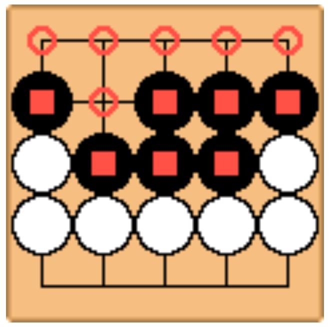
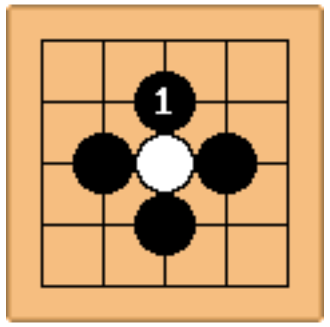
Monte Carlo Go on NVIDIA GPU

Zachary Clifford

Overview of Go

- Go is a game about capturing territory
- Final score under Chinese scoring is number of stones on board plus number of spaces surrounded. Black has 13 and White has 12.
- Groups of stones are removed if they have no free spaces adjacent to them. Here black playing at (1) has captured white. Capturing larger groups is also possible





Algorithm Overview

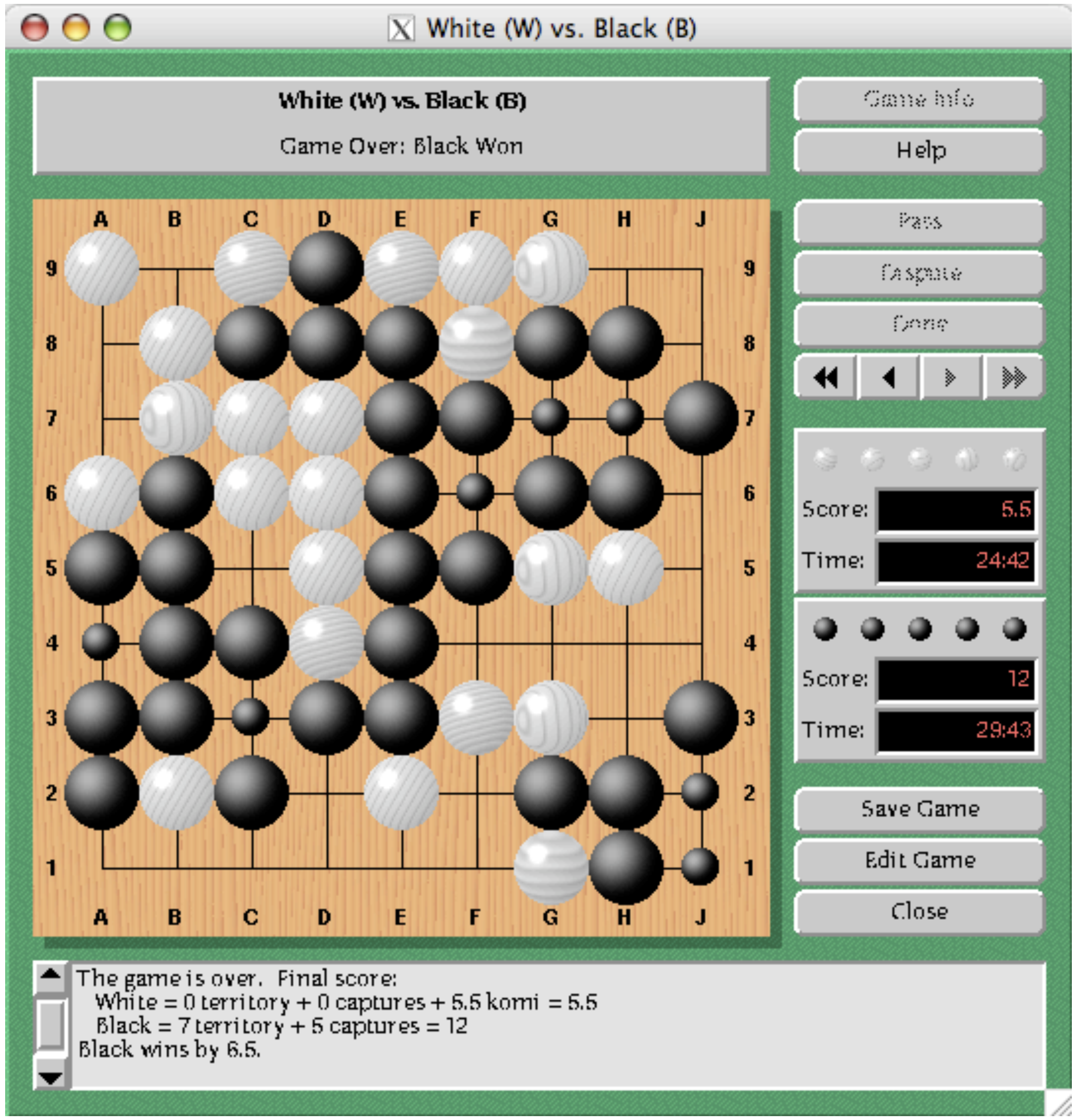
- Monte Carlo methods assume that a random sampling of a space are representative of that space.
- In this case, consider each possible move and randomly sample games resulting from that move. Play out games to completion with both players playing a very simple random strategy.
- Hope that a move that leads to higher scoring games is a strong move.
- On a 9x9 Go board, there are a number of possibilities roughly equal to the number of remaining open spaces factorial.
- In practice, this means that only a really small subset of possible games can be sampled.

