



Parallel Go Solver

BO GAO, DI WANG

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.

Approach

- User Interface in Java
- Life-and-Death Problem Solver in C++ using Alpha-beta Pruning
- Heuristics Function for current board state
- Parallel Searching L&D Problem in CUDA
- Monte-Carlo Tree Search of AI in C++
- Parallel MCTS AI in CUDA

```
#####
#. o . o o . . . #
#x . x x o . . . #
#. . . x o . . . #
#x x x o o . . . #
#o o o . . . . . #
#. . . . . . . . #
#. . . . . . . . #
#. . . . . . . . #
#. . . . . . . . #
#####
blacks can be killed!
```

```
#####
#. o . o o . . . #
#x x x x o . . . #
#. . . x o . . . #
#x x x o o . . . #
#o o o . . . . . #
#. . . . . . . . #
#. . . . . . . . #
#. . . . . . . . #
#. . . . . . . . #
#####
blacks can't be killed
```

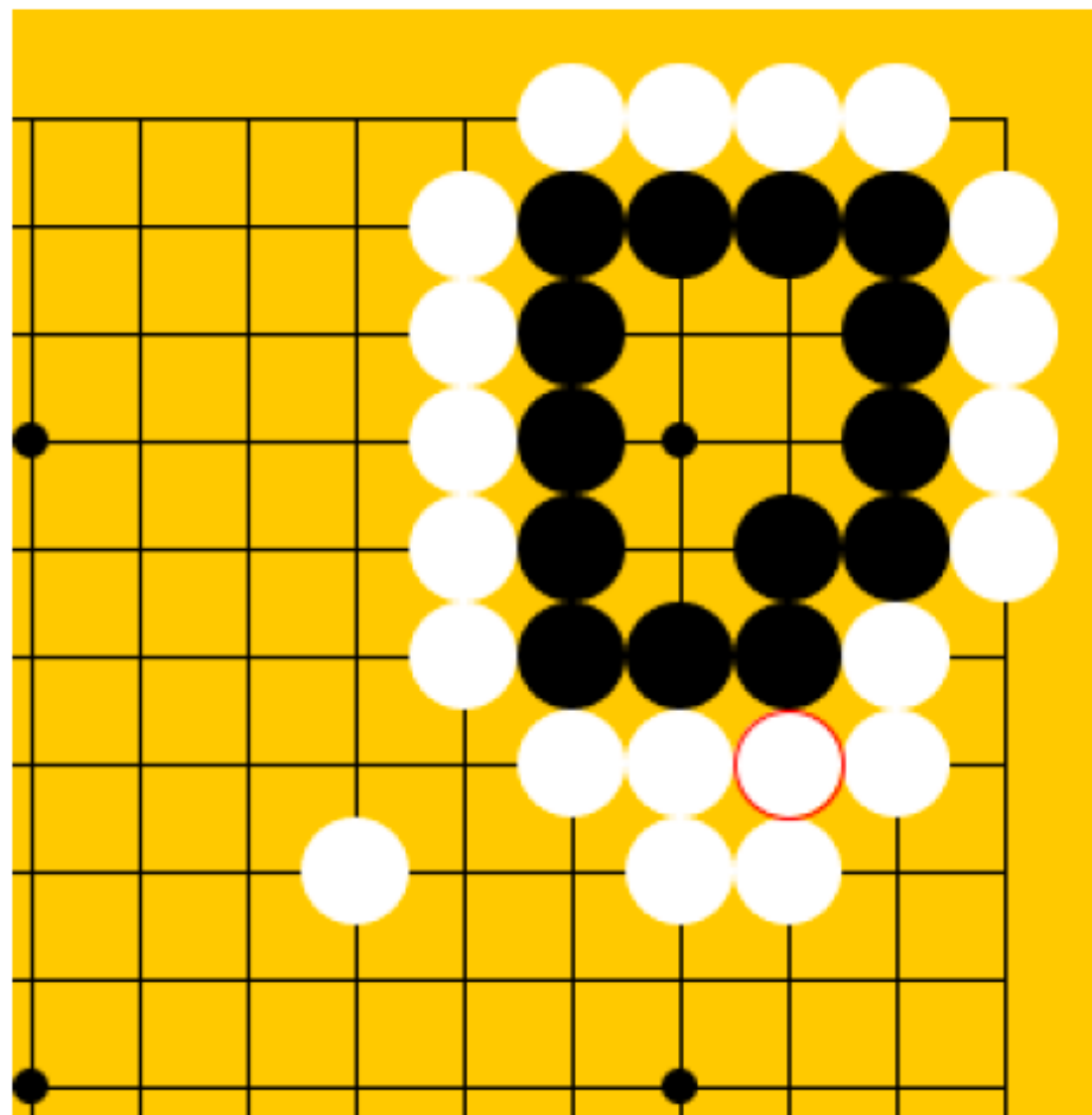
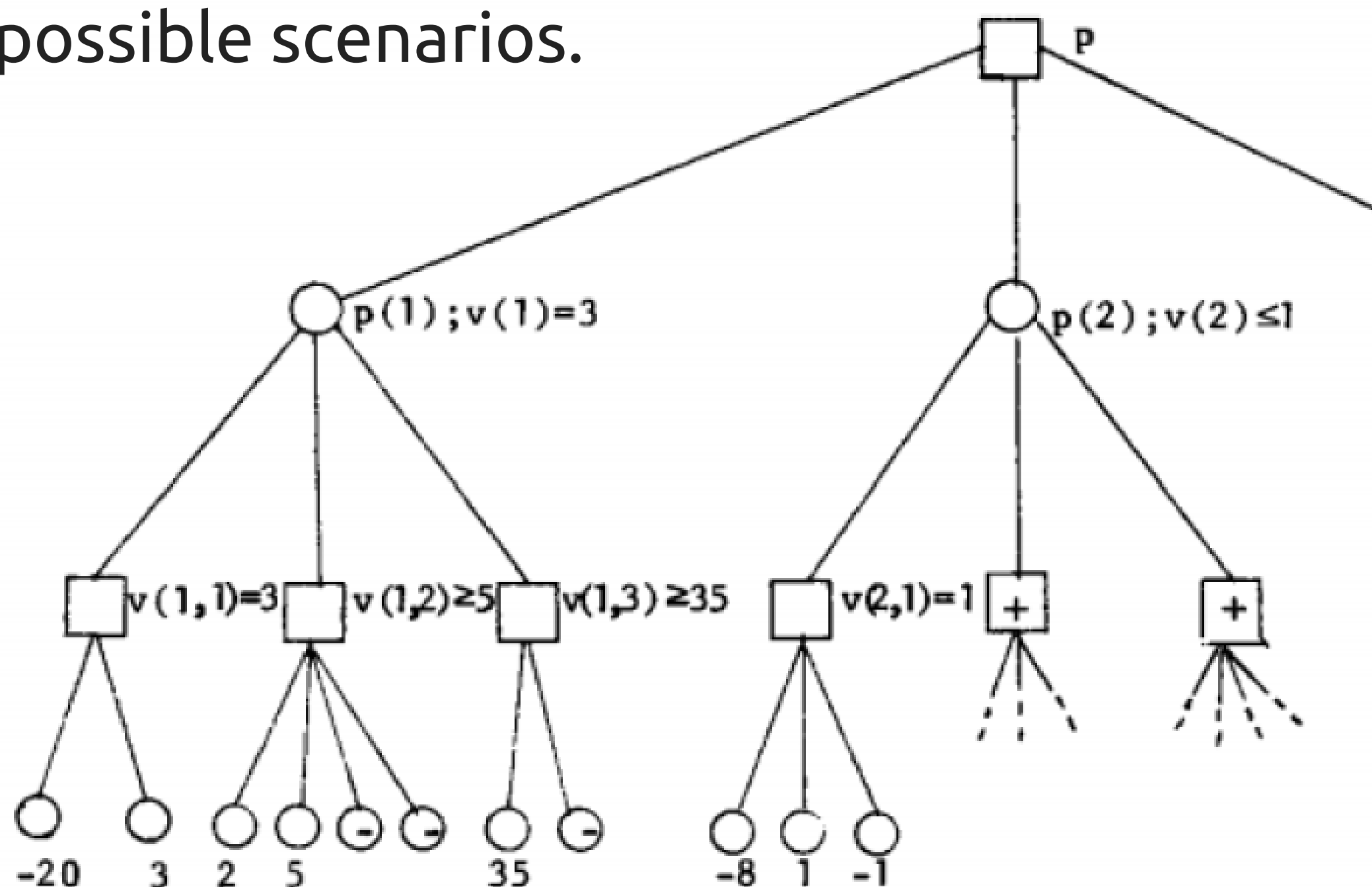


