Journal of Computing and Information Technology - CIT 19, 2011, 4, 269–274
doi:10.2498/cit.1002029

Parallel Alpha-Beta Algorithm on the GPU

Damjan Strnad and Nikola Guid

Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia

In the paper we present the parallel implementation of the alpha-beta algorithm running on the graphics processing unit (GPU). We compare the speed of the parallel player with the standard serial one using the game of reversi with boards of different sizes. We show that for small boards the level of available parallelism is insufficient for efficient GPU utilization, but for larger boards substantial speed-ups can be achieved on the GPU. The results indicate that the GPU-based alpha-beta implementation would be advantageous for similar games of higher computational complexity (e.g. hex and go) in their standard form.

Keywords: alpha-beta, reversi, GPU, CUDA, parallelization

1. Introduction

The alpha-beta algorithm is central to most engines for playing the two player zero-sum perfect information board games [10]. Its efficiency stems from the pruning of the game tree that relies heavily on the serial interdependence of tree traversal. This sequential nature of the algorithm has also proved to be the major culprit for its effective parallelization, which has been the focus of many researchers in the past [3, 6, 13]. The noticeable speed-ups have been achieved in simulation [7], but required special hardware [11, 12, 21] or network configurations [4, 19] to be demonstrated in practice. With multi-threaded solution on broadly accessible multi-core CPUs, a relatively modest acceleration has been obtained [2].

In this paper, we present the parallel alpha-beta algorithm running on the graphics processing unit (GPU). Modern GPUs are massively parallel processors residing on the graphics boards installed in common desktop computers and laptops. They feature hardware accelerated scheduling of thousands of lightweight threads running on a number of streaming multiprocessors, resembling in operation the SIMD (single instruction multiple data) machines. In recent years, the parallel architecture of the GPU has become available for general purpose programming not related to graphics rendering. This was facilitated by the emergence of specialized toolkits like C for CUDA [16] and OpenCL [8] which propelled the successful parallelization of many compute intensive tasks [15].

Our goal with the GPU-based alpha-beta implementation is to compare its speed of play against the basic serial version of the algorithm. The game of reversi (also known as Othello) serves as a testing ground for comparison, where different board sizes are used to vary the complexity of the game. This allows us to set the future directions of research on more complex games that exhibit higher level of exploitable parallelism.

The GPU-based implementation of game tree search has previously been reported by Bleiweiss, but he demonstrated significant speed-ups only for practically limited case of thousands of simultaneously running games [1]. Another report shows good acceleration results with the exhaustive minimax tree search, which represents the foundation of the alpha-beta algorithm [17]. In their work, Rocki and Suda indicate some intricacies of the GPU architecture that impair the performance of parallel minimax and are inherited by the alpha-beta.

The rest of the paper is organized as follows: in Section 2 we describe the principles of general purpose computation on graphics proces-

sors, followed by the description of alpha-beta parallelization methodology and its GPU-based implementation in Section 3. In Section 4 we give the specification of testing details and in Section 5 the results are presented. In Section 6 we conclude and present some ideas for future work.

2. General Purpose GPU Programming and CUDA

In our research the GPU programming toolkit C for CUDA by Nvidia has been used because it is the most mature and well documented [18]. Here we give a short overview of the technology.

The parallel architecture of Nvidia GPUs consists of a set of pipelined multiprocessors. The parallel computation on the GPU is performed as a set of concurrently executing thread *blocks*, which are organized into a 1D or 2D *grid*. The blocks themselves can be 1D, 2D, or 3D with each thread designated by a unique combination of indices. The hardware schedules the execution of blocks on the multiprocessors in units of 32 threads called *warps*. The threads from the same warp run in a lockstep, with each divergent branching in the code suspending some of the threads and thereby reducing the rate of parallelism. The hierarchical structure of GPU processing is also expressed in the necessity for careful thread synchronization to avoid the racing conditions and optimize the performance.

In CUDA, the C-like code to be executed on the GPU is written in the form of functions called *kernels*. For the efficient implementation of kernels on the GPU, one must consider the limited amount of available resources like on-chip memory and registers, as well as restrictions on *execution configuration* (i.e. the dimensions of grid and blocks). The properties of the GPU are described by its *compute capability*, which can be queried at run-time and used to adjust the kernel parameters.

