend it to the large-scale clusters with GPU; 2) Investigate more pruning algorithms with the GPU-based GTS algorithm for Connect6 and Chess; 3) Adopting our method to other games, such as *Go*, to examine the effectiveness and efficiency of our idea.

Acknowledgments

This work is supported in part by the the Hi-Tech Research and Development (863) Program of China (Grant No. 2013AA01A212, 2013AA01A209).

References

[1] A. Bleiweiss, "GPU accelerated pathfinding", The 23rd ACM Symposium on Graphics Hardware, 2008, pp. 65-74.

[2] P. Borovska and M. Lazarova, "Efficiency of parallel minimax algorithm for GTS", The 2007 international conference on Computer systems and technologies, 2007, pp. 14:1-14:6.

[3] M. G. Brockington, A Taxonomy Of Parallel Game-Tree Search Algorithms, 1996.

[4] M. G. Brockington and J. Schaeffer, ”APHID: Asynchronous Parallel Game-Tree Search”, Journal of Parallel and Distributed Computing, 60(2):247-273, 2000.

[5] M. G. Brockington, M. G. Brockington, T. A. Marsland, J. Samson, and M. Campbell, ”Asynchronous Parallel Game-Tree Search”, Journal of Parallel and Distributed Computing, 1998.

[6] M. S. Campbell and T. A. Marsland, ”A comparison of minimax tree search algorithms”, Artificial Intelligence, 20(4):347-367, 1983.

[7] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, ”GPU Cluster for High Performance Computing”, The 2004 ACM/IEEE conference on Supercomputing, 2004, pp. 47-53.

[8] R. Feldmann, P. Mysliwietz, and B. Monien, ”Distributed GTS on a massively parallel system”, Data structures and efficient algorithms, B. Monien and T. Ottmann, Eds. Springer Berlin / Heidelberg, 594:270-288, 1992.

[9] R. Feldmann, B. Monien, and S. Schamberger, ”A Distributed Algorithm to Evaluate Quantified Boolean Formulae”, 2000, pp. 285-290.

[10] H. Hopp and P. Sanders, ”Parallel GTS on SIMD machines”, Parallel Algorithms for Irregularly Structured Problems, A. Ferreira and J. Ro-lim, Eds. Springer Berlin / Heidelberg, 980:349-361, 1995.

[11] X. Huo, V. T. Ravi, W. Ma, and G. Agrawal, ”Approaches for parallelizing reductions on modern GPU”, International Conference on High Performance Computing (HiPC), 2010, pp. 1-10.

[12] R. M. Karp and Y. Zhang, ”On parallel evaluation of game trees”, The first annual ACM symposium on Parallel algorithms and architectures, 1989, pp. 409-420.

[13] G. Karypis and V. Kumar, ”Unstructured tree search on SIMD parallel computers”, IEEE Transactions on Parallel and Distributed Systems, 5(10):1057-1072, 1994.

[14] A. Kishimoto and J. Schaeffer, ”Distributed game-tree search using transposition table driven work scheduling”,The 2002 International Conference on Parallel Processing, 2002, pp. 323-330.

[15] D. E. Knuth and R. W. Moore, ”An analysis of alpha-beta pruning”, Artificial Intelligence, 6(4):293-326, 1975.

[16] D. Luebke, ”CUDA: Scalable parallel programming for high-performance scientific computing”, The 5th IEEE International Symposium on Bi-omedical Imaging: From Nano to Macro, 2008, pp. 836-838.

[17] W. Ma and G. Agrawal, ”An integer programming framework for optimizing shared memory use on GPU”, International Conference on High Performance Computing (HiPC), 2010, pp. 1-10.

[18] V. Manohararajah, ”Parallel alpha-beta search on shared memory multiprocessors”, 2001.

[19] T. A. Marsland and F. Popowich, ”Parallel Game-Tree Search”, IEEE Transactions on Pattern Analysis and Machine Intelligence, 7:442-452, 1985.

[20] J. Nickolls, I. Buck, M. Garland, and K. Skadron, ”Scalable Parallel Programming with CUDA”, Queue, 6(2):40-53, 2008.

[21] K. Rocki and R. Suda, ”Large-Scale Parallel Monte Carlo Tree Search on GPU”, IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011, pp. 2034-2037.