Figure 2: Example life and death problem with $m = 5$

The algorithm to solve life-and-death problem is applying Alpha-beta pruning, stopping at the states where the black stones can be identified as already alive or already been killed. We will search all possible moving sequences within a range of m positions, and check all possible scenarios.

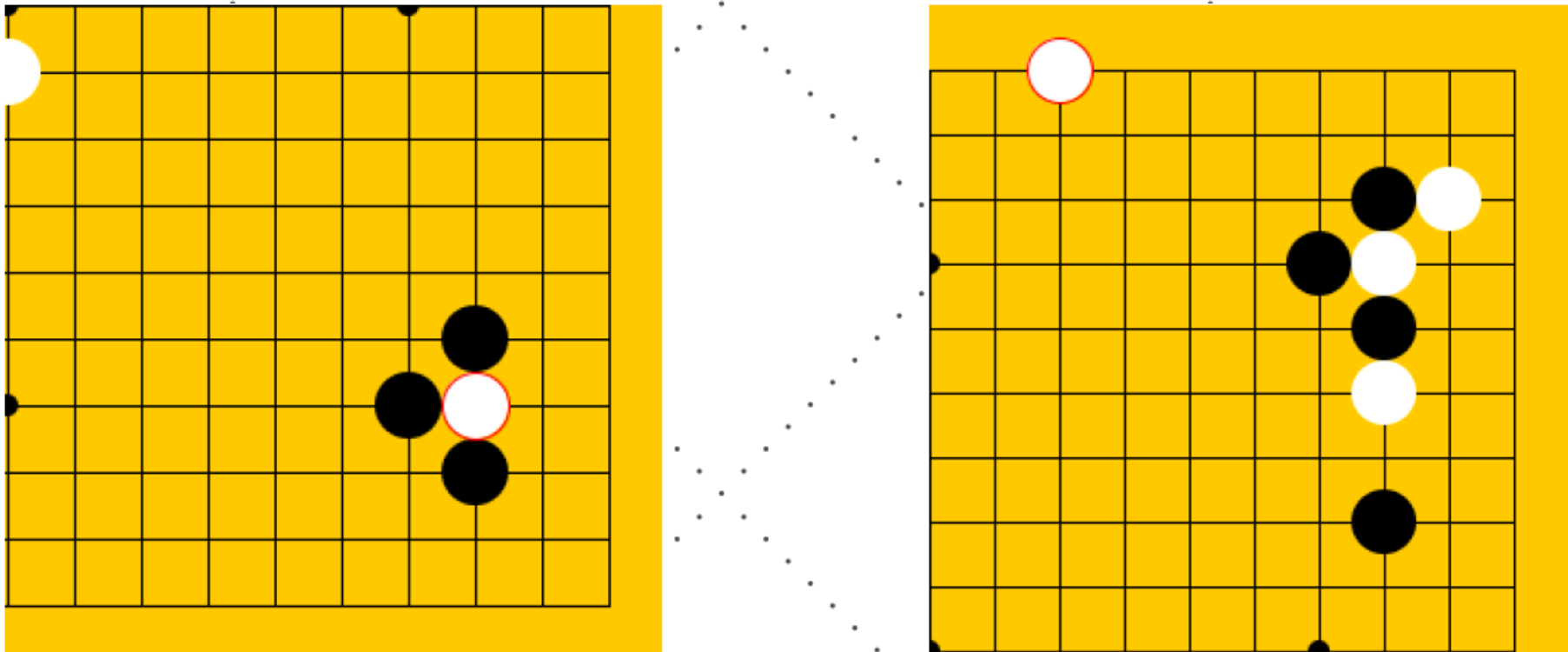


$\boxed{+}$: position not evaluated because of α cutoffs.
 \ominus : position not evaluated because of β cutoffs.