The memory available to the GPU is of several types. Each thread block has at disposal a limited amount (16KB – 48KB) of fast local storage called *shared* memory, which resides on the GPU and has low latency. It is used for internal computation and communication between the threads of the block. The off-chip, on-device memory comes in much larger quantities (1GB and more) and can be used for inter-block communication, but has a high latency. It is further divided into read-only *texture* and *constant* memory, and read-write *global* memory. The purpose and characteristics of each memory type are detailed in the documentation [9].

The usual template of operation in CUDA kernels is to copy the data from global to shared memory, process it there, and copy the results back. All of these steps are performed in parallel. The kernel complexity in terms of shared memory usage and register count determines the number of blocks that can be simultaneously scheduled on the multiprocessor and thereby affects the GPU occupancy. The expected GPU occupancy can be calculated using the tools provided with CUDA so that the execution configuration and kernel compilation can be tuned for optimal utilization of resources. The higher GPU occupancy allows the scheduler to swap the warps waiting for global memory transfer with the queued warps.

Another major impact on kernel performance is the way threads in a warp address the memory locations. The desired addressing scheme uses *coalesced* memory reads and writes, which means that consecutive threads access sequential memory locations or at least a permutation of them. Conflicting memory accesses are serialized and lower the memory bandwidth.

3. Implementation of Parallel Alpha-beta on the GPU

The main source of parallelism in the alpha-beta algorithm is in the concurrent processing of multiple sibling nodes at any level of the game tree. If the best move is evaluated first, then the rest of the moves can be refuted in parallel with maximal efficiency. Unfortunately, the a priori quality of move can only be heuristically estimated. The parallel processing of nodes therefore usually introduces some *search overhead* which may surpass the amount of serial search in case of unfavourable move order.

The basis for our parallel alpha-beta implementation is the PV-split algorithm [14], in which the parallelism is employed at the nodes on the *principal variation* (i.e. the leftmost path in the

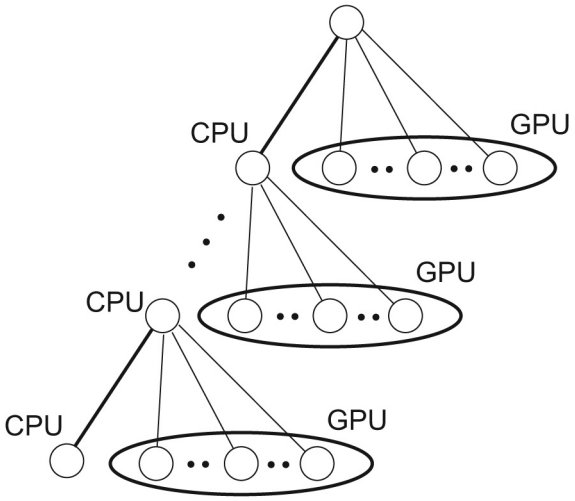

Figure 1. GPU-based PV-split

heuristically ordered game tree), also known as the *PV-nodes* or *type 1 nodes* [10]. In the GPU-based variant the leftmost child of each PV-node is searched on the CPU to establish the lower bound on the PV-node value. The rest of PV-node's descendants are searched in parallel on the GPU using the narrower window. Figure 1 shows the principle of parallel execution in the PV-nodes.

The parallel processing of sibling nodes is realized using multiple thread blocks on the GPU, but this is only the high level of parallelization exerted by the GPU-based alpha-beta. On the lower level, each node (i.e. reversi board) is processed in parallel by the threads of the two-dimensional block where a single block is used per node. Figure 2 depicts the described two level parallelism which differs from the approach by Rocki and Suda, who delegate the nodes to individual warps of the same block [17].

The dimension of the block may be smaller than that of the board because:

- using the smaller block is desirable when it results in higher GPU occupancy, and
- the number of threads in the block is limited to 1024, which requires the use of smaller blocks for boards larger than 32×32 squares.

If the block and board are of the same dimension, then the thread with index pair (x, y) is mapped directly to square (x, y) of the board. If the block is smaller than the board, then the mapping scheme shown in Figure 3 is employed. In such scheme each thread processes

