Parallel Monte Carlo Tree Search on GPU

Kamil Rocki and Reiji Suda
The University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, 113-8654, Tokyo, Japan

Abstract. Monte Carlo Tree Search (MCTS) is a method for making optimal decisions in artificial intelligence (AI) problems, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search. It can theoretically be applied to any domain that can be described in terms of state, action pairs and simulation used to forecast outcomes such as decision support, control, delayed reward problems or complex optimization. The motivation behind this work is caused by the emerging GPU-based systems and their high computational potential combined with relatively low power usage compared to CPUs. As a problem to be solved we chose to develop an AI GPU(Graphics Processing Unit)-based agent in the game of Reversi (Othello) which provides a sufficiently complex problem for tree searching with non-uniform structure and an average branching factor of over 8. We present an efficient parallel GPU MCTS implementation based on the introduced 'block-parallelism' scheme which combines GPU SIMD thread groups and performs independent searches without any need of intra-GPU or inter-GPU communication. We compare it with a simple leaf parallel scheme which implies certain performance limitations. The obtained results show that using my GPU MCTS implementation on the TSUBAME 2.0 system one GPU can be compared to 100-200 CPU threads depending on factors such as the search time and other MCTS parameters in terms of obtained results. We propose and analyze simultaneous CPU/GPU execution which improves the overall result.

Keywords. MCTS; Monte Carlo, GPU, Parallel computing, AI, Artificial Intelligence, Reversi, GPGPU

Introduction

Monte Carlo Tree Search (MCTS)[1][2] is a method for making optimal decisions in artificial intelligence (AI) problems, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search.

Research interest in MCTS has risen sharply due to its spectacular success with computer Go and potential application to a number of other difficult problems. Its application extends beyond games, and MCTS can theoretically be applied to any domain that can be described in terms of state, action pairs and simulation used to forecast outcomes such as decision support, control, delayed reward problems or complex optimization[7][8][9][10]. The main advantages of the MCTS algorithm are that it does not require any strategic or tactical knowledge about the given domain to make reasonable decisions and algorithm can be halted at any time to return the current best estimate. Another advantage of this approach is that the longer the algorithm runs the better the solution and the time limit can be specified allowing to control the quality of the decisions made. This means that this algorithm guarantees getting a solution (although not

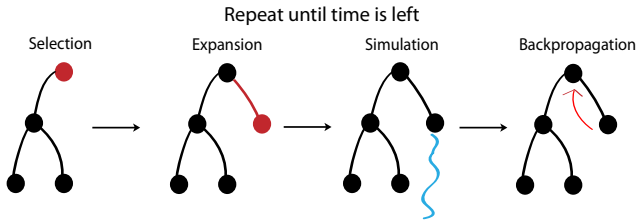

Figure 1. A single MCTS algorithm iteration's steps

necessarily the best one). It provides relatively good results in games like Go or Chess where standard algorithms fail. So far, current research has shown that the algorithm can be parallelized on multiple CPUs.

The motivation behind this work is caused by the emerging GPU-based systems and their high computational potential combined with relatively low power usage compared to CPUs. As a problem to be solved we chose developing an AI GPU(Graphics Processing Unit)-based agent in the game of Reversi (Othello) which provides a sufficiently complex problem for tree searching with a non-uniform structure and an average branching factor of over 8. The importance of this research is that if the MCTS algorithm can be efficiently parallelized on GPU(s) it can also be applied to other similar problems on modern multi-CPU/GPU systems such as the TSUBAME 2.0 supercomputer. Tree searching algorithms are hard to parallelize, especially when GPU is considered. Finding an algorithm which is suitable for GPUs is crucial if tree search has to be performed on recent supercomputers. Conventional ones do not provide good performance, because of the limitations of the GPU's architecture and the programming scheme, threads' communication boundaries. One of the problems is the SIMD execution scheme within GPU for a group of threads. It means that a standard CPU parallel implementation such as root-parallelism[3] fail. So far we were able to successfully parallelize the algorithm and run it on thousands of CPU threads[4] using root-parallelism. In this paper we present an efficient parallel GPU MCTS implementation based on the introduced *block-parallelism* scheme which combines GPU SIMD thread groups and performs independent searches without any need of intra-GPU or inter-GPU communication. We compare it with a simple leaf parallel scheme which implies certain performance limitations. The obtained results show that using my GPU MCTS implementation on the TSUBAME 2.0 system one GPU's performance can be compared to 50-100 CPU threads[4] depending on factors such as the search time and other MCTS parameters using root-parallelism. The block-parallel algorithm provides better results than the simple leaf-parallel scheme which fail to scale well beyond 1000 threads on a single GPU. The block-parallel algorithm is approximately 4 times more efficient in terms of the number of CPU threads needed to obtain results comparable with the GPU implementation.