Figure 1: Alpha Beta Pruning [4]

Monte-Carlo Tree Search Method:

We would try to come up with some heuristic that would allow us to judge the board state. We examine the board state and give a score for Black. If it is Black's move it would want to maximize the average score of all future situations after the next few random moves, and vice versa for White.



Bad moves that can be avoided by the heuristics
(the highlighted white stone is the move)

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.

Limitations of CUDA:

- **CUDA not fit for OOP**
- **Host pointers not accessible to device**
- **Limitation of size of stack in CUDA**
- **Limitation on the number of device functions called**
- **Limitation on the GPU memory**

--> **The best approach is to parallel the Territory Evaluation functions, not recursive calls of adding_stones !**

Result

We analyze:

- the running time performance and speedup in parallel life-and-death problem solve
- the running time performance and speedup in parallel full game AI with different board size and different searching depth
- the sophistication of the actual moves returned by the game AI

Table 1: Full game AI serialized vs. parallel

Steps	Time of serialized algorithm on CPU in milliseconds	Time of parallel algorithm on GPU in milliseconds	Time of Cuda Kernel in milliseconds	Speedup in terms of running time	Speedup in terms of kernel running time
1	14.1435	3.7785	0.0227	3.74315	623.061674
2	18.8677	4.0783	0.0190	4.62636	993.036842
3	23.9899	3.8899	0.0183	6.16723	1310.9235
4	26.4919	3.8477	0.0180	6.88512	1471.7722
5	32.0519	3.8028	0.0287	8.42850	1116.79094

