Parallel Minimax Tree Searching on GPU

Kamil Rocki and Reiji Suda

JST CREST, Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
{kamil.rocki,reiji}@is.s.u-tokyo.ac.jp

Abstract. The paper describes results of minimax tree searching algorithm implemented within CUDA platform. The problem regards move choice strategy in the game of Reversi. The parallelization scheme and performance aspects are discussed, focusing mainly on warp divergence problem and data transfer size. Moreover, a method of minimizing warp divergence and performance degradation is described. The paper contains both the results of test performed on multiple CPUs and GPUs. Additionally, it discusses $\alpha\beta$ parallel pruning implementation.

1 Introduction

This paper presents a parallelization scheme of minimax algorithm[10], which is very widely used in searching game trees. Described implementations are based on the Reversi (Othello) game rules. Reversi is a two-person zero-sum game. Each node of the tree represents a game state and its children are the succeeding states. The best path is the path providing the best final score for a particular player. The desired search depth is set beforehand, each node at that depth is a terminal node. Additionally, each node, which does not have any children, is a terminal node too.

2 Implementation

2.1 Basic Minimax Algorithm

The basic algorithm used as a reference is based on the recursive *minimax* (negamax) tree search.

function minimax(node, depth)

if node is a terminal node or depth = 0
    return the heuristic value of node
else $\alpha \leftarrow -\infty$
foreach child of node
    $\alpha \leftarrow max(\alpha, -minimax(child, depth - 1))$

return α

R. Wyrzykowski et al. (Eds.): PPAM 2009, Part I, LNCS 6067, pp. 449–456, 2010.

2.2 Iterative Minimax Algorithm

CUDA does not allow recursion, therefore the GPU minimax implementation uses modified iterative version of the algorithm [1].

2.3 Parallelization

The main problem which arises regarding parallelization is the batch manner of GPU's processing. The jobs cannot be distributed on the fly and additionally, the communication cost is high. The number of CPU to GPU transfers needs to be minimized, large amount of data has to be processed at once. The idea of the work distribution for thousands of GPU threads is based on splitting the main tree into 2 parts: the upper tree of depth s processed in a sequential manner and the lower part of depth p processed parallelly. The lower part of is then sliced into k subtrees, so that each of the subtree can be searched separately. Thus, the tree search process is divided into 3 steps: 1. Sequential tree search starting with the root node of depth s. Store the leaves to a defined array/list of tasks.

2. For each of the stored tasks execute parallel search of depth p. After the search is finished, store search results (score) to a defined array/list of results. If $k > GPUthreads$ then, several sequential steps are needed. That is solved, by creating a queue of tasks (nodes). GPU would load nodes from the queue until there are some tasks present.

3. Execute sequential tree search once more (like in step 1) andafter reaching leaf node, read stored results (in step 2) from the array/list and calculate main nodes score.

Parallelization brings an overhead of executing the sequential part twice. Therefore, proper choice of values p and s is very important. Depth s should be minimized to decrease that cost, while producing enough leaves (value k)

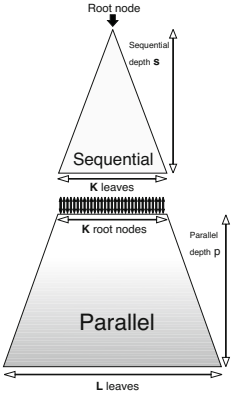

Fig. 1. Parallelization

to keep every GPU thread busy. Hence, following condition should be met: ($k \geq threads$). The value k cannot be determined beforehand due to variable branching factor of each tree. One way of solving the case is estimation and the other is unrolling the root node to the moment when the condition is met, which means searching the tree gradually deeper until k is large enough.

2.4 Modifications

Straightforward minimax port to the CUDA code and parallel execution did not show significant speedup compared to the CPU sequential execution time. One of the reasons for weak performance is the SIMD way of data processing for each warp. The main feature of considered group of trees is the variable branching factor which relies on the actual game state. The approximate average branching factor of a single tree node was calculated to be 8,7.

