Parallel UCT Search on GPUs

Nicolas A. Barriga
Department of Computing Science
University of Alberta
barriga@ualberta.ca

Marius Stanescu
Department of Computing Science
University of Alberta
astanesc@ualberta.ca

Michael Buro
Department of Computing Science
University of Alberta
mburo@ualberta.ca

***Abstract*—We propose two parallel UCT search (Upper Confidence bounds applied to Trees) algorithms that take advantage of modern GPU hardware. Experiments using the game of Ataxx are conducted, and the algorithm's speed and playing strength is compared to sequential UCT running on the CPU and *Block Parallel* UCT that runs its simulations on a GPU. Empirical results show that our proposed *Multiblock Parallel* algorithm outperforms other approaches and can take advantage of the GPU hardware without the added complexity of searching multiple trees.**

I. Introduction

Monte-Carlo Tree Search (MCTS) has been very successful in two player games for which it is difficult to create a good heuristic evaluation function. It has allowed Go programs to reach master level for the first time (see [1] and [2]). Recent results include a 4-stone handicap win by Crazy Stone against a professional player [3].

MCTS consists of several phases, as shown in Figure 1, which for our purposes can be classified as in-tree and off-tree. During the in-tree phases, the algorithm needs to select a node, expand it, and later update it and its ancestors. The off-tree phase consists of possibly randomized playouts starting from the selected node, playing the game until a terminal node is reached, and returning the game value which is then used for updating node information. For this paper we choose UCT (Upper Confidence bounds applied to Trees, [4]) as a policy for node selection and updating because it has been proven quite successful in computer game playing [5].

As with most search algorithms, the more time MCTS spends on selecting a move, the greater the playing strength. Naturally, after searching the whole game tree, there are no further gains. However, games such as Go which are characterized by a very large branching factor have large game trees that cannot be fully searched in a feasible amount of time. Parallelizing MCTS has led to stronger game play in computer programs [6], and state of the art UCT implementations use distributed algorithms which scale well on thousands of cores [7]. Unfortunately, the prohibitive cost of highly parallel machines has limited the full exploration of the potential of these algorithms.

However, a new type of massively parallel processors capable of running thousands of threads in Single Instruction Multiple Thread (SIMT) fashion, with performance in the Teraflops range, has become mainstream. These processors, called Graphics Processing Units (GPUs), are widely available on most of the current computer systems, ranging from smartphones to supercomputers. So far, there have been only a few attempts of harnessing this computing power for heuristic search.

In the remainder of the paper we first describe the traditional MCTS parallelization techniques on multi-core CPUs and computer clusters, which is followed by an introduction of GPU architectures and an existing *Block Parallel* MCTS algorithm designed to take advantage of this hardware. We then propose two new algorithms to improve on *Block Parallel* and describe implementation details. After discussing experimental results, we finish the paper with a concluding section that also suggests future work in this area.

II. Background and Related Work

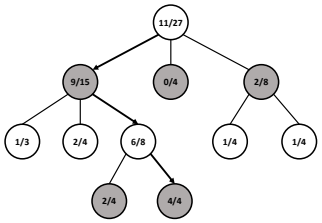
In addition to its success when applied to board games, MCTS is easier to parallelize than $\alpha\beta$ [8], [9], [10], [11], [6]. There are several existing methods for parallelizing MCTS, which we describe in the following subsections. We start by describing CPU only parallel algorithms and then present advantages and disadvantages of GPU parallelization followed by describing an MCTS algorithm which makes use of both CPU and GPU parallelization which we we will improve upon.

A. CPU Only Parallel MCTS

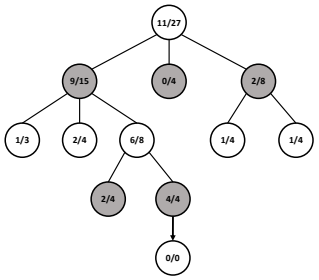
The most common methods for parallelizing MCTS have been classified by [6] as root parallelization (Figure 2), leaf parallelization and tree parallelization (Figure 3). Leaf parallelization is the easiest to implement, but it usually has the worst performance. It works with a single tree and each time a node is expanded, several parallel playouts are performed, with the goal of getting a more accurate value for the node. Root parallelization constructs several independent search trees, and combines them at the end. It has the least communication overhead, which makes it well suited for message passing parallel machines, for which it has shown good results. Tree parallelization works by sharing a single tree among different threads, with access coordinated by either a single global mutex, or several local mutexes. It works best on shared memory systems and its performance is highly dependent on the ratio of time spent in-tree to time spent on playouts.