1. Monte Carlo Tree Search

A simulation is defined as a series of random moves which are performed until the end of a game is reached (until neither of the players can move). The result of this simulation can be successful, when there was a win in the end or unsuccessful otherwise. So, let every

node i in the tree store the number of simulations T (visits) and the number of successful simulations S_i. First the algorithm starts only with the root node. The general MCTS algorithm comprises 4 steps (Figure 1) which are repeated until a particular condition is met (i.e. no possible move or time limit is reached)

1.1. MCTS iteration steps

1.1.1. Selection

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

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

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

Supposed that some simulations have been performed for a node, first the average node value is taken and then the second term which includes the total number of simulations for that node and its parent. The first one provides the best possible node in the analyzed tree (exploitation), while the second one is responsible for the tree exploration. That means that a node which has been rarely visited is more likely to be chosen, because the value of the second terms is greater. The C parameter adjusts the exploitation/exploration ratio. In the described implementation the parameter C is fixed at the value 0.2.

1.1.2. Expansion

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

1.1.3. Simulation

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

1.1.4. Backpropagation

- update the parents' values up to the root nodes. The numbers are added, so that the root node has the total number of simulations and successes for all of the nodes and each node

contains the sum of values of all of its successors. For the root/block parallel methods, the root node has to be updated by summing up results from all other trees processed in parallel.

2. GPU Implementation

This section describes the methods used in the GPU implementations. Since the majority of the operations are data movement, worth mentioning is the fact that the performance has been increased significantly by modifying the data structure compared to standard CPU implementations. Here, a single node is represented by a bitfield structure[6] to maximize the performance. It is crucial both during the simulation part, where less data has to be transferred within the GPUs and while transferring nodes between GPU and CPU. In case the block parallelization is used, the number of root nodes is equal to the number of blocks. The result is an array of shorts. We started using bitwise operations and were able to decrease the size to 128 bits. This is not so crucial for the CPU implementation, but in the GPU case the speedup was almost 10-fold. Minimizing the size was crucial to get good performance. For each of the 64 fields on the board, there is one bit indicated if occupied and one bit indicating color. This is a total of 128-bits of board state.

In the GPU implementation 2 approaches are considered and discussed. The first one (Figure 3a) is the simple leaf parallelization, where one GPU is dedicated to one MCTS tree and each GPU thread performs an independent simulation from the same node. Such a parallelization should provide much better accuracy when the great number of GPU threads is considered. The second approach (Figure 3c), is the proposed in this paper block parallelization method. It comprises both leaf and root parallel schemes. Root parallelism (Figure 3b) is an efficient method of parallelization MCTS on CPUs. It is more efficient than simple leaf parallelization[3][4], because building more trees diminishes the effect of being stuck in a local extremum and increases the chances of finding the true global maximum. Therefore having n processors it is more efficient to build n trees rather than performing n parallel simulations in the same node. In Figure 2 we present our analysis of such an approach. Given that a problem can have many local maximas, starting from one point and performing a search might not be very accurate. Then we should spend more time on the search to build a tree then - that is the sequential MCTS approach, the most basic case. The second one, leaf parallelism should diminish this effect by having more samples from a given point while keeping the depth at the same level. In theory, it should make the search broader. The third one is root parallelism. Here a single tree has the same properties as each tree in the sequential approach except for the fact that there are many trees and the chance of finding the global maximum increases with the number of trees. We can consider those trees as starting points in optimization. The last, our proposed algorithm, combines those two, so each search should be more accurate and less local at the same time.