2.5 Warp Divergence Problem

Every thread of each warp performs the same operation at given moment. Whenever the change of control operation within one warp is present (such as if-else instructions), 'warp divergence' occurs. That means, that all the threads have to wait until if-else conditional operations are finished. Each root node processed in parallel manner represents different board state, therefore every searched tree may have a different structure. Iterative tree searching algorithm is based on one main outer loop and 2 inner loops (traversing up and down). Depending on number of children for each node, the moment of the loop break occurrence varies, and therefore that causes the divergence in each warp. The ideal case, when no divergence is present, was estimated by calculating average time of searching same root node by each thread. The result proved that reducing divergence, could significantly improve the search performance (reduce the time needed).

2.6 Improving the Performance

A method of reducing warp divergence has been implemented. Decreasing the number of different root nodes in each warp minimizes the divergence, but also results in longer processing time. A way of achieving the parallel performance and keeping low warp divergence is to modify the algorithm, so that the parallel processing parts are placed where no conditional code exists and allow warps to process shared data (each warp should processes one root node). In analyzed case each node represents a game board. Each game board consists of 64 fields. Processing a node is:

1. Load the board. If node is not terminal, then for each field:
    a. If move is possible, generate a child node
    b. Save list of children (after processing each field)
2. Else calculate the score then for each field:

a. Get value (score)
b. Add or subtract from the global score depending on the value and save board score

Operations performed for each field can be done in parallel. Because each warp consists of 32 threads, it is not difficult to distribute the work. Each thread in a warp searches the same tree, while analyzing different node parts. In order to implement described method, both the shared memory and the atomic operations have to be used. Each warp has common (shared) temporary memory for storing children generated by the threads and for storing the score. CUDA atomic operations are needed for synchronizing data access of each thread. I.e. adding a value to shared score memory involves reading the memory, adding value and then storing the updated score. Without such a synchronization one thread could easily overwrite a value, previously store by the other one. Warps can also be divided into smaller subgroups to dedicate fewer threads to one node, i.e. if each 16 threads processes one node, then one warp searches 2 boards in parallel. Further, the results of dividing the warps into groups of 8 and 16 are also presented.

Summarizing, for each node, the modified algorithm works as follows:

1. Read node
2. If node is a leaf, calculate score in parallel - function *calculateScore*(). Join (synchronize) the threads, return result
3. Else generate children in parallel - function *getChildren*(). Join (synchronize) the threads, save children list
4. Process each child of that node

Described procedure decreases thread divergence, but does not eliminate it, still, some flow control instructions are present while analyzing possible movement for each board field.

Many papers describe load balancing algorithm usage in detail in order to increase the performance. Here the GPU's Thread Execution Manager performs that task automatically. Provided that a job is divided into sufficiently many parts, an idle processor will be instantly fed with waiting jobs.

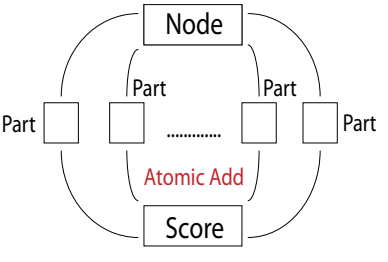

Fig. 2. Score calculation

3 Results

Each value obtained is an average value of searching 1000 nodes, being a result of continuously playing Reversi and moving at random thus the result may be an approximation of the average case.

3.1 CPU

First, the basic parallelized algorithm was tested on a 8-way Xeon E5540 machine to check the correctness and to obtain a reference for further GPU results. Graph shows the advantage of using several CPU cores to search tree of depth 11, where 7 levels are solved sequentially and 4 levels in a parallel manner. In this case, each CPU thread processed one node from a given node queue at once.