B. GPU Hardware for Parallelization

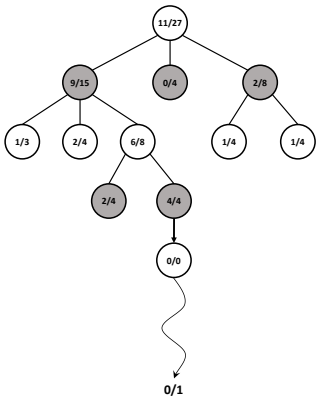
In 2006 NVIDIA launched its GeForce 8800 videocard with the G80 GPU, which at the time was a major leap forward both in terms of graphics performance and in architecture. It was the first card to allow full *General-Purpose computing on Graphics Processing Units* (GPGPU) through its *Compute Unified Device Architecture* (CUDA) programming platform.



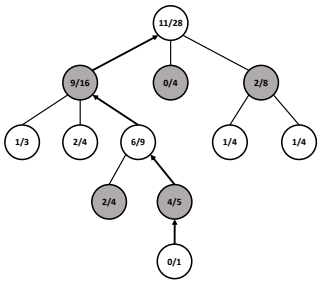
(a) The selection function is applied recursively until a node with unexpanded children is found



(b) One (or more) leaf nodes are created



(c) One (or more) simulated game(s) is played



(d) The game result is propagated up the tree

Fig. 1. How Monte Carlo tree search works.

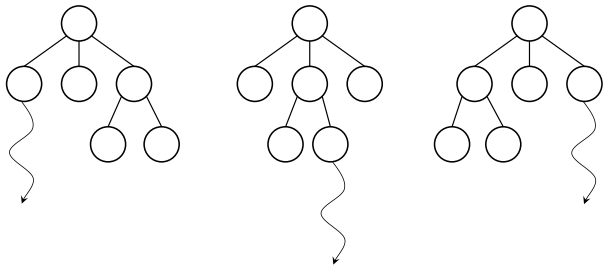


Fig. 2. Root Parallel.

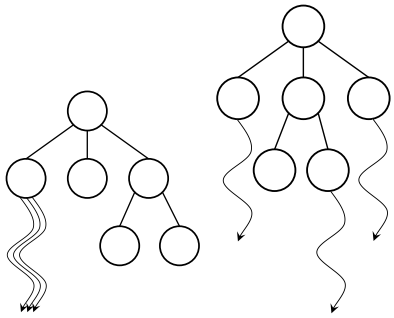


Fig. 3. Leaf Parallel vs. Tree Parallel.

This new architecture has since then evolved into processors capable of executing thousands of threads in parallel, using simple cores called *Streaming Processors* (SPs) which are grouped into several *Multiprocessors* (MPs). These cores are optimized for throughput, running as many threads as possible, but not focusing on the speed and latency of any individual thread. On the other hand, a CPU runs a small number of parallel threads, but focuses on running them as fast as possible while assuring each thread a fair share of processing time. A modern CPU is an *out-of-order superscalar* processor with *branch prediction* [12]:

- it reorders instructions to make use of instruction cycles that would otherwise be wasted,
- it provides instruction-level parallelism, executing more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor,
- and it improves the flow in the instruction pipeline by trying to predict the outcome of the next conditional jump instruction.

A GPU has none of those features, making each thread run much more slowly.

The keys to the vast GPU processing power are a couple of design decisions: *Single Instruction Multiple Thread* (SIMT) architecture, in which a group of threads, called a Warp (presently comprised of 32 threads), all execute the same instruction. If there is control divergence within the threads in a Warp, some threads will be deactivated while others execute the divergent code, and then the roles will be reversed — which effectively decreases the instruction throughput. The second feature that contributes to performance is *zero-overhead scheduling*, which, by means of the GPU having thousands of registers, can maintain tens of thousands of 'live' threads with almost no context switch cost. This allows the GPU to hide global memory latency by switching to another thread group when one thread group requests a global memory access. The number of threads that can execute concurrently is given by the number of SPs or cores a GPU has (on current hardware between the high hundreds to low thousands). The number of live threads that can be scheduled depends on the amount of shared memory (a manually controllable L1 cache) and registers all those threads need, and can go up to tens of thousands. The particular number of standby threads in an application divided by the theoretical maximum supported by a specific GPU is called the *occupancy factor*.