2.0.5. Leaf-parallel scheme

This is the simplest parallelization method in terms of implementation. Here GPU receives a root node from the CPU controlling process and performs n simulations, where n depends on the dimensions of the grid (block size and number of blocks). Afterwards

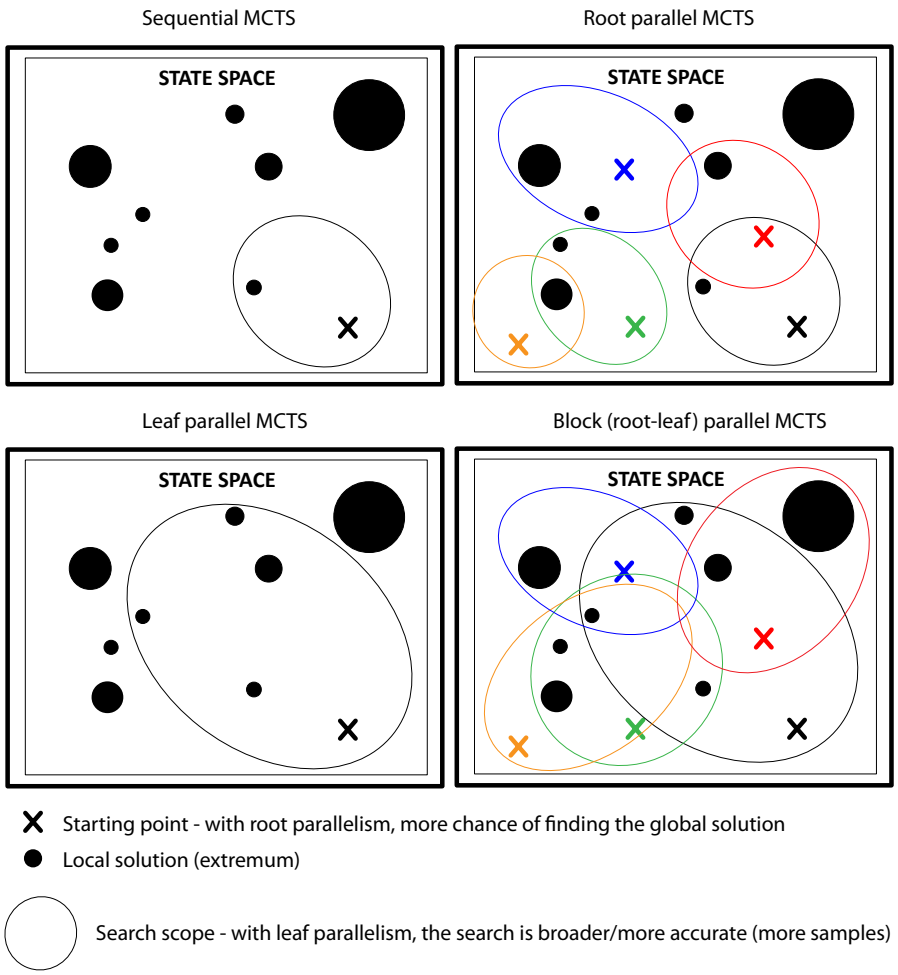

Figure 2. Block parallelism's theoretical impact on the performance

the results are written to an array in the GPU's memory (0 = loss, 1 = victory) and CPU reads the results back.

