Final Project report

Bo Gao(bgao), Di Wang(diw2)

Summary

We implemented a parallel Go artificial intelligence (AI) program in CUDA on the GPU. Our implementation achieved perfect accuracy when used to solve life-and-death problems (tsumego) on the corners and edges. The interactive AI could win games against amateur players without dan grading, with a considerably fast speed. It also sometimes provided elegant moves that could be great examples for beginners.

Background

Go is an abstract strategy zero-sum board game for two players, in which the aim is to surround more territory than the opponent. Our interactive application is an AI creating a computer program that plays the game with users. Because of its complexity and different patterns involved, the minimax search method used in chess alone may not work, as there can be in total $3^{19\times19} = 3^{361}$ different board compositions. Therefore a Go AI with the same sophistication level as a professional champion had always been an important research of study, as it was said to be able to model human thinking [1].

However, with the advent of AlphaGo in March 2016, and AlphaGo Zero in May 2017, more and more AIs become able to beat professional Go players. All of them use deep learning to train the neural network, both from human and computer playing [2]. This includes both pattern matching on partial boards, as well as pan-board calculations.

Because we lacked the knowledge of complex pattern matching, we decided to use only the idea of mini-max and Alpha Beta pruning, and to try to incorporate parallelism using CUDA. However, without ways to bring down the total number of moves to search for using the aforementioned pattern matching idea, we would try to come up with some heuristic that would allow us to judge the board state. In other words, we would examine the board state and give a score for Black. If it is Black's move it would want to maximize the minimum of all future scores after the next

White's move, and vice versa for White. Hence the minimax algorithm. In addition, our algorithm would also include Alpha-beta pruning which is an adversarial search algorithm used commonly for machine playing of two-player games. It is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. We would use this to bring down the total number of checks.

Various research has been done on the heuristic for board/move evaluation [3]. However, even for a rough approximation, considerable accuracy is only achieved by training in a convolutional neural network. The more layers, the higher the accuracy, but even so the maximum accuracy achieved is only 55.2% [3], which leaves room for improvement, but this is not an important topic in this project. We would adopt a similar heuristic, yet simpler.

Approach

Our project included a User Interface (UI) in Java, and everything else in C++. We first spend time on building the Java interactive UI, which is a completely distinct program from the AI implemented using C++ and specifically CUDA. The interactive UI lets the player play Black, and then responds with a White move. The UI and underlying decision implementation are linked by Java Native Interface (JNI).

Then we began to implement the life-and-death problem solver. This is also a good starting point for the full game AI. Without loss of generality we made all the problems starting by White, with an attempt to kill the Black piece. Our algorithm would be the combination of minimax and Alpha-Beta discussed in **Background**. Here is an illustration of Alpha-Beta pruning:

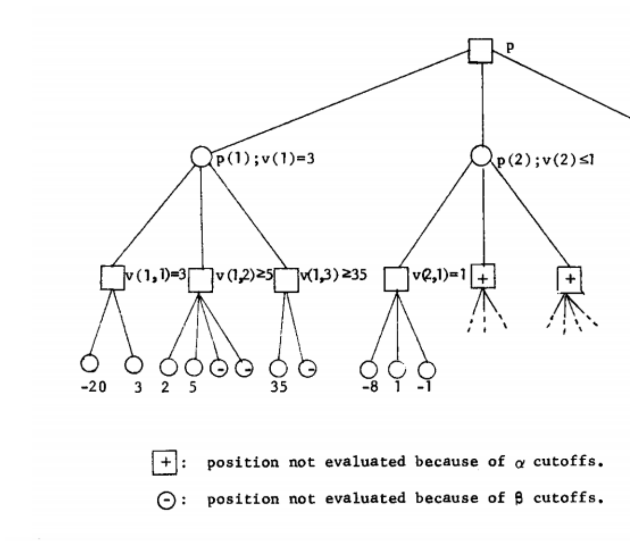
P
p(1);v(1)=3
p(2);v(2)≤1
v(1,1)=3
v(1,2)≥5
v(1,3)≥35
v(2,1)=1
-20 3 2 5 35 -8 1 -1
+ : position not evaluated because of α cutoffs.
⊖ : position not evaluated because of β cutoffs.

Figure 1: Alpha Beta Pruning [4]

Our algorithm for the life-and-death problems is a little different, as the end state would result in either the Black piece DEAD or ALIVE. Also, it is possible to infer from the exact shape of the Black piece whether it is alive or dead, even before all the possibilities are exhausted. This is essentially the same as pruning, as it eliminates possibilities that are about to result in Black alive from the perspective of White, or Black dead for Black.

Therefore our sequential algorithm would exhaust all the possible end states using Alpha-Beta pruning. The parallel algorithm, on the other hand, would parallel on the combined state of first n moves, where n is to be varied. Each life-and-death problem has a fixed finite number of crosses to be searched for, which we call m. For example, the following problem has $m = 5$.