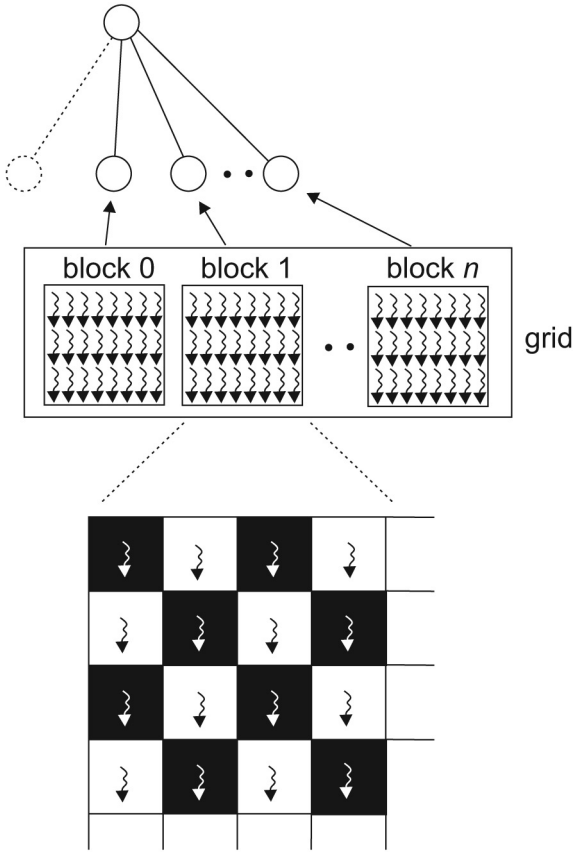

Figure 2. Two level parallelism of GPU

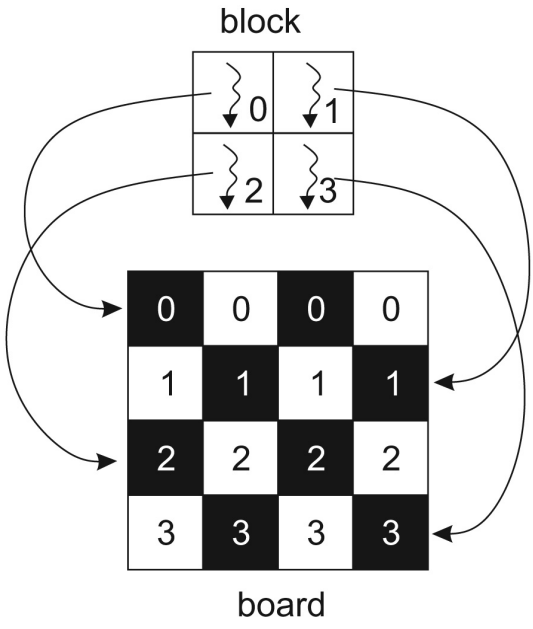

Figure 3. The mapping of threads to consecutive board squares

the appropriate number of consecutive squares which ensues in nicely coalesced memory accesses on byte-sized arrays.

Traditionally, the alpha-beta algorithm is implemented as a recursive procedure, but only the latest generation of graphics cards supports recursion. We therefore implemented the iterative alpha-beta version using the manually controlled stack of nodes in the shared memory. The parts of the alpha-beta algorithm that are executed in parallel by all of the threads in a block are node evaluation, move generation, and move execution. The remaining administrative work such as checking for cuts and score updating are performed by a single thread.

Thread divergence is the main cause of degraded efficiency in the parallel move generation, where each thread marks the corresponding square as a valid move, and in parallel move execution, where each thread determines the flipping of the corresponding square. In the board evaluation phase, each thread separately evaluates the corresponding square, after which the board value is obtained by summing the square values using the method of parallel *reduction* [18].

4. Testing Setup

The well known game of reversi was used as testing application to compare the standard serial alpha-beta algorithm and its parallel adaptation for the GPU. The comparison was performed by measuring the total time spent by the serial and parallel player when playing two symmetric rounds of the reversi game, once as black and once as white. For smaller search depths the two plays were mostly identical. For deeper searches the parallel player sometimes chose a different, but equivalent move in the middlegame, which resulted in slight difference in the end result. The ratio of CPU vs. GPU time is used as an estimation of achieved speedup. In the GPU case, the measurements include kernel execution as well as all of the memory transfers from the host to the device and back.

Reversi is a suitable game for the implementation on the GPU because of its simple rules. However, on standard 8×8 board it provides a weak workload for the GPU. The complexity of the game can easily be increased by using a larger board, as shown in Table 1 which lists the average branching factors of reversi game trees achieved in our experiments with search depth 2.

Board size	b
8×8	7.90
16×8	10.84
16×16	15.79
32×16	18.46
32×32	34.04
64×32	72.60
64×64	78.30

Table 1. Average branching factors

Our implementation supports non-square boards with power-of-two lateral dimensions up to 64 squares. While boards of that size are unlikely to be used for human play, they present no problem for computer tournaments. Similar, but more complex games than reversi also exist, which compete in complexity with the oversized reversi version (e.g. hex, gomoku, and go). The results obtained on large reversi boards thus indicate the possible qualities of GPU-based implementations of those games.