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

2.0.6. Block-parallel scheme

To maximize the GPU's simulating performance some modifications had to be introduced. In this approach the threads are grouped and a fixed number of them is dedicated to one tree. This method is introduced due to the hierarchical GPU architecture, where threads form small SIMD groups called *warps* and then these *warps* form *blocks*(Figure 4). It is crucial to find the best possible job division scheme for achieving high GPU performance. The trees are still controlled by the CPU threads, GPU simulates only. That means that at each simulation step in the algorithm, all the GPU threads start and end

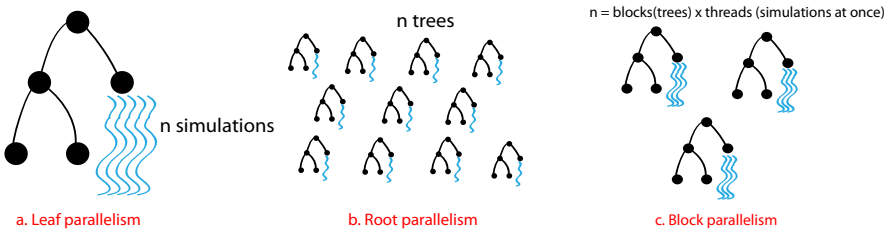

Figure 3. An illustration of considered schemes

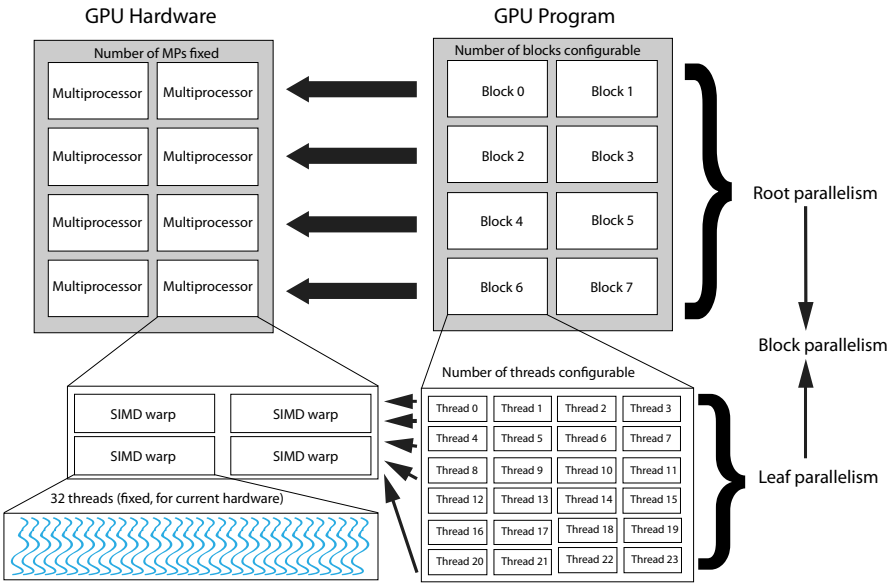

Figure 4. Correspondence of the algorithm to hardware

simulating at the same time and that there is a particular sequential part of this algorithm which decreases the number of simulations per second a bit when the number of blocks is higher. This is caused by the necessity of managing each tree by the CPU, therefore the more blocks exist, the more time without actually simulating is spent. On the other hand the more the tree, the better the performance. Naturally, a compromise has to be established. In our experiments the smallest number of threads used is 32 which corresponds to the warp size.

