Monte Carlo Tree Search Parallelization Using JCuda

Candice Schumann (La Salle University '15), Brandon Cieslak (Lewis & Clark College '15)

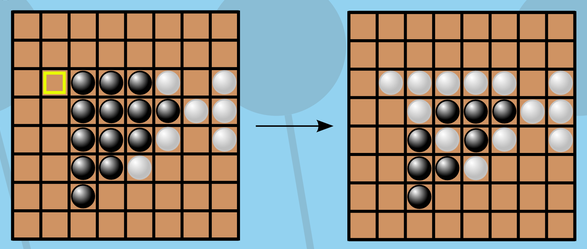
Abstract

Monte Carlo Tree Search (MCTS) is a state of the art for game tree searches. For games with extremely large state spaces, an MCTS player grows stronger as more simulations (playouts) are completed. Using multiple processors in parallel increases the number of playouts that can be completed in a given amount of time.

We are from the Orego research team trying to improve our computer Go player via parallelism. However, before we implement this for Go, we look at what effect GPU (Graphics Proccessing Unit) parallelism has on simpler but similar games. We use the massive parallel processing power of a GPU, using CUDA and JCuda, to speed up MCTS for Othello and Gomoku. Expanding on the techniques described by Rocki [5], we increase the strength and speed of an MCTS Othello player.

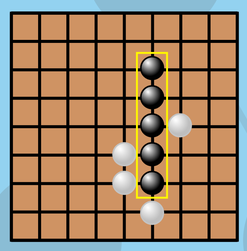
Othello

In Othello there are two players, black and white. The black player always plays first. When a player surrounds enemy pieces on both ends those pieces are turned over to become the opposite color. When there are no moves left for either player, the game is over. The player with the most pieces of its color on the board wins.



Gomoku

Gomoku has similar rules to Tic Tac Toe but the objective is to get five stones in a row.



CUDA

CUDA (Compute Unified Device Architecture), developed by NVIDIA [2], allows a C/C++ program to access the GPU. This access allows a program to exploit the large amount of parallel processing power contained within the GPU.

We use a GeForce GTX 460 NVIDIA card with 336 cores. There are 336 cores grouped into 7 hardware blocks. To avoid idling, we usually use 14 code blocks, each of which has 512 threads. Each code block has access to shared memory which allows communication between threads.

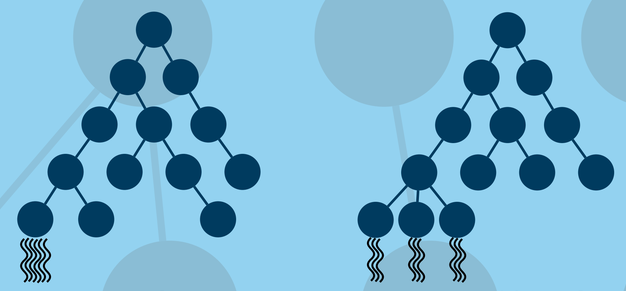
JCuda

JCuda is a Java library that allows access to CUDA [4]. This means that we can access CUDA functions and copy memory to and from the GPU in a Java program.

JCuda is helpful for programmers who are more comfortable with Java. It eases parallelization for existing Java programs. However, JCuda requires more setup than CUDA. Also, JCuda does not allow us to send primitive data types (like integers or doubles) as GPU parameters. Instead, pointers must be used. This is further complicated by Java's use of references instead of pointers.

Leaf and Multi-Leaf Parallelism

Leaf and multi-leaf parallelism are two different ways of taking advantage of parallelism. Leaf parallelism chooses one leaf on which to do (number of blocks)×(number of threads) number of playouts. Multi-leaf parallelism looks at a single level in the tree and chooses the same number of leaves as there are blocks, and does the same number of playouts of them as there are threads.



Leaf Multi-Leaf

Results

In Othello, white has an advantage. In order to balance the game, we added a komi value of 3 to black (komi are points added to black's score at the end of the game).

Figure 1 shows the number of playouts a CUDA player does versus the number of playouts a serial player does in 0.5 seconds. Figure 2 shows the average number of wins out of 100,000 games between a CUDA player and a serial player, with 0.01 seconds per move. For both data sets, we have averaged the games where CUDA plays as white as well as black.

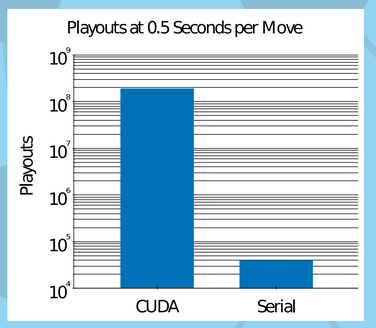


Figure 1

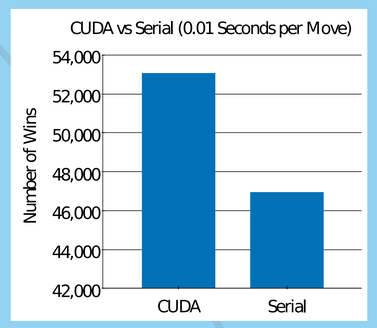


Figure 2

Conclusions and Future Work

Parallelizing Othello with JCuda results in an over 4000 fold increase in playouts. It also strengthens the player, though not to the degree we were anticipating. The parallel player is clearly stronger than the serial player, however, given more time, the parallel player does not seem to increase in strength. We believe this is because random playouts do not evaluate a game state for Othello as accurately as they do for Go, for which MCTS is used widely. We expect that parallelizing Go with JCuda will yield a greater increase in strength for our player.

In our current implementation the CPU cannot continue until the GPU has finished its playouts. In the future, we hope to make our parallel processes asynchronous, so that the CPU can further expand our search tree while the GPU is exploring playouts. This will further increase the depth of our search tree. Furthermore, we hope to implement this type of parallelism for Go, in order to strengthen the player.

Acknowledgments

- Peter Drake
- Jens Mache
- The Orego Team
- The Willamette Valley REU-RET Consortium for Mathematics Research (funded by the National Science Foundation)
- The John S. Rogers Science Research Program (Lewis & Clark College)

Bibliography

[1] C. Mitchell, et al., "Learning CUDA: Lab Exercises and Experiences, part 2", ACM SPLASH/ OOPSLA (poster), 2011

[2] CUDA, "http://www.nvidia.com/object/cuda_home_new.html"

[3] D. Luebke, et al., "Introduction to Parallel Programming", Udacity, "https://www.udacity.com/course/cs344", 2013

[4] JCuda, "http://www.jcuda.org"

[5] K. M. Rocki, "Large Scale Monte Carlo Tree Search on GPU," Ph.D. dissertation, Dept. Comp. Sci., University of Tokyo, Japan, 2011