Finally, it is worth mentioning that the latest GPUs support multiple kernels (the CUDA name for a function executing on the GPU) running in parallel, which cannot share MPs, however. Moreover, these GPUs can have one or two copy engines, which can manage data transfers simultaneously to and from main memory to GPU memory.

These details make it difficult to estimate the performance that can be achieved by a particular GPU on a particular application.

1) CUDA Programming Terminology: In the CUDA programming model, a function to be executed on the GPU is called a *kernel*. A *kernel* is executed by several threads grouped in thread *blocks*. All blocks in a kernel invocation compose a *grid*.

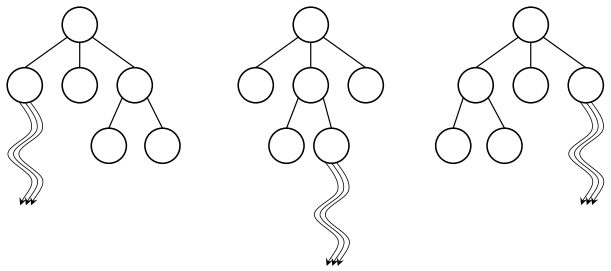


Fig. 4. Block Parallel.

Each thread can synchronize and communicate via shared memory with other threads in the same block, but there can be no synchronization between blocks. Blocks need to be completely independent of each other, so that they can run and be scheduled in different MPs without any communication overhead.

We can estimate how efficiently we are utilizing the GPU by calculating the occupancy. An occupancy of 100% means that based on the hardware resources available — mainly registers, cores and shared memory — and on the resources needed by each thread, the entire theoretical computing power of the GPU is used. The resources needed depend on the *kernel* to be run; the number of registers and the memory needed is provided by the CUDA compiler. Reaching 100% occupancy using some number of blocks B each with T threads doesn't imply that all $B \cdot T$ threads will be running at the same time. It means that they are all standing by and ready to be scheduled by the thread scheduler. Thus, a configuration with 50% occupancy will not necessarily run twice as long as one with full occupancy to perform the same amount of work.

C. CPU/GPU Parallel MCTS

In [13], the authors implemented a hybrid Root/Leaf Parallel MCTS with the playouts running on a GPU. Their *Block Parallel* algorithm expands several *Root Parallel* trees on the CPU, and then runs blocks of *Leaf Parallel* playouts on the GPU, as shown in Figure 4. The root parallel part of the algorithm is shown in Figure 5. It searches several game trees in parallel in line 2 and then combines the values of all children of the root node and selects the best move.

Figure 6 describes leaf parallel search of one tree. After selecting a node and expanding one child, a CUDA kernel is called in line 7: PLAYOUTKERNEL⋘1,threads⋙(CHILD,T). The parameters between triple angle brackets specify that this kernel will run one block, with *threads* parallel playouts on the GPU. Their results will be copied back to main memory in the following line. Then, the tree is updated, from the last expanded node up to the root.

In a CUDA kernel, the following variables are automatically defined by the system:

- *threadId*: the index of the current thread inside a thread block.
- *blockDim*: the number of threads in a block.
- *blockId*: the index of the current block in the grid.

The CUDA kernel shown in Figure 7 takes as parameters an array of board positions and whose turn it is to move. Each thread will run a playout for the board position indexed by its *blockId*, effectively running *blockDim* playouts for each board. Algorithm BLOCK only uses one block to run parallel playouts for one position. Lines 3 to 12 compute the sum over the array of *blockDim* playout outcomes in parallel.

III. PROPOSED ENHANCEMENTS

In this paper, we discuss the implementation of two algorithms:

1) *GPU Parallel* MCTS, which is a *Block Parallel* MCTS with the trees residing on the GPU.
2) *Multiblock Parallel* MCTS, similar to the *Block Parallel* algorithm proposed by [13], but running simulations for all the children instead of only for one child of the selected node — with the goal of fully utilizing the GPU hardware.

A. GPU Parallel

GPU Parallel is a *Block Parallel* MCTS completely executed by the GPU. After receiving the state to be searched, the GPU will construct enough independent search trees, starting from that position, to fully occupy the GPU (a few hundred are enough in our experiments). A multiple of 32 threads will be used for leaf parallelism in each of the trees. For occupancy purposes, several trees are handled by each thread block. The value of 32 threads is chosen because that is the minimum thread scheduling unit in the CUDA architecture, a *Warp*. The number of trees and threads is dependent on the specific hardware available and chosen to fully utilize the GPU.