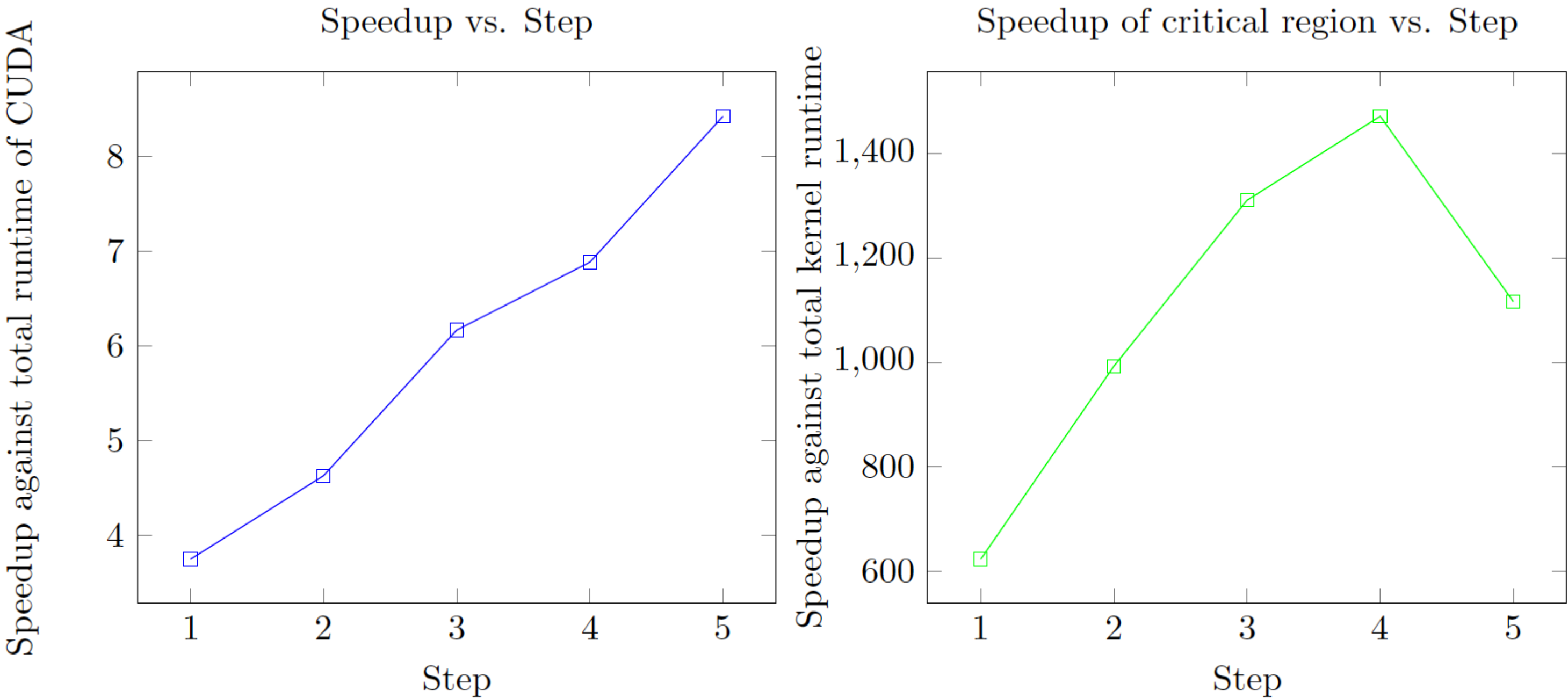


Table 2: Full game AI serialized vs. parallel cont.

Step	Serialized runtime (ms)			Parallel runtime (ms)			Speedup		
	b = 9	b = 9	b = 19	b = 9	b = 9	b = 19	b = 9	b = 9	b = 19
	n = 1	n = 2	n = 2	n = 1	n = 2	n = 2	n = 1	n = 2	n = 2
1	0.8601	54.9245	68.5128	2.1970	26.4787	23.2450	0.3914	2.074	2.947
2	1.0967	76.8910	95.6443	2.1112	27.5114	23.3673	0.5195	2.795	4.093
3	1.5444	97.6486	119.3768	2.2408	27.7287	23.3265	0.6892	3.522	5.118
4	2.0792	123.890	147.3680	2.1727	27.6843	23.7210	0.9569	4.475	6.212
5	2.3825	143.503	178.5760	2.2378	27.8785	23.2771	1.0646	5.147	7.671

- the sequential algorithm actually increases in runtime is because it needs the previously placed stones to calculate the potential board state of each proceeding path. (not easily avoiable)
- The main difference between the columns Time of Cuda Kernel vs. Time of parallel algorithm is because we have allocated enough memory and initialized the states for each possible path beforehand.
- The heuristic function is the only thing in the kernel, while the actual kernel computation was kept to a minimum
- The trend on larger b and n , the speedup increases. This is because if the problem size increases, the benefit of parallelism overweighs the overhead.

Table 3: Speedup of life-and-death problem solver

Iteration	Average time of parallel algorithm (ms)			Average time of serialized algorithm (ms)			Average Speedup
	m = 5	m = 4	m = 3	m = 5	m = 4	m = 3	m = 3,4,5
1	2933.4	770.04	81.87	11.59	2.06	1.30	0.00395
2	2700.7	789.72	72.48	11.37	2.02	1.30	0.00258
3	2866.9	807.96	83.40	11.38	2.04	1.24	0.01619
Avg	2833.7	789.24	79.05	11.45	2.04	1.28	0.00757

