1 Summary

We implemented a real-time game theory optimal strategy calculator for heads up Texas Hold’em poker. The game theory optimal strategy, and in this case the Nash equilibrium, is a set of strategies for which neither player can change their strategy and do better than they were with the GTO strategy. Because our goal was to use this application in online poker, we needed to make it fast enough for real time use, or under 30 seconds for any reasonable input. We achieved this bound, and found that we could calculate ranges for the out of position player (OOP) of sizes up to 20 in under 30 seconds. We were able to see speedups of over 400x compared to a baseline run on a CPU and over 3,000x compared to a baseline run on a GPU.

2 Background

To better explain the project, I will go over the basics of Heads up Texas Hold’em. Heads up is a variant of poker with only two players, and Texas Hold’em is the standard 5 card poker game. There are three rounds of cards being shown, the flop, the turn, and the river. We will only deal with post river computation (all cards on the board are shown). In heads up there is an out of position player (the player to act first) and the in position player (the player to act second). We will denote these with OOP and IP. Each player has a range of hands which they could have, this could be anywhere from 1 to 1326 (52 choose 2). The goal of a poker player is to narrow down your opponents range in order to best play against what they could have. Given the two players ranges, a pot size, and a bet size, this is exactly what our program does.

Once we have both players ranges, we have to find the optimal strategy for both players. To do this, we enumerate all such strategies. To make this computation feasible, we limit the OOP to the moves check-call (check but call anything the IP bets), check-fold (check but fold anything the IP bets), and bet. We limit the IP player to check, bet, call, and fold. This is 3 moves for the OOP player, and 4 moves for the IP player. With just 3 moves available, the number of strategies for the OOP player becomes $3^r$, where $r$ is the size of their range. Any of these strategies could be optimal, so we need to check them all to find the best. We do this in the following way: Given a strategy by the OOP player, calculate the move for every card in the IP player’s range that maximizes the IP players value. We then find the minimum value of all such strategy for the IP player, and this is our OOP player’s optimal strategy.

To store all of the relevant information to compute this we only need $O(r)$ bytes of memory, which is a very small amount. With $3^r$ things to compute, but only $O(r)$ memory required, our program will be very obviously compute bound. Furthermore, each of the strategies is independent of the rest, so we can compute the value of each strategy in parallel.

3 Approach

Because our workload is so heavily compute bound, we immediately decided to use a GPU for the computation. In order to maximize efficiency we needed to find a way to keep the GPU busy for the most amount of time. Our first implementation had the CPU calculate the strategies for the GPU to compute the value of, but sending these strategies back and forth was added too much memory IO and caused our application to become overly memory bound. This resulted in a very slow application that did not utilize the GPU effectively.

To combat this we moved all computation to the GPU because generating the next strategy given a strategy was extremely cheap. Using this idea we computed a set of initial strategies for the GPU to use, and from there the GPU was responsible for everything. This turned out to be a very efficient way of splitting up the work because we were able to get full utilization of the GPU for the vast majority of the computation.

To get into the specifics of our implementation, we used cuda to execute our code on the gpu, and had c code on the CPU. We had each cuda block execute a range of strategies (starting with the one given to it by the CPU), and each cuda thread responsible for a hand in the IP players range. This allowed for all of the possible strategies to be computed completely
in parallel by all of the cuda blocks.

We ran our program on the ghc machines. Each of them had a Nvidia GeForce GTX 1080 graphics card. These graphics cards have 20 SM cores, 128 cuda cores per SM, which results in 2560 cuda cores. In the cuda programs we have written before we had many cuda threads for each cuda block, but in our program we have a very small number of threads per block (¡30). To still utilize as much of the GPU as possible, we ran experiments to see the optimal block size to result in the maximal speedup for the same input. The results can be seen in the graph below. We ran the program with varying cuda blocks, and fixed the input size to a range of size 17. Because of such a low thread count per block, we needed many cuda blocks to fully utilize the GPU. Experimentally we found this number to be 448 cuda blocks.

![Time with variable Cuda blocks, fixed input](image)

4 Results

After sufficient tuning, we were pleased with our results and efficiency. We were able to compute range sizes of 20 in around 40 seconds, which requires checking $3^{20}$ or roughly 3.5 billion possible strategies. In order to verify our solution was efficient, we had two different baselines. The first was a singly threaded program run on the CPU, and the second was one that was run on the GPU but only with one cuda block and thread. We decided to have the baseline on the GPU to see if we could achieve the full potential of the GPU (2560x speedup).

After running all three implementations on a variety of different input sizes (OOP player’s range size), we saw very good performance of our optimized code compared to both baselines. The 4 graphs below show our optimized version compared to the baselines. The first two graphs show the timing on various inputs for all three implementations. The second graph uses a logarithmic scale for the y axis. The 3rd and 4th graphs show the speedup our optimized solution sees compared to the two baselines.
As seen by the graphs our implementation vastly outperforms both of the baseline implementations. In graph 2, you can see that our implementation does not need its full resources until an input size of 15, at which point subsequent inputs scale computational complexity by 3 (and therefore our performance). You can see the under utilization of resources in the speedup graphs. Our speedup against the CPU baseline increases steadily until we reach size of 16 or 17, at which point it levels off. Again this shows that until we have a high enough computational complexity our program will not be able to fully utilize the CPU. One final interesting feature of the graphs is in the speedup graph of the singly threaded GPU baseline. Earlier we talked about on a 1080p GPU the max speedup was 2560x, however we experienced a maximum speedup of 3023x. We hypothesize that this increased speedup from expected is due to the latency hiding each block is able to do in our multi threaded implementation, along with the load of each of the machines being extremely variable during our time of testing.

After analyzing the timing of our code, we felt that our implementation achieved a very good measure of parallelism and speedup from it. If we were to try to further improve performance of our project on a single machine, we would have to look for more optimizations in our algorithm itself rather than trying to parallelize more. With this being said, the workload of this project would be very fitting for a cluster of nodes, because each could compute a segment of the strategies and aggregating the best strategy between them would be very cheap.

We felt that the machine we ran our code on was a good choice for our project because we needed a machine that could handle a huge amount of parallelism without needing much memory bandwidth. The GTX 1080 was the perfect fit for these needs, which is why we think the target machine was fit for the project.

5 References
15-418 lectures, CUDA

6 Work Division
Equal work was performed by both project members.

7 Github
https://github.com/dasteere/15418-Final-Project