CUDA Implementation of Computer Go Game Tree Search

Christine Johnson*, Lee Barford*†, Sergiu Dascalu*, and Frederick Harris, Jr.*
*Department of Computer Science and Engineering, University of Nevada, Reno
†Keysight Laboratories, Keysight Technologies
email: c_johnson72@hotmail.com, lee_barford@ieee.org, dascalus@cse.unr.edu,Fred.Harris@cse.unr.edu

***Abstract*—Go is a fascinating game that has yet to be played well by a computer program due to its large board size and exponential time complexity. This paper presents a GPU implementation of PV-Split, a parallel implementation of a widely used game tree search algorithm for two-player zero-sum games. With many game trees, it often takes too much time to traverse the entire tree, but theoretically, the deeper the tree is traversed, the more accurate the best move found will be. By parallelizing the Go game tree search, we have successfully reduced the computation time, enabling deeper levels of the tree to be reached in smaller amounts of time. Results for the sequential and GPU implementations were compared, and the highest speedup achieved with the parallel algorithm was approximately 72x at 6 levels deep in the game tree. Although there has been related work with respect to game tree searches on the GPU, no exact best move search algorithms have been presented for Go, which uses significantly more memory due to its large board size. This paper also presents a technique for reducing the amount of required memory from previous game tree traversal methods while still allowing each processing element to play out games independently.**

I. Introduction

Computer Go has been a popular research topic for over 50 years [1] due to its classification as a PSPACE-hard problem, meaning it is at least as hard as the most difficult problems in the PSPACE class, and it has not been proven to be solved using a polynomial amount of space [2]. The characteristic of this problem that attributes to its exponential time complexity is the number of possible games is $\approx 10^{171}$ [3], and a program needs to check all of these games to determine which one yields the best score for a player. Thus, running multiple of these checks in parallel is a reasonable way of reducing the computation time.

Graphics Processing Units (GPUs) have become popular devices for parallel computing due to their low price to performance ratio. NVIDIA has allowed programmers to take advantage of their GPUs for general-purpose computing with the CUDA platform. CUDA C has been one of the most successful languages ever designed for parallel computing [4], making it an optimal choice for reducing program execution time by accelerating computation with the GPU.

The remainder of this paper is structured as follows. Section III provides the necessary background information, such as the rules for playing Go and game tree search methods. Section III references related research on the Go game tree search and parallel game tree searches. Section IV discusses the GPU implementation of the Go game tree search. Section V compares the sequential and parallel results, and explains analyses of those results. Finally, Section VI concludes and discusses future work.

II. Background

Although researchers have been studying the complexity of Computer Go since 1962, the game of Go was invented in China around 2300 B.C.. The first program that was capable of beating a novice player was written in 1968 [1]; however, there has yet to be a program written that can beat any expert player [5]. Such a program is challenging for a number of reasons: the search space of all possible moves is very large, it is difficult to construct long-term strategies, and determining the end of the game is not intuitive in a program [6]. The latter two reasons are the most difficult to address, so most research is aimed at speeding up the necessary computation needed to search the possible move space.

Go is often compared to Chess, due to them being similar with regards to being two-player strategy games [3]. However, Chess is considered a *solved* game because a Chess program was written that successfully defeated the world champion [7]. The search space of Chess is estimated to be $\approx 10^{70}$ where the search space of Go is estimated to be $\approx 10^{171}$ [3]. Additionally, Chess has a distinct *end game*, such as a checkmate state, where Go does not [6], as mentioned previously.

A. Rules

Go has many rules for specific board states; however, only the basic rules and those relevant to the presented research will be listed. The following rules are referenced from [1], [6].

Standard Go is played on a 19x19 board, creating 361 possible *intersections*, or places for players to place their stones. There are two players, one black and one white, who take turns placing their stones on the intersections of the game board. The black player has 181 stones and white has 180, the sum of them equaling the number of intersections. Stones of the same color that occupy consecutive intersections on the board are called *blocks*. The number of *liberties* for a stone is the number of neighboring intersections that are not occupied by any stones. A stone can have a maximum amount of four liberties. When the opponent's stones occupy all of a stone's liberties, the stone is considered *captured* by the opponent, and is removed from the board. Blocks of stones can be captured

as well if the entire block is surrounded by the opponent's stones.