Designing time controls for GPU processing is non-trivial because the programmer has no control of the thread scheduling which is optimized for throughput rather than latency. It

```
 1: function PARALLELSEARCH(Board current, Turn t)
Require: trees                          ▷ Number of root parallel trees
Require: time                           ▷ Time to search
 2:     parallel for i ← 0, trees do
 3:         roots[i]←BLOCK(current, t, time)
 4:     end for
 5:     int num←NUMCHILDREN(root[0])
 6:     for i←0, trees do
 7:         for j←0, num do
 8:             total[j]+=roots[i].children[j].mean
 9:         end for
10:     end for
11:     value← −∞
12:     for i←0, children do
13:         if total[i]>value then
14:             value←total[i]
15:             bestChild←roots[0].children[i].board
16:         end if
17:     end for
18:     bestMove←GETMOVE(current, bestChild)
19:     return bestMove
20: end function
```

Fig. 5. Main Parallel MCTS, computes the best move for a given state.

```
 1: function BLOCK(Board current, Turn t, Time time)
Require: int threads                          ▷ threads per tree
 2:     Node root(current)          ▷ root tree at current board
 3:     int startTime←CURRENTTIME( )
 4:     while CURRENTTIME( )-startTime<time do
 5:         Node node←SELECT(root)          ▷ UCT selection
 6:         Node child←EXPAND(node)         ▷ UCT expansion
 7:         PLAYOUTKERNEL⋘1,threads⋙(CHILD,T)
 8:         int[] wins←transfer results from GPU
 9:         UPDATETREE(child, wins)
10:     end while
11:     return root
12: end function
```

Fig. 6. Block Parallel MCTS, computes a game tree for a given state.

is impossible to tell each thread to run for a certain amount of time because some of them may be suspended indefinitely and never get a chance to run in the allotted time. To solve this problem our program estimates in advance the number of nodes each thread should search. This estimate depends on the number of nodes per second it was able to compute during the previous move, which allows the system to adapt as the game makes a transition into easier or more difficult positions.

B. Multiblock Parallel

The *Multiblock Parallel* algorithm, shown in Figure 8, is quite similar to the *Block Parallel* approach proposed by [13]. However, to increase GPU occupancy, instead of performing several leaf simulations for a selected node, all the selected node's children are expanded and several simulations are performed for each of them. The algorithm in Figure 9 is quite similar to the *Block Parallel* shown in Figure 6, but in line 6 all children are expanded for the selected node, *num* stores the number of children expanded, and then the PLAYOUTKERNEL is called in line 8 with the array of boards corresponding to those children. The kernel is configured to run *num* blocks of *threads* threads. The number of *threads* is a parameter of the

```
 1: function PLAYOUTKERNEL(Board[] boards, Turn t)
Require: threadId                        ▷ index of current thread
Require: blockId                         ▷ block of current thread
Require: blockDim                        ▷ threads in a block
Require: __shared__ values[blockDim]     ▷ shared array for
                                         ▷ intermediate results
 2:     values[threadId]←RANDPLAYOUT(boards[blockId])
 3:     offset←blockDim/2
 4:     while offset>0 do
 5:         if threadId < offset then          ▷ parallel sum
 6:             values[threadId] += values[threadId + offset]
 7:         end if
 8:         offset←offset/2
 9:     end while
10:     if threadId==0 then
11:         wins[blockId]←values[0]
12:     end if
Output: int[] wins      ▷ array with the wins for each board
13: end function
```

Fig. 7. CUDA kernel for playouts.

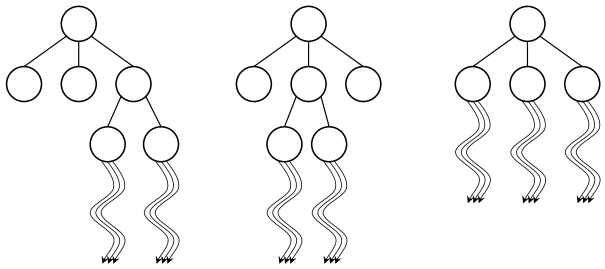


Fig. 8. Multiblock Parallelism.

```
 1: function MULTIBLOCK(Board current, Turn t, Time time)
Require: int threads                          ▷ threads per tree
 2:     Node root(current)          ▷ root tree at current board
 3:     int startTime←CURRENTTIME( )
 4:     while CURRENTTIME( )-startTime<time do
 5:         Node node←SELECT(root)          ▷ UCT selection
 6:         Node[] children←EXPANDALLCHILDREN(node)
 7:         int num←SIZE(children)
 8:         PLAYOUTKERNEL⋘num,threads⋙(children,t)
 9:         int[] wins←transfer results from GPU
10:         for i←0, num do
11:             UPDATETREE(children[i], wins[i])
12:         end for
13:     end while
14:     return root
15: end function
```

Fig. 9. MultiBlock Parallel MCTS, computes a game tree for a given state.