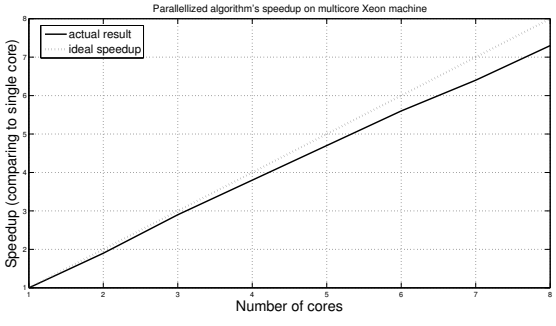

Fig. 3. CPU results

3.2 GPU

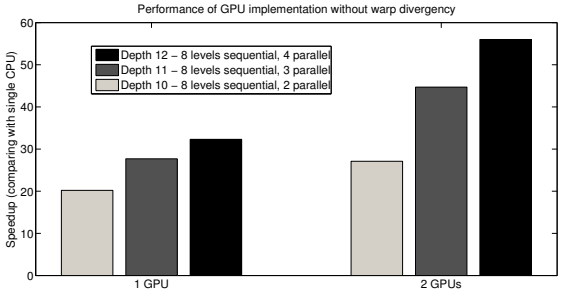

Fig. 4. GPU results - no divergence

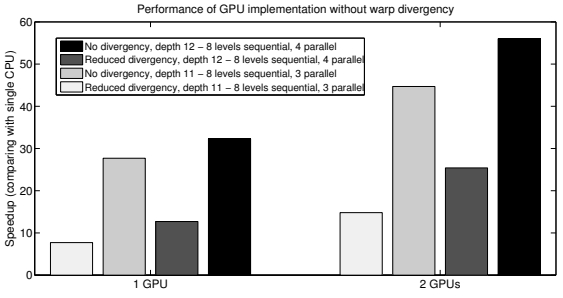

Fig. 5. GPU results

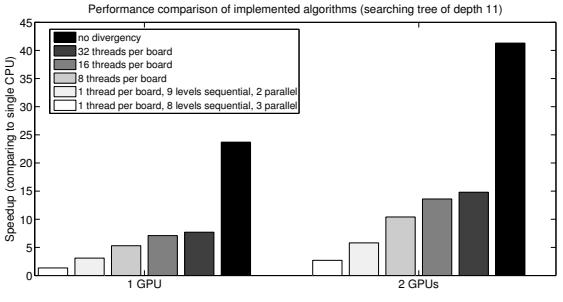

Fig. 6. GPU results

GPU tests were performed on a machine equipped with a 2.83 GHz Q9550 Intel CPU and 2 GeForce 280 GTX GPUs. To ensure that GPU is busy, the configuration was set to 128 threads per block (4 warps) and 240 blocks in total which equals 128 * (240 / 30 processors) = 1024 threads and 8 blocks per MP (physical multiprocessor) - maximum that can work concurrently on GTX 280. Here: 1 core time for level 11 $\simeq 1070s$, and for level 10 $\simeq 115s$.

4 Analysis

CPU results show that the overhead of the parallel algorithm compared to the standard one is minimal. GPU results present significant improvement in the case when no divergence is present (average speed of searching same node by each thread). I.e., when sequential depth is set to 8 and parallel depth is set to 2 we observe 20x (20.2) speedup, then for value of 3 nearly 27x (26.7) speedup for single GTX 280 GPU and over 30x (32.6) for parallel depth of 4. This can be explained by communication cost overhead, solving more on the GPU at once saves time spend on transferring the data. Algorithm also scales well into 2 GPU giving 55x (55.2) speedup for 2 GPUs in the best case of parallel depth 4.

When different nodes/board are analyzed by each thread then the results are much worse in the basic case showing only 1.4x speedup for a single GPU

and 2.7x for 2 GTXs. These are the results for sequential/parallel depth of 8/3. The performance increases slightly when parallel depth is decreased to 2 and sequential is increased to 9. Then the observed speedup for a single GPU was 3x and 5.8 for 2 GPUs. Nevertheless, such a way of improving the effectiveness is not applicable when sequential depth is greater (too many nodes to store) and the increase in performance is still insignificant.

Analyzing the reduced thread divergence version of the algorithm, we can observe that the performance increases as number of threads dedicated to solve node grows. For single GPU 5.3/7.1/7.7 and for 2 GPUs 10.5/13.6/14.8 speedups are noticed respectively. Still it is only approximately 1/3 of the ideal case, nevertheless performance is much better than the basic algorithm's one. Moreover, increasing the parallel depth from 3 to 4 also effected in the performance increase as in the no divergence example.