The black player always goes first, and *komi* applies, meaning the white player receives a small amount of extra points on the final score since he or she is at the disadvantage of going second.

1) Objective: The objective of the game is to control the most territory. Stones are strategically placed to protect stones from being captured, and to capture the opponent's stones.

2) Scoring: There exist two variants of scoring methods: *territory scoring*, which count the surrounded territory plus the number of captured stones, and *area scoring*, which count the surrounded territory plus the *alive* stones on the board. For this research the latter is used. Contrary to standard Go rules but for simplicity, alive stones are just those that have not been captured. *Surrounded territory* is defined as blocks of empty intersections adjacent to stones of a single player.

3) End Game: The game is over when both players pass consecutively because they can no longer make any moves. Since it is difficult to detect when humans would surrender, in this research the game ends after a number of moves equal to the number of stones has been made.

B. Game Tree

Like most two-player games, Go program implementations typically use a game tree to store all possible moves that can be made from the remaining set of moves to choose from. The aim of game tree traversal algorithms is to allow the player to make the best move. This is achieved by traversing the tree to observe how each combination of moves would end a game, allowing the player to choose the move that will yield the best score. In the tree, the root represents the current game state, and all other node represents possible game states. Consecutive game states are reached by making a move in the game [8]. The children of each node represent all remaining moves that can be made after the move represented by the node, and some evaluation function is used at each leaf to determine the score for the player by ending the game with this combination of moves [9]. No algorithm exists that can traverse the entire game tree for Go in a reasonable amount of time [1], [9].

1) Tree Traversal Methods: The following are descriptions for the most common methods for game tree traversal, which all use Depth-First Search [8]. Each consecutive method is an enhancement on the previous, leading up to the method used in this research.

Minimax: Minimax tree search or variations of it are commonly used in two-player games because they attempt to maximize the score of the player while also trying to minimize the score of the opponent. All the even levels in the tree represent moves for the current player, and thus contain the moves at which the player wants to maximize its score. All the odd levels in the tree represent moves for the opponent, or moves where the player wants to minimize the score [8]. Starting at the leaf nodes and choosing best and worst nodes on the even and odd levels, respectively, yields the best possible move [1]. The NegaMax implementation has a single maximize function that it uses the negation of when minimizing [9].

Alpha-Beta Pruning: Alpha-Beta Pruning is an enhancement to Minimax that yields the same move, but finds it more efficiently by pruning away entire branches of the tree not worth exploring, thus reducing the search space [9]. A variable α represents the worst possible score the maximize player, and a variable β represents the worst possible score for the minimize player. The variables are updated if better values for each are found. Any node with a value below the current α will not be chosen by the maximizer, so they are pruned. Similarly, any nodes with values greater than β will not be chosen by the minimizer, so they are pruned [1].

Principal Variation Search: Principal Variation Search (PVS), also known as NegaScout, is an enhancement to Alpha-Beta pruning that finds the same move more efficiently by reducing the size of the search window, thus pruning more of the tree branches. The algorithm assumes the tree is ordered in such a way that the best move out of the children of a node will be the left-most child. The path that leads to the best move is the *principal variation* [10]. At each node, if the first child of that node does not fall into the minimal search window, then the current node is pruned. In order for this algorithm to be most efficient, the best move should be explored first, followed by the next best move, etc. Iterative-deepening and transposition tables are enhancements that address move-reordering [9]. If the tree is poorly ordered, each sub-tree that is better than its elder siblings must be searched again, making it potentially less efficient than the Alpha-Beta pruning [10]. The pseudocode for PVS is shown in Algorithm 1.