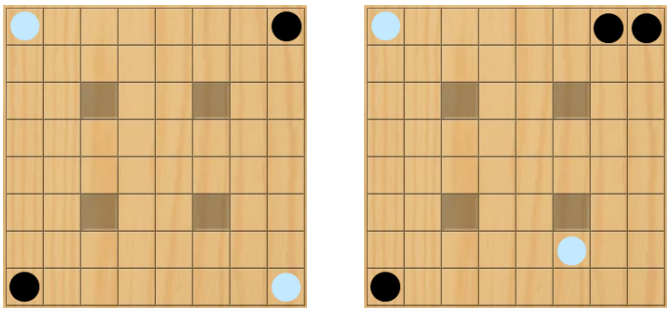
algorithm.

IV. EXPERIMENTAL RESULTS

For all experiments we used a PC with Intel Core2 Quad CPU Q8300 processor at 2.5GHz, with 8GB of RAM, running Ubuntu 12.04.2 LTS. The video card is a GeForce GTX 650 Ti, which contains a GK106 GPU with compute capability 3.0, and 2GB of RAM. This card has 4 multiprocessors running at 928Mhz, each of which can execute 192 single precision floating point operations per clock cycle. The card is therefore referred to as having 768 (4x192) CUDA Cores. Integer performance is much slower than floating point at 160 32-bit integer add/compare/logical operations per clock cycle, and 32 32-bit integer shift/multiply operations per clock cycle. As 64-bit integer operations are usually a combination of two or three 32-bit operations, the throughput is estimated to be around 200 arithmetic and 40 bitwise 64-bit integer operations per clock cycle for the combined 4 multiprocessors. The bulk of the operations in our simulations are 64-bit integer bitwise operations. This card has only one copy engine, meaning data cannot be transferred simultaneously in both directions.

The algorithms are compared by playing 8 × 8 Ataxx. Ataxx is a 2-player, perfect information, zero-sum game. The game usually starts with two pieces in the corners of the board for each player, as shown in Figure 10(a). There are two type of moves in the game:

- a *clone* move adds a new piece to an empty square adjacent to any piece of the same color



(a) Initial Ataxx board

(b) Ataxx board after one move from each player

Fig. 10. Game of Ataxx played on a 8x8 board with 4 blocked squares.

TABLE I. COMPLEXITIES OF SOME GAMES.

Game	Game-tree Complexity	State-space Complexity	Average Branching Factor	Average Game Length
Othello	10^{58}	10^{28}	10	58
Chess	10^{123}	10^{50}	35	80
Ataxx	10^{163}	10^{28}	65	90
Go	10^{360}	10^{172}	250	150

- a *jump* move takes one piece and transfers it to an empty square two spaces away from its original position.

At each turn, each player must make one move as long as there are legal moves available, or pass otherwise. Figure 10(b) shows the board position after White made a jump move and Black responded with a clone move. It is common to have blocked squares on the board, where no pieces can be played. After each move, all opponent pieces adjacent to the moved piece are 'captured' and switch color. The game ends when there are no more empty squares. The player with most pieces wins the game.

We computed the average branching factor and Ataxx game length for the board in Figure 10 by running a few hundred games. We obtained an average branching factor of 65 which is almost double to that of chess, and a game length of 90 moves. These values, along with the estimated state-space complexity and game-tree complexity for Ataxx, Othello, Chess, and Go are shown in Table I. For Ataxx, we estimated the state-space complexity as approximately $3^{60} \simeq 10^{28}$ (each of the 60 non-blocked squares can be empty, white or black) and the game-tree complexity around $65^{90} \simeq 10^{163}$ (average branching factor to the power of average game length). For the other games we use the values from [14].

Ataxx was chosen because it has a bigger game tree complexity than Othello — the game used to evaluate the *Block Parallel* algorithm in [13] — but is still simple enough to be easily implemented in CUDA.