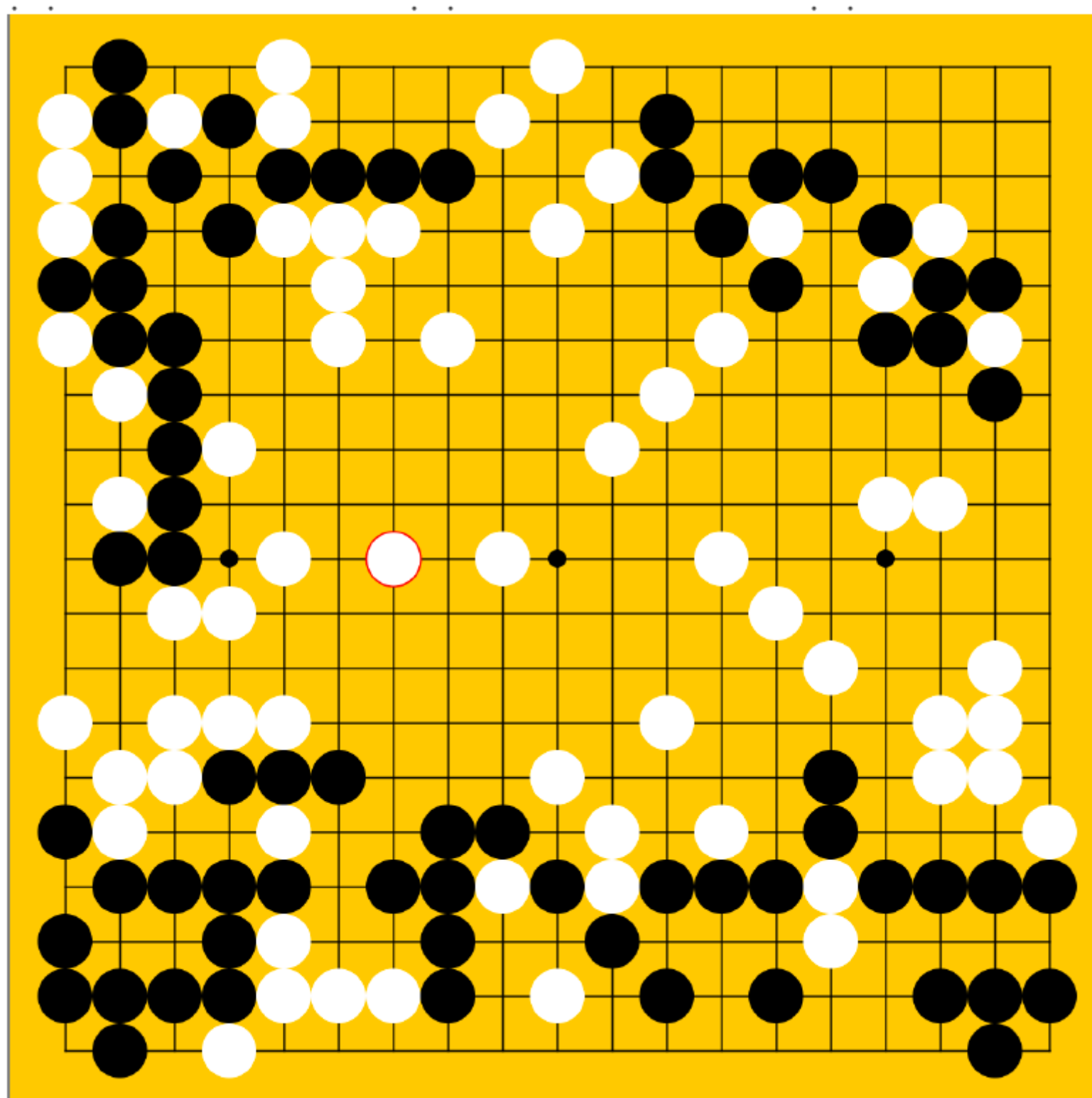
- The parallel algorithm is much worse than the sequential algorithm. We conjecture that the overhead of the CUDA threads greatly outweighs any improvement by parallelism.

- Increasing m would definitely improve the speedup, which is also a trend we see from $m = 3$ to $m = 5$. However, after $m = 7$ the running time increases drastically, because the number of recursions that we are doing is increasing exponentially.