2.1. Hybrid CPU-GPU processing

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

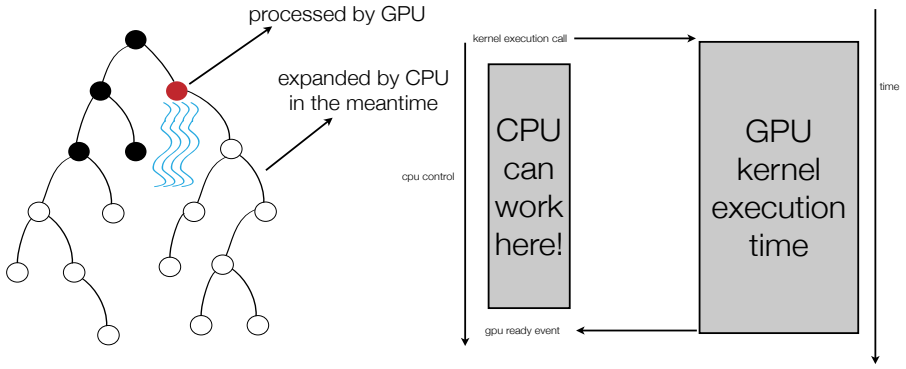

Figure 5. Hybrid CPU-GPU processing scheme

accurate, since the CPU tree grows faster in the direction of the optimal solution. As a solution we experimented on using hybrid CPU-GPU algorithm (Figure 5). In this approach, the GPU kernel is called asynchronously and the control is given back to CPU. Then CPU operates on the same tree (in case of leaf parallelism) or trees (block parallelism) to increase their depth. It means that while GPU processes some data, CPU repeats the MCTS iterative process and checks for the GPU kernel completion.

3. Results and analysis

Here the results are presented. Our test platform is one TSUBAME 2.0 node equipped with a NVIDIA TESLA C2050 GPUs (14 (Multiprocessors) x 32 (Cores/MP) = 448 (Cores) @ 1.15 GHz)) and Intel Xeon X5670 CPUs. In all cases the search time is 500ms. We compare the speed(Figure 6) and results(Figure 7) of leaf parallelism and block parallelism using different block sizes. The block size and their number corresponds to the hardware's properties. In those graphs a *GPU Player* is playing against one CPU core running sequential MCTS. The main aspect of the analysis is that despite running fewer simulations in a given amount of time using block parallelism, the results are much better compared to leaf parallelism, where the maximal winning ratio stops at around 0.75 for 1024 threads (16 blocks of 64 threads). Also the other important thing to observe is the block size comparison. In fact the results are better when the block size is smaller (32), but only when the number of threads is small (up to 4096, 128 blocks/trees), then the lager block case(128) performs better. In case of running 14336 threads, there are 448 and 112 trees respectively. It can be observed in Figure 6 that as we decrease the number of threads per block and at the same time increase the number of trees, the number of simulations per second decreases. This is due to the CPU's sequential part. So there is a trade-off between the number of trees and simulation speed. Both factors affect the performance. We found the block size of 128 to be optimal.

In Figure 8 and 9 we also show a different type of result, where the X-axis represents current game step (in Reversi maximally 60) and the Y-axis is the average point difference, meaning our score minus the opponent's score. In situation when the point difference is greater than 0 is a winning one. In Figure 8 we observe that one GPU outperforms 256 CPUs in terms both intermediate and final scores. Also we see that the