All experiments were performed with 200 games per data point and 100ms per move, counting a win as 1 point, a tie as 0.5 points and a loss as 0 points. The games were played on 10 different starting configurations, with 0, 4 or 8 blocked squares. The UCT exploration constant was tuned using the same parameters.

TABLE II. SIMULATIONS PER SECOND, MID-GAME.

Sequential	**Trees×Threads**	GPU
88×10^3	**128×32**	1817×10^3
	256×32	2015×10^3
	512×32	1458×10^3

TABLE III. SIMULATIONS PER SECOND SPEEDUP OVER SEQUENTIAL.

Game stage	GPU 256x32
Start	17.5
Mid	22.9
End	17.2

TABLE IV. NODES PER SECOND, MID-GAME.

Sequential	**Trees×Threads**	GPU
88049	**128×32**	56784
	256×32	62971
	512×32	45578

We will assess the performance of the parallel algorithms running on the GPU by playing them against a sequential single threaded UCT running on the CPU.

All of the evaluated algorithms use the same playout function, a simple random playout until the end of the game is reached. This function is not optimized for any specific target architecture.

A. GPU Results

In this first experiment the *GPU Parallel* algorithm uses 128, 256, or 512 trees, each with 32 leaf parallel threads. Four trees are handled by each thread block. The sequential algorithm always uses one thread and one tree.

Table II shows the average number of simulations (playouts) per second, performed by each algorithm in middle-game positions (at move 30). Table III shows the speedup of *GPU Parallel* at its fastest settings (256 trees and 32 threads per tree) over sequential at a start, mid-game (move 30) and late-game (move 60) positions. The speed for 256 trees and 32 threads per tree is the highest because it is with this configuration that we get 100% occupancy of our GPU. Using 128 trees only gets us 50% occupancy, wasting some GPU resources, while using 512 trees gets us 200% occupancy having more threads than the GPU can schedule.

Finally, Table IV shows the speed of the algorithms in terms of nodes expanded per second. Note that although the number of nodes for the GPU algorithm (around $50,000$) is comparable to that of the sequential algorithm (close to $88,000$), those nodes are spread over a few hundred trees. Therefore, each tree is much smaller than the single tree expanded on the CPU. This accounts for the poor game strength results shown in Figure 11 which indicates the *GPU Parallel* algorithm performance for four occupancy percentages. We get 50% occupancy when using 128 trees and 32 threads per tree. The 75% mark corresponds to 192 trees. The 100% mark shows parameters for full occupancy, with 256 trees, while the 200% mark has 512 trees and twice the amount of threads that can be supported by the hardware. For this last setting, on average, half the threads are waiting idle and are not even available for scheduling. The 100% mark shows a peak performance of 42.5% winning rate. We can see a significant

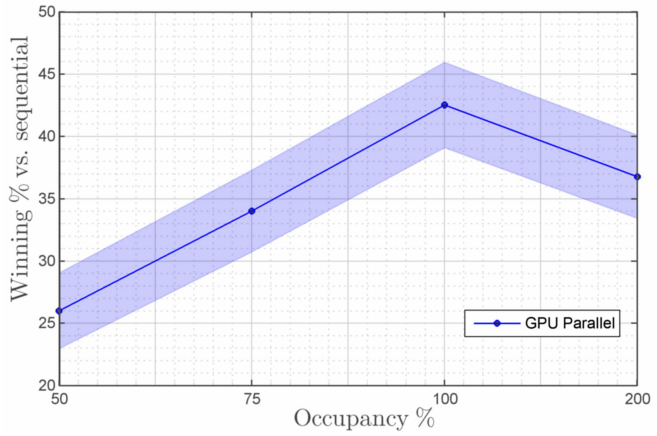


Fig. 11. GPU Parallel winning percentage against sequential. Shaded areas represent 95% confidence intervals.

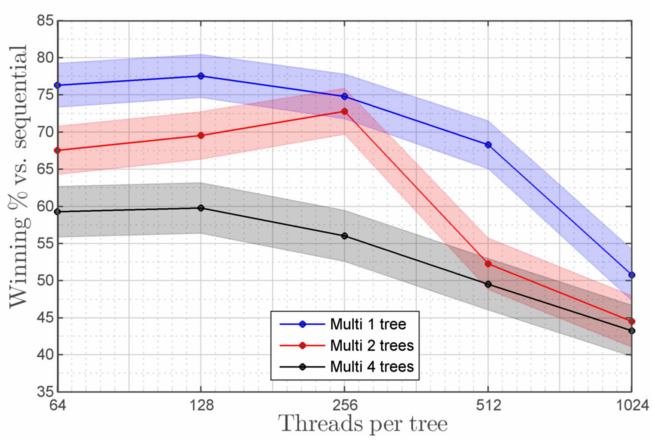


Fig. 12. *Multiblock Parallel* winning percentage against sequential. Shaded areas represent 95% confidence intervals.

performance increase between 50% and 100% occupancy, and a slight drop when going to 200%. This seems encouraging as the expected performance on a GPU with more MPs should be higher.