Implementation

- This implementation uses NVIDIA CUDA to parallelize the playing of random games.
- The GPU kernel takes as input a board state, active player, and chosen move.
- It then plays out 128 games to completion in parallel. In addition, each position on the 9x9 grid is evaluated in parallel within each game.
- It proceeds through different steps identifying groups, identifying free spaces around groups, marking valid moves, making a move, and finally killing dead stones. This proceeds on each of the 128 games until no player has valid moves.
- Finally a prefix sum kernel computes the average output score of that move.
- This is done for each possibility, and the highest scoring move is chosen.

Performance

- This engine was integrated with GNU Go, an open source Go engine.
- This modified GNU Go was played against a stock GNU Go player to test for speed and strength.
- In this example game the CUDA player, White, lost. This is representative of its strength compared to GNU Go.
- GNU Go never takes more than a second or two to decide on a move.
- CUDA Go can evaluate a possible move in 1/4 second, but considering an entire board can take 20 seconds because it has no strategy for discarding obviously bad moves.



```
The game is over.  Final score:
  White = 0 territory + 0 captures + 5.5 komi = 5.5
  Black = 7 territory + 5 captures = 12
Black wins by 6.5.
```

Evaluation

- Strength was low because of limited sample size. To gather a representative sample many more games would need to be played. The delay in performing this computation became prohibitive.
- Other Monte Carlo Go codes use a directed search. They use Monte Carlo to decide between moves that another algorithm already declared as a strong move. Implementing that in parallel was beyond the scope of this project.
- Truly random play is easiest to implement but not representative of actual good play. For example, random players may happen to create defended groups, but they will probably be unable to organize an assault on a pre-existing group. This biases the algorithm towards a "turtle" strategy.
- This problem does not map well to the CUDA architecture. CUDA excels in problems with high arithmetic intensity and relatively little memory movement. As a SIMD architecture it handles conditional branching and memory movement exceptionally badly. It is forced to serialize operations that take different execution paths.
- Although each play out is embarrassingly parallel, the locations on a board are tightly coupled. Some positions have stones while others do not. Positions can freely move between occupied and not occupied. This severely limits parallelism within a game, but CUDA requires such granular parallelism for good performance.

Future Work

- This algorithm runs more slowly on a GPU than an equivalent algorithm on a CPU. The GPU is inherently slower at memory movement and logical operations than the CPU, but for other problems it leverages massive parallelism.
- The GPU could be used asynchronously in addition to the CPU to offload some game processing, but the AI would have to remain local to the CPU.
- If the CPU AI routine had an operation to evaluate a board that involved little decision making and many FLOPs, the GPU could accelerate that.
- The GPU could also aid in evaluating a board position to assist a human player.