Iterative-Deepening: Iterative-deepening is an enhancement to Depth-First Search, which sets the max depth for traversal to some shallow level initially, and re-traverses the tree after increasing the depth each time [1]. This is beneficial for a number of reasons; one of them being that information from previous iterations can be stored so those levels do not actually need to be traversed again in each iteration [6]. A second reason is this allows us to essentially convert a Depth-First Search into a Breadth-First Search without the high memory requirements of BFS. This is advantageous with game trees because frequently a timeout value is used to stop the search, and the best move found when the cut-off was reached is returned. Checking all the nodes in a level rather than the nodes explored by ordinary DFS allows a "reasonably" good move to be found before the cut-off [1]. Finally, since a transposition table is not currently used, iterative-deepening assists with move-reordering. After each iteration of the search, the moves are reordered from best to worst so the algorithm can prune more efficiently on the next iteration.

C. Heuristic

As stated previously, one of the challenges of programming Go is the lack of a good evaluation function [1]. Common heuristics for Go include aiming to achieve the following: maximizing the number of liberties and the number of stones

Algorithm 1 PVS

```
1: function PVS(side, remainingLevels, alpha, beta)
2:     if remainingLevels = 0 then
3:         return evaluateLiberties(side)
4:     end if
5:     bestScore ← −∞
6:     n ← beta
7:     Generate successor moves
8:     for each move in successor moves do
9:         make move
10:        score ← -PVS(-side, remainingLevels - 1,
11:                     -n, -alpha)
12:        if score > bestScore then
13:            if n = beta or remainingLevels ≤ 2 then
14:                bestScore ← score
15:            else
16:                bestScore ← -PVS(-side,
17:                                 remainingLevels-1,
18:                                 -beta, -score)
19:            end if
20:        end if
21:        if score > alpha then
22:            alpha ← score
23:        end if
24:        undo move
25:        if alpha ≥ beta then
26:            return alpha
27:        end if
28:        n ← alpha + 1
29:    end for
30:
31:    return alpha
32: end function
```

on the board, avoiding moves on the edges, connecting stones, and making two *eyes*, which guarantee the life of a block of stones [6]. All of these goals excluding connecting stones and making eyes can be addressed essentially by maximizing liberties.

III. Related Work

For over 50 years, Go has been an intriguing research topic. Research has been performed involving the use of neural-networks for supervised learning [1] as well as Monte-Carlo methods, which choose the best move out of a set of randomly selected paths in the game tree to obtain decent results [11]. This research focuses on related work done to optimize game tree search for an exact best move using variations of Alpha-Beta Search.

Van der Werf *et al.* [6] solved Go for any possible opening move on a 5x5 board in 2002. The authors incorporated the five goals for their heuristic function discussed previously in Section . Assumptions were made that may only hold for smaller sized boards. The best move was found 23 levels deep in the game tree in approximately 4 hours. Future work included solving Go on 6x6 and 7x7 boards.

Strnad *et al.* [12] implemented PV-Split on the GPU for the game Reverse, also known as Othello in 2011. The authors designed an iterative algorithm, since GPU recursion was not supported on NVIDIA cards at that time. Future work included implementing transposition tables and iterative deepening to improve the algorithm.

Elnaggar *et al.* [9] did comparisons of the variations of game tree traversal, such as Minimax, Alpha-Beta Search, and Principal Variation Search. Results for a subset of the methods were presented; implemented sequentially and in parallel using MPI in 2014. Future work included using the OpenCL or CUDA libraries for GPU implementations.

Li *et al.* [8] designed a parallel algorithm for game tree traversal for Connect6 and Chess that was ran on the GPU in 2014. As opposed to using a variation of Alpha-Beta Search, the authors use a node-based parallel algorithm; meaning sets of nodes rather than sub-trees are assigned to each GPU processor to avoid the complexities involved with tree splitting. Future work included running the presented method on multiple GPUs, and on larger search spaces such as the Go game tree to observe the effectiveness and efficiency of the method.

Elnaggar *et al.* [13] designed a robot that can autonomously play Checkers with a human in 2014. The Checkers game tree was traversed using an iterative-deepening Alpha-Beta Search that was ran on the GPU. The novel feature of the presented method is the GPU kernel recursion, which was recently made available by NVIDIA in CUDA 5.0 on devices of Compute Capability 3.5 or higher [14]. Future work included using the presented method to solve more complex games like Go and Stratego.