B. Multiblock Results

We implemented the *Block Parallel* and *Multiblock Parallel* algorithms with 1, 2, and 4 trees, each using 64 to 1024 GPU threads. Note that when only one tree is used, the *Block Parallel* algorithm degenerates into leaf parallelization.

Figure 12 shows the win percentages of different *Multiblock* configurations against the sequential MCTS. The best result is obtained by the implementation using only one three. Figure 13 compares *Block* and *Multiblock* algorithms using this setting. For the *Block Parallel* algorithm the number of trees didn't make a significant difference. Therefore, we used the one tree implementation for comparison. This contradicts findings by [13] which concludes that the number of trees has a bigger impact than the number threads per tree. This is likely due to the different hardware used: the Nvidia Tesla C2050 has 14 MPs (multiprocessors), each of which can execute 32 single precision floating point operations per clock cycle, for a total of 448 CUDA Cores, while the GeForce GTX 650 Ti we use has 4 multiprocessors, each of which can execute 192 single precision floating point operations per clock cycle, for a total of 768 CUDA Cores. An increased number of MPs means better parallelism for concurrent kernels, as kernels cannot share an MP. Also, the Tesla card has two copy engines, which allows it to copy data to and from the GPU memory at the same time, while the GeForce only has one copy engine, so simultaneous data transfers will be queued.

The best overall performer is the *Multiblock Parallel*, using one tree and 128 threads per node, winning 77.5% of its games. The maximum performance for the *Block Parallel* algorithm is 49.0% win rate, achieved for one tree and 512 threads.

Tables V, VI and VII, respectively, show the average number of simulations (playouts) per second, the speedup over the sequential algorithm, and the number of nodes expanded per second.

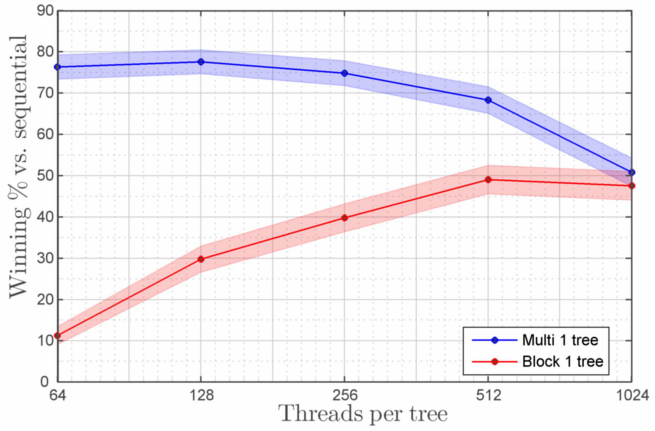


Fig. 13. Strength comparison between *Multiblock Parallel* and *Block Parallel*. Shaded areas represent 95% confidence intervals.

TABLE V. SIMULATIONS PER SECOND, MID-GAME.

Sequential	**Trees×Threads**	Block	Multi
88×10^3	**1×128**	106×10^3	1877×10^3
	1×256	210×10^3	2235×10^3
	1×512	376×10^3	2438×10^3

TABLE VI. SIMULATIONS PER SECOND SPEEDUP OVER SEQUENTIAL.

Game stage	Block 1x512	Multi 1x128
Start	3.7	17.4
Mid	4.3	21.3
End	4.9	13.7

TABLE VII. NODES PER SECOND, MID-GAME.

Sequential	**Trees×Threads**	Block	Multi
88049	**1×128**	828	14667
	1×256	820	8729
	1×512	734	4761

Block Parallel's speed in terms of nodes per second doesn't significantly change when increasing the number of threads on each tree. This is likely due to one tree not being enough to fully utilize the GPU. This means that the generated trees are of similar size, but the kernel is running more threads to perform simulations. Hence, the quality of the trees is likely improving which explains the rise in performance seen in Figure 13. On the other hand, *Multiblock Parallel* performs very differently. When using one tree both its speed (in terms of nodes per second) and its performance diminish when increasing the number of threads over 128. At this point, the trade-off between obtaining a better quality tree by adding more threads is offset by the time slowdown caused by expanding each node. Doubling the number of threads roughly halves the number of explored nodes per second in this case (table VII). This is due to *Multiblock Parallel* already being close to fully utilizing the GPU, as exhibited by the scant increase in simulations per second in table V.