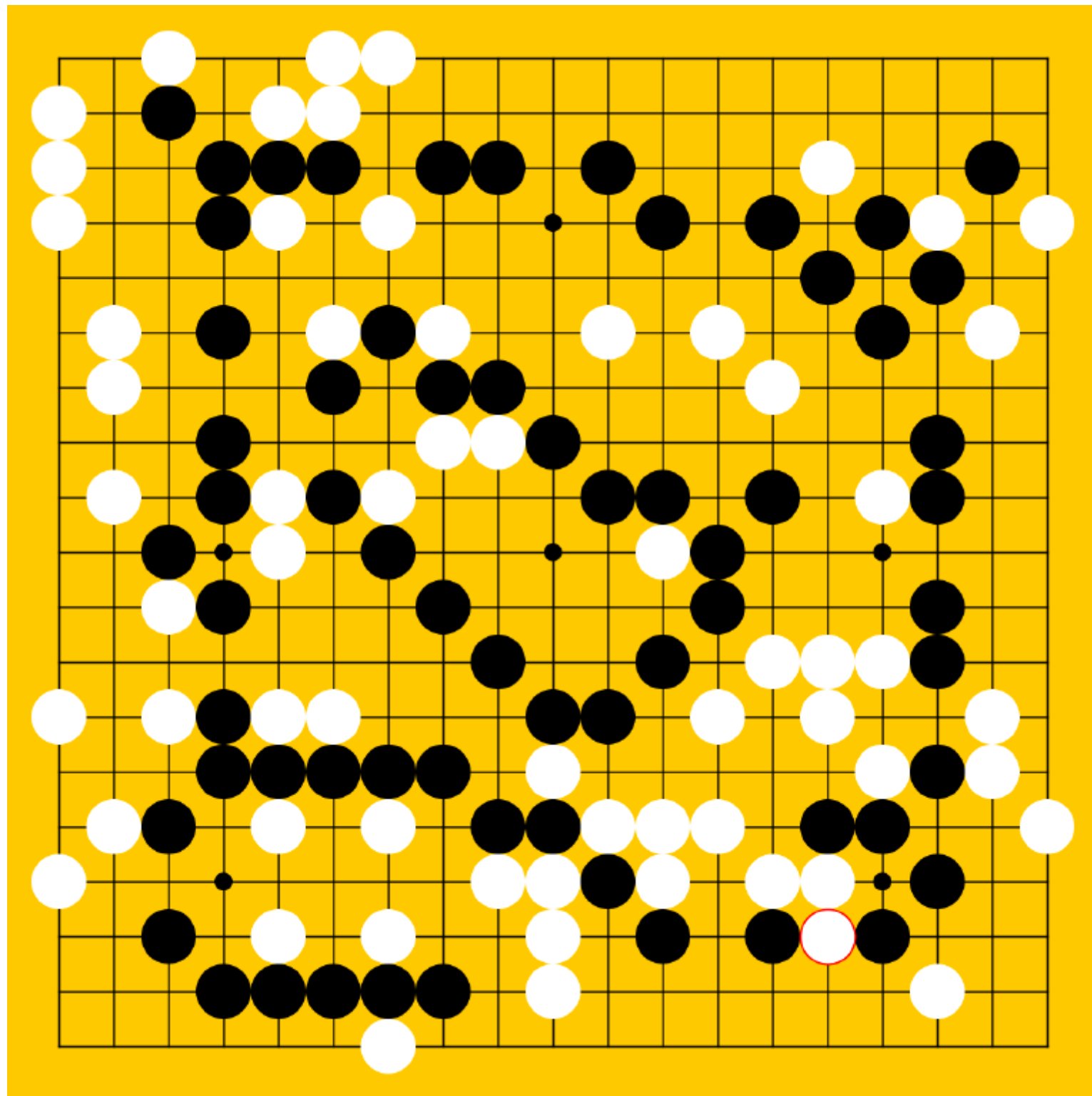
To sum up, the reason for not high speedup is a lack of parallelism. Our algorithm is merely parallelizing on several possible moves. The load balance for our program is poor. Therefore it would be definitely helpful to assign the work dynamically. Also, our programs have a lot of serialized work such as `add_Stone` and `check_terr`. Maybe come up with some other data structure that yields a better tradeoff between communication and speedup.

Performance analysis: The parallel algorithm usually places moves that make much intuitive sense, which makes it appear like a human player.