IV. GPU Method

The deployed GPU implementation is a parallel version of the Alpha-Beta algorithm called PV-Split, which was first presented by [10]. This algorithm was chosen because it allows for high parallelism opportunities, and the data dependencies are minimal and manageable on the GPU. The following is a more detailed description of the algorithm.

A. PV-Split

If the game tree is naively divided into sub-trees that are processed in parallel, it is likely that redundant nodes or nodes that would pruned in the sequential algorithm will be explored due to the search window not being set properly [10]. PV-Split was designed based upon how the sequential Alpha-Beta algorithm searches the tree with optimal move-ordering. As stated previously, optimal move-ordering is defined as having the moves in the tree ordered from best to worst, implying the principal variation is the left-most path. PV-Split starts by traversing the left-most path sequentially to initialize the Alpha-Beta search window. Once the cut-off level for this iteration has been reached, the remaining siblings in each level are processed in parallel as the algorithm recursively goes back up the tree [10]. Figure 1 shows how PV-Split would divide the game tree among multiple processing elements.

The minimal search space is achieved when the best α value for each level is possessed by each processing element [10]. Alpha-beta windows are stored globally for each level of the tree, so if one processing element finds the best move in a

Algorithm 2 PV-Split

```
1: function PVSPLIT(side, remainingLevels, alpha, beta)
2:     if remainingLevels = 0 then
3:         return evaluateLiberties(side)
4:     end if
5:
6:     find the left-most child
7:     make the move
8:     score ← -PVSPLIT(-side, remainingLevels - 1,
9:                      -beta, -alpha)
10:    if score > alpha then
11:        alpha ← score
12:        updateAlphaBeta(side, depth, score)
13:    end if
14:    undo move
15:    if alpha ≥ beta then
16:        return alpha
17:    end if
18:
19:    divide all threads among the remaining siblings
20:    while move < totalMoves do
21:        make move
22:        score ← -TREESPLIT(-side,
23:                           remainingLevels - 1,
24:                           -beta, -alpha)
25:        if score > alpha then
26:            alpha ← score
27:            updateAlphaBeta(side, depth, score)
28:        end if
29:        undo move
30:        if alpha ≥ beta then
31:            return alpha
32:        end if
33:
34:        move ← next available move
35:
36:    end while
37:
38:    return alpha
39: end function
```

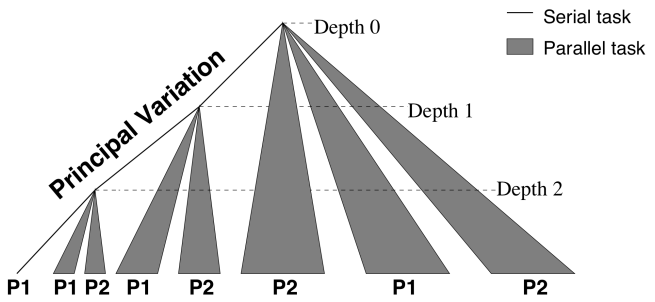

Fig. 1: PV-Split Search using Two Processing Elements.

Algorithm 3 Tree-Split

```
1: function TREESPLIT(side, remainingLevels, alpha, beta)
2:     if remainingLevels = 0 then
3:         return evaluateLiberties(side)
4:     end if
5:
6:     divide available threads among remaining moves
7:     while move < totalMoves do
8:         make move
9:         score ← -TREESPLIT(-side,
10:                           remainingLevels - 1,
11:                           -beta, -alpha)
12:        if score > alpha then
13:            alpha ← score
14:            updateAlphaBeta(side, depth, score)
15:        end if
16:        undo move
17:        if alpha ≥ beta then
18:            return alpha
19:        end if
20:
21:        move ← next available move
22:
23:    end while
24:
25:    return alpha
26: end function
```

level, it atomically updates the window for that level and the rest of the moves can be pruned from the search space [12]. Similar to PVS, PV-Split is most efficient with optimal move-ordering, thus iterative-deepening is used to reorder the moves at each iteration.

The GPU implementation uses two recursive device functions to perform the traversal of the left-most path and the tree splitting. The pseudocode for the device functions is shown in Algorithms 2 and 3.