V. Conclusions and Future Work

Our experiments indicate that in 8×8 Ataxx, the *Multiblock Parallel* algorithm was able to successfully turn its simulation speed advantage into a playing strength advantage, while the *GPU Parallel* algorithm was not. A possible explanation is that because *GPU Parallel* is expanding many trees, each tree is not searched very deeply. Using fewer trees, but exploring each one in a *Tree Parallel* fashion may be worth trying. However, this approach may be difficult to implement because of the restrictions imposed by CUDA, which doesn't support mutexes, for example. However, the lock-less approach described in [15] might be applicable here.

An unexpected result is that even when seeing a speedup comparable to what Rocki et al. [13] found for the *Block Parallel* algorithm, we did not obtain a similar playing strength improvement. One reason for this could be the different application domain: Ataxx has a bigger branching factor than Othello, which leads to a shallower search (due to having to expand more siblings of each node, before expanding a child). However, for *Multiblock Parallel*, more siblings means a better GPU utilization, by scheduling more threads to execute the kernel.

Another reason is the different hardware used: their GPU had 14 multiprocessors with 32 CUDA cores each and 2 copy engines, while ours had 4 multiprocessors with 192 CUDA cores each and 1 copy engine. To investigate the true cause more experiments need to be conducted on different domains and a wider hardware range. An obvious follow-up would be to test our algorithms on Othello, which would allow for a more direct comparison with [13]. A different alternative would be to use artificial game trees with configurable parameters, which could give us broader insights on the interaction between game characteristics — like branching factor, game length and playout speed — and hardware characteristics like number of MPs, CUDA cores and copy engines.

References

[1] R. Coulom, "Efficient selectivity and backup operators in Monte Carlo tree search," *Computers and Games*, pp. 72–83, 2007.

[2] S. Gelly, "A contribution to reinforcement learning; application to computer Go," Ph.D. dissertation, Universite Paris-Sud, 2008.

[3] "Human-computer go challenges," April 2013. [Online]. Available: http://www.computer-go.info/h-c/index.html#2013

[4] L. Kocsis and C. Szepesvári, "Bandit based Monte Carlo planning," in *Machine Learning: ECML 2006*. Springer, 2006, pp. 282–293.

[5] F. Teytaud and O. Teytaud, "Creating an upper-confidence-tree program for Havannah," in *Proceedings of the 12th international conference on Advances in Computer Games*. Springer-Verlag, 2009, pp. 65–74.

[6] G. Chaslot, M. Winands, and H. van Den Herik, "Parallel Monte Carlo tree search," *Computers and Games*, pp. 60–71, 2008.

[7] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa, "Scalable distributed Monte Carlo tree search," in *Fourth Annual Symposium on Combinatorial Search*, 2011.

[8] T. Cazenave and N. Jouandeau, "A parallel Monte Carlo tree search algorithm," *Computers and Games*, pp. 72–80, 2008.

[9] S. Gelly, J.-B. Hoock, A. Rimmel, O. Teytaud, Y. Kalemkarian *et al.*, "On the parallelization of Monte Carlo planning," in *ICINCO*, 2008.

[10] T. Cazenave and N. Jouandeau, "On the parallelization of UCT," in *Proceedings of the Computer Games Workshop*, 2007, pp. 93–101.

[11] H. Kato and I. Takeuchi, "Parallel Monte Carlo tree search with simulation servers," in *13th Game Programming Workshop (GPW-08)*, 2008.

[12] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 451–460.

[13] K. Rocki and R. Suda, "Parallel Monte Carlo tree search on GPU," in *Eleventh Scandinavian Conference on Artificial Intelligence: Scai 2011*, vol. 227. IOS Press, Incorporated, 2011, p. 80.

[14] L. V. Allis, "Searching for solutions in games and artificial intelligence," Ph.D. dissertation, University of Limburg, Maastricht, 1994.

[15] M. Enzenberger and M. Müller, "A lock-free multithreaded Monte Carlo tree search algorithm," *Advances in Computer Games*, pp. 14–20, 2010.