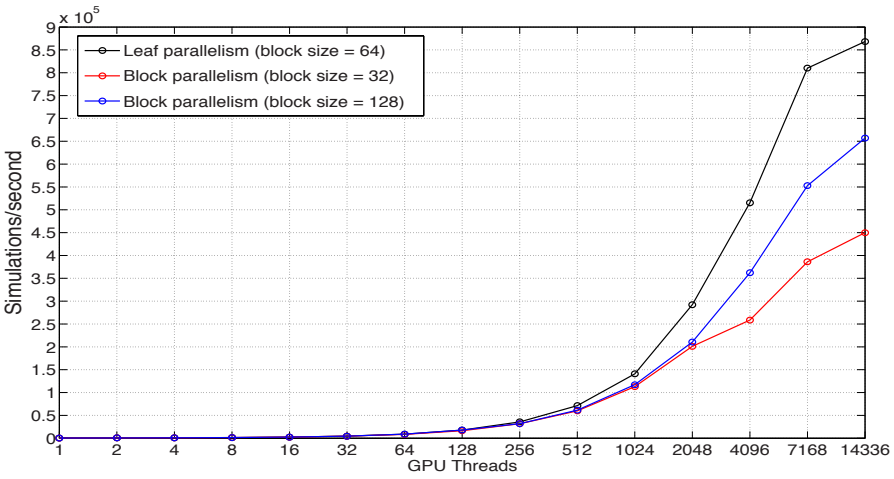

Figure 6. Block parallelism vs Leaf parallelism, speed

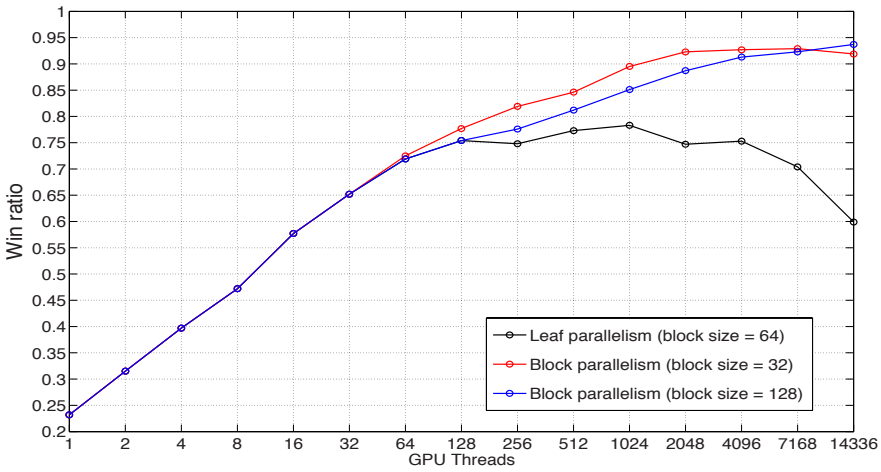

Figure 7. Block parallelism vs Leaf parallelism, final result

characteristics of the results using CPUs and GPU are slightly different, where GPU is stronger at the beginning. We believe that it might be caused by the larger search space and therefore we conclude that later the parallel effect of the GPU is weaker, as the number of distinct samples decreases. Another reason for this is mentioned depth of the tree which is lower in the GPU case. We present this in Figure 8.

Also we show that using our hybrid CPU/GPU approach both the tree depth and the result are improved as expected especially in the last phase of the game. This is interesting also, because of the fact that earlier game step's tree depth affect the final game stage's result so much. We also expect better energy efficiency with the GPU/CPU approach.