PV-Split suffers from one main disadvantage: the sub-trees are not guaranteed to be of equal size due to the pruning, and PV-Split does not enforce any type of load-balancing. If one processing element takes longer to complete its sub-tree, the remaining processing elements are forced to wait until it completes, lessening the efficiency of the system.

B. Memory Optimization

Processing sub-trees in parallel allows for the play out of multiple games simultaneously, thus requiring multiple game boards. A common method for maintaining independent games is to create a copy of the current board state for each processing element [13]. This method works well for small game trees, but becomes infeasible with large trees such as the one required for Go. The infeasabililty stems from the fact that the maximum width of the Go game tree is 361!, requiring the maximum amount of blocks and threads with striding to process the nodes in parallel. A thread's board copy can be reused as it strides, so the maximum amount of board copies is bounded by the total number of threads, which is 67,107,840 threads for the architecture of the device on which the experiments were performed [14]. Storing this many copies of the board exceeds the available device global memory.

In order to reduce the amount of memory required, there were no additional copies of the game board created. A single copy of the board was used, and each thread was allocated a key that was used to determine if a move had been processed by the thread. Each key is represented by a bit. The keys are stored in an array of integers, each integer holding 32 keys. A

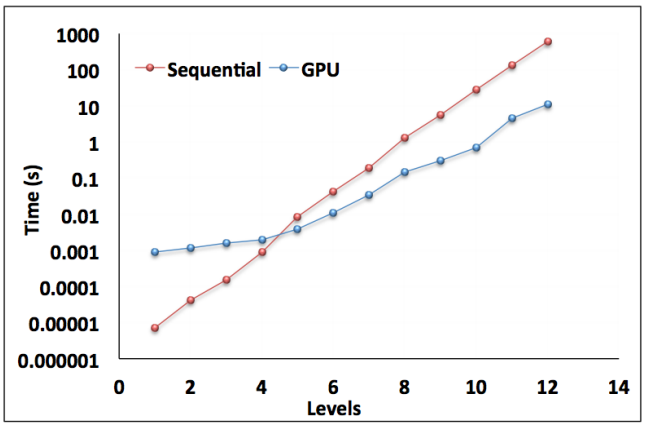

Fig. 2: Execution times for a 5x5 board.

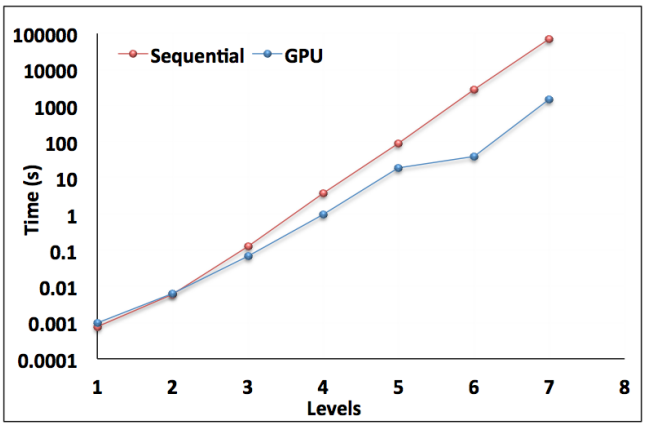

Fig. 3: Execution times for a 19x19 board.

second integer array of the same size was used to determine if an occupied intersection was occupied by a white or black stone. Therefore, the total memory used is that required to store the game board and two bits for each thread, which reduces the memory from the previous method by a factor of over 1,400 when using the maximum amount of threads.

V. Results and Analysis

Results were collected for the situation where the black player has placed the first stone, and the game tree is searched for the best move for the white player. PVS was implemented in C++ and PV-Split was implemented in CUDA C. Sequential and parallel timings were collected for a 5x5 board and the standard 19x19 board. The system used for the experiments has the following specifications:

- Intel i7 4790k running at 4.0 GHz
- 16 GB DDR3 memory
- NVIDIA GeForce GTX 970

Tables I and II show the execution times and speedups for the two board sizes, and this data are shown in graphs in Figures 2 - 4.