Second important factor observed was the data turnaround size. Tree searching is a problem, where most of the time is spent on load/store operations. Just to estimate the data size for tree of depth 11, if the average branching factor equals 8.7: Average number of total nodes in a tree $\sum_{k=0}^{10} k = (8.7)^k \simeq 2.8 * 10^9$. In the presented implementation mostly bitwise operations were used to save space for a single node representation occurring in 16B memory usage for each node/board. Therefore at least $16B * 2.8 * 10^9 = 44.8GB$ data had to be loaded/stored (not including the other data load while analyzing the board). While implementing the algorithm, decreasing the size of a single node from a naive 64B board to bit-represented one [3] increased the overall speed $\sim$10 times.

5 Possible Improvements

Further warp divergence decrease could effect in an additional speedup. A general solution for every non-uniform tree could be branch reordering, so that search process starts in a branch with largest number of children and ends in the one with the smallest number. If a tasks cannot be parallelized in the way presented in this paper, a method of improving SIMD performance with MIMD algorithms is described [8][9]. Parallel node pruning (i.e. $\alpha\beta$) algorithm might be implemented [2][4][5][6], however the batch way of GPU processing implies serious difficulties in synchronizing the jobs processed in parallel. Moreover, the inter-thread communication is very limited (only within the same block). In some papers, SIMD $\alpha\beta$ implementations are presented, but they differ from this case. I.e. in [6] no particular problem is being solved, moreover the tree is synthetic with constant branching factor.

6 Conclusions

Considering the results obtained, GPU as a group of SIMD processors performs well in the task of tree searching, where branching factor is constant. Otherwise, warp divergence becomes a serious problem, and should be minimized. Either by parallelizing the parts of algorithm that are processed in a SIMD way or by

dividing main task into several smaller subtasks [8][9]. Direct implementation of the algorithm did not produce any significant improvement over the CPU sequential execution or effected in even worse performance. GPU outperforms a single CPU if high level of parallelism is achieved (here many nodes have to be searched in parallel). For 4 levels searched in parallel, a single GTX 280 GPU performs the task approximately 32 times faster than single CPU core if no divergence is present. 2 GPUs give 55x speedup factor (that is 72% faster than a single GPU). Regarding the modified algorithm, the values are 7.7x speedup for a single GPU and 14.8x for 2 GTXs (92% faster). That shows that the algorithm could be scaled to even larger amount of GPUs easily. One of the main disadvantages of the GPU implementation is that modifications like $\alpha\beta$ pruning are hard to implement. Limited communication and necessity to launch all the thread at once cause the difficulty. Another aspect of analyzed approach is possibility of concurrent GPU and CPU operation. While GPU processes bigger chunks of data (thousands) at once, each CPU searches other nodes that are waiting in a queue.

References

[1] Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence 6, 293–326 (1975)
[2] Manohararajah, V.: Parallel Alpha-Beta Search on Shared Memory Multiprocessors, Master Thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto (2001)
[3] Warren, H.S.: Hacker's Delight. Addison-Wesley, Reading (2002)
[4] Schaeffer, J.: Improved Parallel Alpha Beta Search. In: Proceedings of 1986 ACM Fall Joint Computer Conference (1986)
[5] Borovska, P., Lazarova, M.: Efficiency of Parallel Minimax Algorithm for Game Tree Search. In: Proceedings of the International Conference on Computer Systems and Technologies (2007)
[6] Schaeffer, J., Brockington, M.G.: The APHID Parallel $\alpha\beta$ algorithm. In: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing, p. 428 (1996)
[7] Hewett, R., Ganesan, K.: Consistent Linear speedup in Parallel Alpha-beta Search. In: Proceedings of the ICCI 1992, Fourth International Conference on Computing and Information, pp. 237–240 (1992)
[8] Hopp, H., Sanders, P.: Parallel Game Tree Search on SIMD Machines. In: Ferreira, A., Rolim, J.D.P. (eds.) IRREGULAR 1995. LNCS, vol. 980, pp. 349–361. Springer, Heidelberg (1995)
[9] Sanders, P.: Efficient Emulation of MIMD behavior on SIMD Machines. In: Proceedings of the International Conference on Massively Parallel Processing Applications and Development, pp. 313–321 (1995)
[10] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, pp. 163–171. Prentice Hall, Englewood Cliffs (2003)