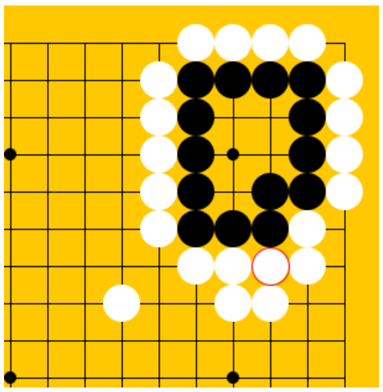

Figure 2: Example life and death problem with $m = 5$

For this problem, if $n = 2$, then we are parallelizing on the first White move and Black move. Therefore the total number of threads is 25 in this case. Each CUDA thread, in the kernel, would get a copy of the board state with the first two moves placed, and then use the aforementioned Alpha-Beta pruning algorithm to decide if this path has all subsequent paths resulting in DEAD or at least one resulting in ALIVE. Then the sequential algorithm on the host would use this to decide if White can kill the Black, and if so, under which move(s).

The results for the solver are accurate on $m \leq 8$ with considerable speed. This will be analyzed more in **Result**.

With this life-and-death problem solver handy, we were able to implement the full game AI. As introduced in **Background**, the number of possible Go board states greatly exceeds that of other board games, making it impossible to do a brute force search on all the possibilities. The approach we took was, again, parallelize on most of all possible board states after n moves, where n can be varied. We were able to use some Go intuition to eliminate many of the blatantly bad cases, for example when jumping far from the last opponent's move, or placing in a dangerous cross soon to be captured (examples below).

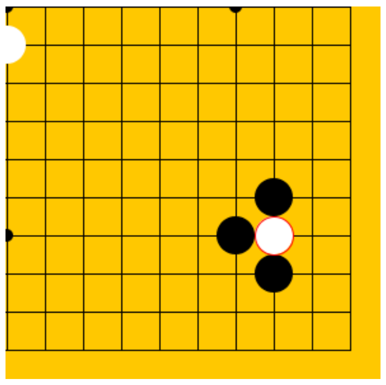

Figure 3: Example bad move by White, as Black can capture it within one move.

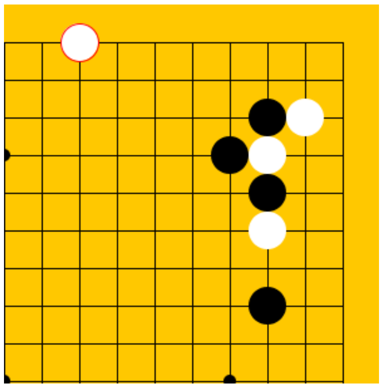

Figure 4: Example bad move by White, as it wanders off the current battlefield.

The moves like above would not be included in our CUDA search. In other words, we would only look at moves that are closer to the last opponent's move, and that make some Go-intuitive sense. Without this clever search, we would have to say with $n = 2$, with 361^2 threads. With the precheck implemented, we are able to make n as great as 6.

The rest of the parallel algorithm works similarly to the one used in the parallel life-and-death problem solver. However, since we do not reach the end game at the end of the nth step, instead of deciding DEAD or ALIVE, we would use a heuristic similar to the one referred to in **Background** to determine the current board state. Our heuristic works like a model of magnetic field of different magnets: each stone can be seen as a magnetic, that exerts force on points close to it. The closer the point, the greater the influence. Black influence and White influence cancel each other. Also, the edges and corners of the board act as "mirrors" so the magnetic field can bounce back. This is to represent the fact that stones on corners and edges have better control over the territory, than stones in the middle.

In summary, there are not much dependencies in the program except for the original, current board state to search from. Therefore, parallelism would be very amenable and applicable for the searches. Our model is similar to data parallel, as each part of the input array would be mapped to an output array without communication. It should also have a good temporal and spatial locality.

Trials and Iterations

Our design and approach described above went through a series of trials and revises. Given the fact that Go is a board game where new stones were being added, and previous multiple stones might be captured at once, the checking algorithm needed careful recursion which is sometimes not able to be parallelized. Here are the problems that we encountered which changed our approach:

1. CUDA not fit for OOP: We first designed the game semantics in an object-oriented programing (OOP) style. However, we soon realized that in CUDA, there is no way for device functions to call host functions. We first tried to get around that by converting everything to device, but this is not only confusing, hard to implement, but also slow in practice. Therefore we finally changed the idea and made it more adaptive by C++.

2. Host pointers not accessible to device: Similar to above, the device cannot access host struct pointers. Therefore we modified the structure again so that we were passing structs directly instead of their pointers.

3. Limitation on size of stack in CUDA: it turned out that our kernel implementation, which would call many device functions interchangeably, would make the stack exceed the default size of 1024. Therefore we had to increase that as we went along.

4. Limitation on the number of device functions called: Our original approach for life-and-death problems had a host function invoke the kernel many times, but the results were mostly inaccurate. It turned out that in our implementation the kernel could only be called 5 times, and would silently die after that. Therefore we modified the division of work so that the host function would initialize the array with indicator bits telling which task each thread should perform. This is similar to the style in Assignment 2 Rendering.

To sum up, the kernel should only be used to calculate something or do some uniform operation, but not as a recursive routine. In order to avoid the above problems, we must delegate some tasks to the sequential part of the implementation.

Result

References

[1] M. Temming. The newest AlphaGo mastered the game with no human input. *ScienceNews* , 192(8):13, 2017.

[2] D. Silver. Mastering the game of Go with deep neural networks and tree search. *Nature* , 529:484-489, 2016.

[3] C. Maddison. Move evaluation in Go using deep convolutional neural networks. *ICLR* , arXiv:1412.6564v2, 2015.

[4] S. Fuller. Analysis of the alpha-beta pruning algorithm. *Research Showcase @ CMU*, 1973.