The graph in Figure 2 shows that it is only beneficial to move the search computation to the GPU once the number of levels exceeds four for the 5x5 board. Figure 3 shows

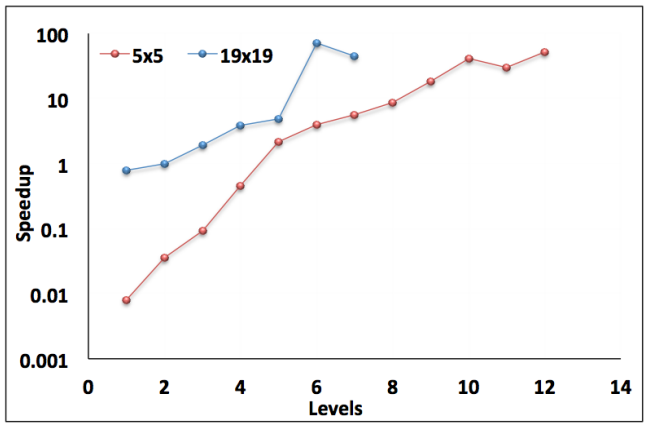

Fig. 4: Speedups for the 5x5 and 19x19 boards.

TABLE I: Execution Times and Speedup for a 5x5 Board.

Levels	Sequential Time (s)	GPU Time (s)	Speedup
1	7.0E-06	8.8E-04	0.008
2	4.2E-05	1.2E-03	0.036
3	1.5E-04	1.6E-03	0.094
4	9.1E-04	2.0E-03	0.456
5	8.5E-03	3.9E-03	2.172
6	0.042	0.011	3.941
7	0.186	0.033	5.559
8	1.268	0.148	8.558
9	5.571	0.305	18.264
10	28.392	0.689	41.227
11	133.440	4.465	29.887
12	591.876	11.261	52.561

TABLE II: Execution Times and Speedup for a 19x19 Board.

Levels	Sequential Time (s)	GPU Time (s)	Speedup
1	7.8E-04	9.9E-04	0.787
2	6.2E-03	6.3E-03	0.984
3	0.128	0.070	1.902
4	3.736	0.971	3.846
5	93.193	19.331	4.821
6	2815.280	39.150	71.910
7	69114.300	1538.940	44.910

it is beneficial to move the computation to the GPU once the amount of levels exceeds two for the 19x19 board. It is expected that the level threshold at which it is beneficial to move the computation to the GPU is lower for the 19x19 board since the breadth of the 19x19 game tree is larger.

For the GPU implementation, various combinations of block and thread amounts were executed to determine the optimal combination for both board sizes. The combinations were tested for deeper levels of the trees, but the same combination was used for all levels for the results for that board size. For the 5x5 board, 4096 blocks and 32 threads per block were used, and for the 19x19 board, 512 blocks and 32 threads per block were used. As the number of threads passed these values, the execution time would start increasing.

Increased execution time, when a result of an increase in the number of threads, can typically be attributed to either thread scheduling or memory throughput. To further understand the

reasoning for the increased execution time, the CUDA profiler [14] was used to analyze the differences in various memory usages as the number of threads increased. The analysis data generated by the profiler for a static number of blocks, and 1, 32, and 128 threads is shown in Table III.

TABLE III: CUDA Profiler register analysis for 512 blocks and 1, 32, and 128 threads.

Number of Threads: 1		
Variable	Theoretical	Device Limit
Registers/Thread	87	255
Registers/Block	2816	65536
Block Limit	20	32

Number of Threads: 32		
Variable	Theoretical	Device Limit
Registers/Thread	87	255
Registers/Block	2816	65536
Block Limit	20	32

Number of Threads: 128		
Variable	Theoretical	Device Limit
Registers/Thread	87	255
Registers/Block	11264	65536
Block Limit	5	32

As expected, the results for 1 and 32 threads are the same since a warp will always schedule 32 threads. However, as the number of threads increases from 32 to 128, the number of blocks that can be simultaneously executed on a Streaming Multiprocessor reduces from 20 to 5. In this situation, the number of blocks that can be executed is limited by the number of registers since the recursive kernel has high register usage. Each GPU model has a set limit on the number of registers that can be used by each block, so if this amount is exceeded then some of the blocks must wait to be scheduled. Increasing the number of threads increases the number of times the kernel is executed, thus increasing the register usage. As a result, the decreased warp occupancy theoretically reduces the number of elements processing simultaneously and also reduces the efficiency at which the GPU can perform latency hiding, both attributing to an increase in execution time [14].