[22] K. Rocki and R. Suda, ”Parallel Minimax Tree Searching on GPU”, Parallel Processing and Applied Mathematics, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Springer Berlin / Heidelberg, 6067:449-456, 2010.

[23] J. Schaeffer, ”Distributed game-tree searching”, Journal of Parallel and Distributed Computing, 6(1):90-114, 1989.

[24] C. E. Shannon, ”Programming a computer for playing chess”, Philosophical Magazine Series 7, 41(314):256-275, 1950.

[25] J. Soman, M. K. Kumar, K. Kothapalli, and P. J. Narayanan, ”Efficient Discrete Range Searching primitives on the GPU with applications”, 2010 International Conference on High Performance Computing (HiPC), 2010, pp. 1-10.

[26] D. Strnad and N. Guid, ”Parallel Alpha-Beta Algorithm on the GPU”, Journal of Computing and Information Technology, 19(4):269-274, 2011.

[27] J. Tromp and G. Farneback, ”Combinatorics of Go”, Computers and Games, H. van den Herik, P. Ciancarini, and H. Donkers, Eds. Springer Berlin / Heidelberg, 4630:84-99, 2007.

[28] H. J. van den Herik, J. W. H. M. Uiterwijk, and J. van Rijswijck, ”Games solved: Now and in the future”, Artificial Intelligence, 134(1-2):277-311, 2002.

[29] J. C. Weill, ”The ABDADA distributed minimax search algorithm”, ACM 24th annual conference on Computer science, 1996, pp. 131-138.

[30] K. Crowley and R. S. Siegler, ”Flexible strategy use in young children’s tic-tac-toe”, Cognitive Science, 17(4):531-561, 1993.

[31] I.C. Wu and D.Y. Huang, ”A New Family of k-in-a-Row Games”, Advances in Computer Games, H. van den Herik, S.-C. Hsu, T. Hsu, and H. Donkers, Eds. Springer Berlin / Heidelberg, 4250:180-194, 2006.

[32] C. Xu, Z. M. Ma, J. Tao, and X. Xu, ”Enhancements of proof number search in connect6”, Control and Decision Conference, 2009, pp. 4525-4529.

[33] Hyatt, R.M. ”The Dynamic Tree-Splitting Parallel Search Algorithm”, ICCA Journal, 20(1):3-19, 1997.

[34] Hyatt, R.M. ”A High-Performance Parallel Algorithm to Search Depth-First Game Trees”, Ph.D. Thesis, 1988, University of Alabama, Birmingham.

[35] ”Connect6 in Computer Olympiad”, http://www.grappa.univ-lille3.fr/icga/game.php?id=18

[36] ”Cloudict”, https://github.com/lang010/cloudict

[37] ”Chessmate”, https://github.com/pate/chessmate

[38] Murray Campbell, A. Joseph Hoane Jr. and Feng-hsiung Hsu, ”Deep Blue”, Artificial Intelligence, 134(1-2):57-83, 2002.

[39] Yew Jin Lim, ”On Forward Pruning in Game-Tree Search”, Ph.D. Thesis, National University of Singapore, 2007.

Liang Li is a master degree candidate in Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS) and will be graduated in 2013. He is a student member of the IEEE and the IEEE Computer Society and interested in high performance computing, distributed computing and artificial intelligence. He has published three papers as first author in conferences related to these research areas.

Hong Liu is an Assistant Professor in Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS). His research interests include high performance computing and distributed computing. Dr. Hong Liu received his Ph.D. degree from Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS) in 2010. He is a member of the IEEE and the IEEE Computer Society.

Hao Wang is a Research Associate in the Department of Computer Science at Virginia Polytechnic Institute and State University. His research interests include high performance computing, parallel computing architecture, and big data analytic. He received his Ph.D. degree from Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS) in 2008.

Taoying Liu is an Associate Professor at the Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS). Her research interests include distributed system and parallel computing. Dr. Taoying Liu re-ceived her Ph.D. degree from Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS) in 2009. She is a member of the IEEE and the IEEE Computer Society.

Wei Li is an Associate Professor at the Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS). His research interests include distributed system, parallel computing architecture, and big data analytic. He has published more than 40 papers in conferences and journals related to these research areas. Dr. Wei Li received his Ph.D. degree from Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS) in 2008. He is a member of the IEEE and the IEEE Computer Society.