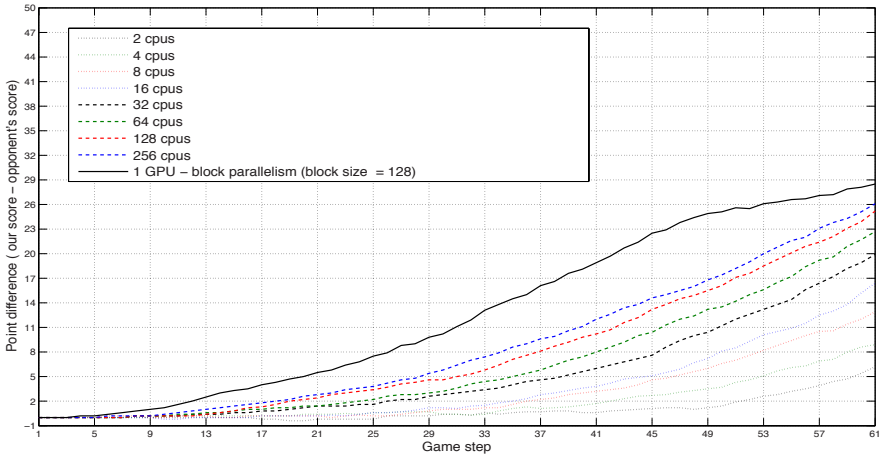

Figure 8. GPU vs root-parallel CPUs

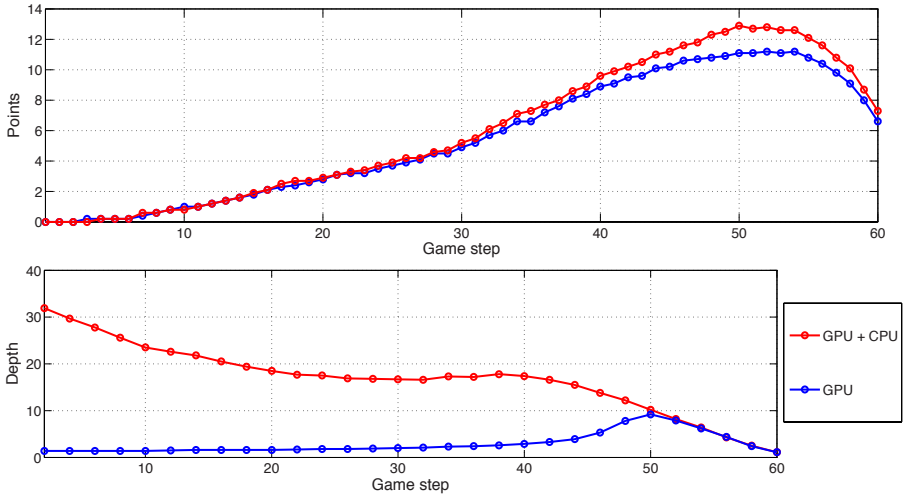

Figure 9. Hybrid CPU/GPU vs GPU-only processing

4. Conclusion

We introduced an algorithm called *block-parallelism* which allows to efficiently run Monte Carlo Tree Search on GPUs achieving results comparable with a hundred of CPU cores(Figure 8). Block-parallelism is not flawless and not completely parallel as at most one CPU controls one GPU, certain part of the algorithm has to be processed sequentially which decreases the performance. We show that block-parallelism performs better that leaf-parallelism on GPU and probably is the optimal solution unless the hardware limitations are not changed. We also show that using CPU and GPU at the same time we get better results. The are challenges ahead, such as unknown scalability and universality of the algorithm.

5. Future work

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

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

Modifying the move selection criteria The moves do not necessarily have to be chosen based on the proportion of wins/losses. The other choices are, the average score or voting methods, such as the majority voting method.

Implementation of a PRNG (parallel random number generator) PRNG based random numbers' sequences' generation. This would make the whole algorithm more robust and the results more reliable for a large number of threads. We also think that this might be one of the factors affecting overall scalability for millions of threads

Power usage analysis Currently we are measuring the energy consumption of both GPU and CPU. Our goal is to make the algorithm more energy-efficient.

Acknowledgements

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

References

[1] Monte Carlo Tree Search (MCTS) research hub, http://www.mcts-hub.net/index.html
[2] Kocsis L., Szepesvari C.: Bandit based Monte-Carlo Planning, 15th European. Conference on Machine Learning Proceedings, 2006
[3] Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik: Parallel Monte-Carlo Tree Search, Computers and Games: 6th International Conference, 2008
[4] Rocki K., Suda R.: Massively Parallel Monte Carlo Tree Search, Proceedings of the 9th International Meeting High Performance Computing for Computational Science, 2010
[5] Coulom R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search, 5th International Conference on Computer and Games, 2006
[6] Warren H.S.: Hacker's Delight, Addison Wesley, 2002
[7] Romaric Gaudel, MichŔle Sebag - Feature Selection as a one-player game (2010)
[8] Guillaume Chaslot , Steven Jong , Jahn-takeshi Saito , Jos Uiterwijk - Monte-Carlo Tree Search in Production Management Problems (2006)
[9] O. Teytaud et. al, High-dimensional planning with Monte-Carlo Tree Search (2008)
[10] Maarten P.D. Schadd, Mark H.M. Winands, H. Jaap van den Herik, Guillaume M.J-B. Chaslot, and Jos W.H.M (2008)