VI. Conclusion and Future Work

In this paper, we have presented a CUDA implementation of the recursive PV-Split algorithm for searching the Go game tree. Running the game tree search in parallel allows for a reduction in computation time so more accurate *best moves* can be found by reaching deeper levels of the tree in a less amounts of time. The results for the standard board size show speedup for all levels greater than 2, with a high speedup of $\approx$72x at ply 6. In addition to reducing the computation time, a memory optimization method was introduced that addressed the issue of having to store independent copies of the game board for each processing element on the GPU.

In the future, we would like to make optimizations to the current GPU implementation, such as reducing the amount of register usage so a higher number of threads can be used efficiently, and taking advantage of shared memory to promote faster memory accesses. We would also like to run the game tree search on multiple GPUs. Currently, the amount of children nodes that can be processed in parallel is bounded above by amount of available device memory. Traversing the tree on multiple devices would increase the number of available resources and allow for higher degrees of parallelism. Since our single GPU implementation uses device function recursion, the algorithm would need to be restructured so the host is responsible for dividing the children nodes in a level among the available devices. It would also be necessary to update the Alpha-Beta window across the devices when it is updated on a particular level.

Acknowledgment

This material is based in part upon work supported by: The National Science Foundation under grant number(s) IIA-1329469, and by Cubix Corporation through use of their PCIe slot expansion hardware solutions and HostEngine. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or Cubix Corporation.

References

[1] E. C. D. van der Werf, *AI techniques for the game of Go*. UPM, Universitaire Pers Maastricht, 2005.

[2] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[3] "Comparision between chess and go." [Online]. Available: http://users.eniinternet.com/bradleym/Compare.html

[4] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[5] "The mystery of go, the ancient game that computers still can't win." [Online]. Available: http://www.wired.com/2014/05/the-world-of-computer-go/

[6] E. C. van der Werf, H. J. Van Den Herik, and J. W. Uiterwijk, "Solving go on small boards." *ICGA Journal*, vol. 26, no. 2, pp. 92–107, 2003.

[7] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1, pp. 57–83, 2002.

[8] L. Li, H. Liu, H. Wang, T. Liu, and W. Li, "A parallel algorithm for game tree search using gpgpu," 2014.

[9] A. A. Elnaggar, M. A. Aziem, M. Gadallah, and H. El-Deeb, "A comparative study of game tree searching methods," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 5, no. 5, 2014.

[10] T. A. Marsland and M. Campbell, "Parallel search of strongly ordered game trees," *ACM Computing Surveys (CSUR)*, vol. 14, no. 4, pp. 533–551, 1982.

[11] C.-W. Chou, P.-C. Chou, H. Doghmen, C.-S. Lee, T.-C. Su, F. Teytaud, O. Teytaud, H.-M. Wang, M.-H. Wang, L.-W. Wu *et al.*, "Towards a solution of 7x7 go with meta-mcts," in *Advances in Computer Games*. Springer, 2012, pp. 84–95.

[12] D. Strnad and N. Guid, "Parallel alpha-beta algorithm on the gpu," in *Information Technology Interfaces (ITI), Proceedings of the ITI 2011 33rd International Conference on*. IEEE, 2011, pp. 571–576.

[13] A. Elnaggar, M. Gadallah, M. A. Aziem, H. Aldeeb *et al.*, "Autonomous checkers robot using enhanced massive parallel game tree search," in *Informatics and Systems (INFOS), 2014 9th International Conference on*. IEEE, 2014, pp. PDC–35.

[14] "Cuda toolkit documentation." [Online]. Available: https://docs.nvidia.com/cuda/index.html