The heuristics of the tested implementations were the same, incorporating the weighted piece counter and mobility metric as components of the evaluation function [5]. The weight matrix used for piece evalution was the one obtained with coevolution by Szubert [20]. The same weight matrix was also used in move ordering to select the first move in the PV-nodes on the CPU.

The testing was performed on an idle computer with quad-core Intel i5 CPU and 4 GB of system memory. The system had two graphics boards, where one was used for the display and the other one was Nvidia 480 GTX GPU with 1.5 GB of video memory used for the computation.

5. Results and Analysis

We present the acceleration factors for the reversi boards with lateral sizes given in Table 1. The size of the thread blocks on the GPU was the same as the size of the board except for the largest three boards, where it was accordingly set to maximize the GPU occupancy.

Board dimension	Search depth							
	1	2	3	4	5	6	8	10
8 × 8	0.136	0.217	0.444	0.622	0.722	0.492	0.396	0.295
16 × 8	0.246	0.381	0.730	0.821	0.798	0.944	0.702	–
16 × 16	0.454	0.666	1.219	1.000	0.875	0.809	–	–
32 × 16	0.853	1.183	2.276	1.370	1.031	–	–	–
32 × 32	1.213	2.017	4.469	2.275	–	–	–	–
64 × 32	1.492	3.681	4.632	–	–	–	–	–
64 × 64	1.627	6.095	23.647	–	–	–	–	–

Table 2. The GPU acceleration factors

We used the search depths from 1 to 10 for the smallest board and from 1 to 3 for the largest one. Table 2 shows the ratios of CPU vs. GPU times (i.e. the acceleration factors) when playing on boards of various sizes. The values of acceleration factors below 1 denote the better performance of the CPU.

Figure 4 graphically depicts the data from Table 2 on logarithmic scale and emphasizes the tendency of the GPU to progressively outperform the CPU on larger boards.

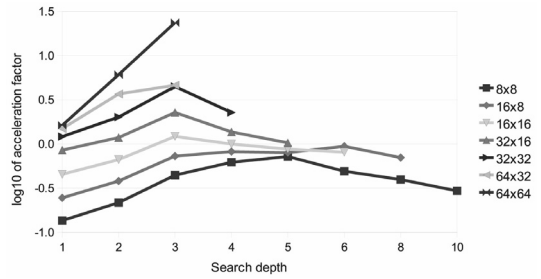

Figure 4. The plots of data from Table 2 on logarithmic scale

Based on the analysis of the observed results, we established the following findings:

- the GPU resources are underused on smaller boards, where the search overhead and memory transfers take up a significant portion of measured time,
- on larger boards the high pruning ratio of many equivalent successors allows for multiple parallel acceleration,
- with increasing search depth the limits of GPU resources are met, which results in relative degradation of parallel efficiency.

6. Conclusions and Future Work

In this paper we presented the parallel implementation of the alpha-beta algorithm on the Nvidia GPU using the CUDA framework. We demonstrated that substantial speed-ups can be achieved with the GPU-based algorithm in the game of reversi as the board size increases. This lets us surmise that similar games of much higher complexity, in particular the game go, should be amenable to efficient parallelization on the GPU. The expansion of research to these games is therefore one of our priority tasks for the future.

Since its inception the alpha-beta algorithm has seen many improvements, some of them general and some tailored to specific game instances, especially chess. To make better comparison with the enhanced serial alpha-beta, we have already analyzed the prospects for the transfer of the most significant improvements to the parallel GPU variant. The implementation of GPU-side transposition tables in combination with iterative deepening is already underway.

Finally, we also intend to look into possible hybrid CPU/GPU solutions with heuristic GPU activation for deeper searches at promising or dynamic leaf positions.

References

[1] A. BLEIWEISS, Playing Zero-Sum Games on the GPU. *GPU Technology Conference*, 2010. `http://www.nvidia.com/content/GTC2010/ /pdfs/2207_GTC2010.pdf` [02/14/2011]

[2] P. BOROVSKA, M. LAZAROVA, Efficiency of Parallel Minimax Algorithm for Game Tree Search. In: B. Rachev, A. Smrikarov, D. Dimov, editors. *Proceedings of the 2007 International Conference on Computer Systems and Technologies*, 2007 Jun 14-15; Rousse, Bulgaria.