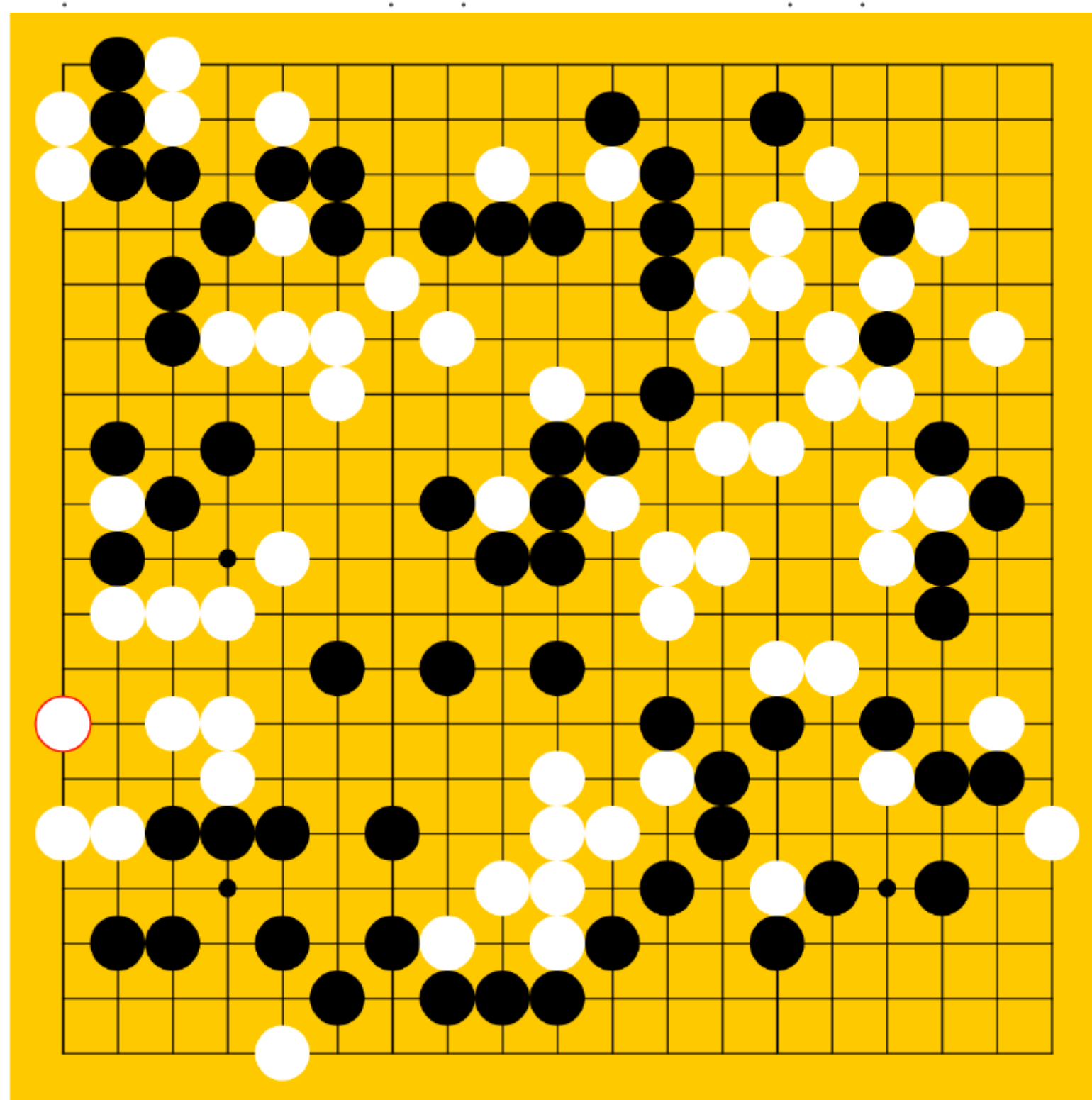
Example game 1



Example game 2




Example game 3



The last white move(highlighted stone) is relatively legit since it either marks the contour of the territory, cuts the Black pieces to capture them, or secures two eyes of the white piece to make it alive.

In summary, even though the AI is unable to make precise predictions in battles that require clear and sophisticated judgment, sometimes it gets the locally optimal solution. Also, it is often able to make good moves that direct the overall of the game, which is an important ability often lacking in amateur players. These qualities allow the AI to be a good teacher and practice player for amateur players.



Potential Improvement & Further Research

- Maybe try other platforms other than CUDA, which will avoid us from some limitations discussed before
- Maybe apply the same idea of "moving-sequences" to life-and-death problem solve, in order to avoid recursive calls inside kernel functions
- Maybe try to use neural networks instead of "naive" heuristics to improve the AI judgment using enough datasets

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.