[3] M. BROCKINGTON, A Taxonomy of Parallel Game-Tree Search Algorithms. *International Computer Chess Association Journal* 1996, 19(3): 162–174.

[4] M. BROCKINGTON, J. SCHAEFFER, APHID: Asynchronous Parallel Game-Tree Search. *Journal of Parallel and Distributed Computing* 2000, 60(2): 247–273.

[5] M. BURO, The Evolution of Strong Othello Programs. In: R. Nakatsu, J. Hoshino, editors. *First International Workshop on Entertainment Computing*, 2002 May 14-17; Makuhari, Japan; pp. 81–88.

[6] R. FELDMANN, Game Tree Search on Massively Parallel Systems. *Ph.D. Thesis*. Paderborn: University of Paderborn; 1993.

[7] F-H. HSU, Large-Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess. *Ph.D. Thesis*. Pittsburgh: Carnegie Mellon University; 1990.

[8] Khronos Group. OpenCL Overview; 2010. `http://www.khronos.org/opencl/` [02/14/2011]

[9] D.B. KIRK , W-MW. HWU, Programming Massively Parallel Processors: *A Hands-on Approach*. Amsterdam: Morgan Kaufmann; 2010.

[10] D.E. KNUTH, R.W. MOORE, An Analysis of Alpha-Beta Pruning. *Artificial Intelligence* 1975; 6(4): 293–326.

[11] B.C. KUSZMAUL, Synchronized MIMD Computing. *Ph.D. Thesis*. Cambridge: Massachusets Institute of Technology; 1994.

[12] C-PP. LU, Parallel Search of Narrow Game Trees. *M.Sc. Thesis*. Edmonton: University of Alberta; 1993.

[13] V. MANOHARARAJAH, Parallel Alpha-Beta Search on Shared Memory Multiprocessors. *M.Sc. Thesis*. Toronto: University of Toronto; 2001.

[14] T.A. MARSLAND, M.S. CAMPBELL, Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys* 1982, 14(4): 533–551.

[15] H. NGUYEN, EDITOR, GPU Gems 3. New Jersey: Addison Wesley Professional; 2007.

[16] Nvidia. CUDA Zone; 2010. `http://www.nvidia.com/cuda` [02/14/2011]

[17] K. ROCKI, R. SUDA, Parallel Minimax Tree Searching on GPU. In: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski, editors. *Parallel Processing and Applied Mathematics*. Heidelberg: Springer; 2010. pp. 449–456.

[18] J. SANDERS, E. KANDROT, CUDA by Example: *An Introduction to General-Purpose GPU Programming*. New Jersey: Addison Wesley; 2010.

[19] J. SCHAEFFER, Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing* 1989, 6(2): 90–114.

[20] M. SZUBERT, W. JASKOWSKI, K. KRAWIEC, Coevolutionary Temporal Difference Learning for Othello. In: Lanzi PL, editor. *IEEE Symposium on Computational Intelligence and Games*; 2009 Sep 7-10; Milano, Italy; pp. 104–111.

[21] J-C. WEILL, The ABDADA Distributed Minimax-Search Algorithm. *International Computer Chess Association Journal* 1996; 19(1): 3–16.

Received: June 2011
Accepted: November 2011

Contact addresses:

Damjan Strnad
Faculty of Electrical Engineering and Computer Science, University of Maribor
Smetanova ulica 17, 2000 Maribor, Slovenia
e-mail: `damjan.strnad@uni-mb.si`

Nikola Guid
Faculty of Electrical Engineering and Computer Science, University of Maribor
Smetanova ulica 17, 2000 Maribor, Slovenia
e-mail: `guid@uni-mb.si`

DAMJAN STRNAD is working as an assistant professor at the Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia. He received the M.Sc. degree in computer science in 2000 and the Ph.D. in computer science in 2006, both from the University of Maribor. His primary research areas are artificial intelligence, computer graphics, and parallel computing.

NIKOLA GUID is presently a professor of computer science with the Faculty of Electrical Engineering and Computer Science at the University of Maribor, Slovenia, and the head of the Laboratory of Computer Graphics and Artificial Intelligence. He received the M.Sc. degree in computer science from the University of Ljubljana, Slovenia, in 1977, and the Ph.D. degree in computer science from the University of Maribor, Slovenia, in 1984. His current research interests are artificial intelligence, computer graphics, and computer aided geometric design. He is a member of the IEEE and the ACM.