ECSE 420, Fall 2020, McGill University

Project Write-up: Reversi AI

Group 24

Mai Zeng 260782174
Borui Tao 260676208
Hanwen Wang 260778557
Wenhao Geng 260660496

Problem Addressed

In this project, we implemented an AI for the game - Reversi. The AI has two levels, easy and medium. The game between different levels and a random player will be executed in the terminal, we can even play against each of them ourselves by changing players. All the available next moves, current board state and current score can be shown in the process of the game. For each of the two levels, we implemented both sequential and parallel algorithms. We use C as our programming language for sequential parts and CUDA C as an extension for parallel parts. For easy level AI, it will check every playable step and choose the one which can give it the best score. For medium level AI, it will look 3 steps ahead using the minimax method. Minimax is a backtracking algorithm that is used to find the optimal move for a player, assuming the opponent also plays optimally. In other words, the player is its own score's maximizer and the opponent is their own score's minimizer, the medium level AI will take into this consideration and choose to run the step which can give it the best score after 3 steps. The logic for sequential and parallel algorithms are quite similar, but the implementation for parallel algorithms is more complicated. For parallel easy level AI, each cell on the board has one thread, all threads run in parallel to first detect if their corresponding position is a playable cell. If it is, it will copy the current board state, play the simulated step in this cell, record the score as well as its coordinate x and y and save them into a shared array. After all threads complete running, the best score in the array is calculated and its corresponding cell becomes the next move on the real board. For parallel medium level AI, it repeats the steps in easy level AI three times using kernel calling kernel.

Code Structure

Game program

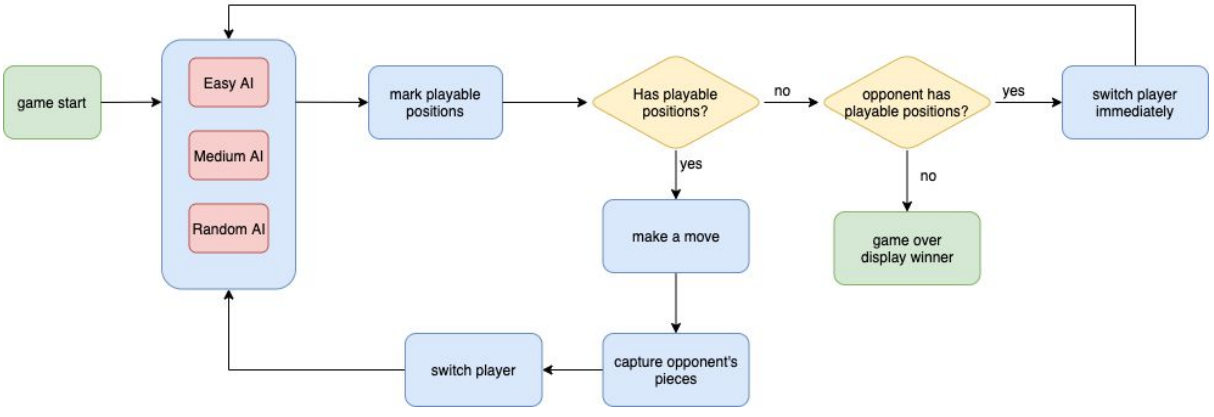

Figure 1. Reversi game design flowchart

Figure 1 illustrates the basic flow of the two-player Reversi game. The game board (Figure 2) is implemented as a 2-d array, with 'O' and 'X' indicating the tiles which two players have placed or captured. When the game is started on the command line, the game will start running with the specified player types (Human player, Random AI, Easy AI, Medium AI). Next, for the current player, the playable positions on the board are marked for the player to choose (as dots on the board). If there is no playable position for the current player, the

program switches to the opponent to check if there is any playable position. If both players have no playable positions, the game ends and the winner is printed on the command line.

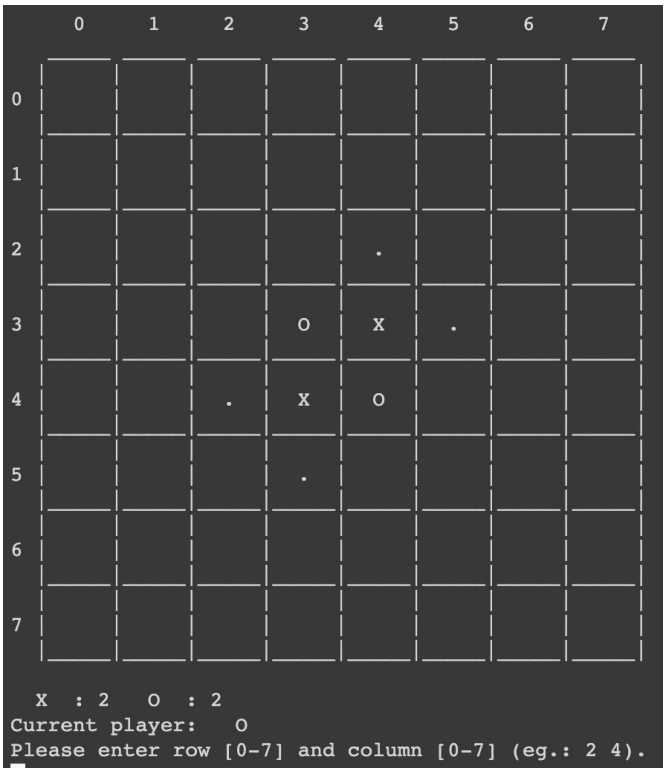

Figure 2. Reversi game board

For the human player, a move can be directly played by specifying the coordinates of the tile which he/she wishes to play on the command line. For the Random AI, the program would generate a move based on a random number generator. For the easy AI and medium AI, the program uses the Minimax logic to generate the best moves for the current player.

Easy AI

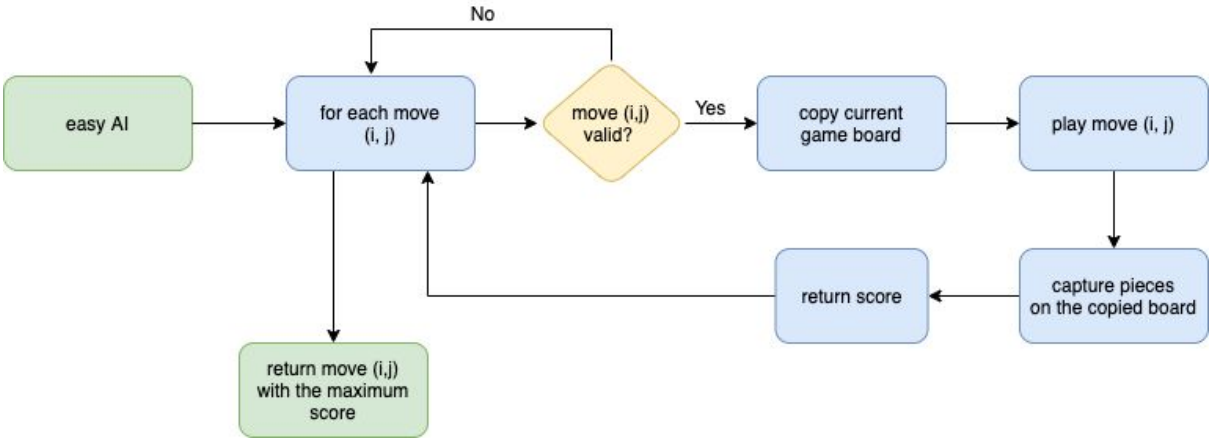

Figure 3. Easy AI logic flowchart

Sequential:
As we can see in the figure above, for sequential easy level AI, we first go through each cell on the board in sequence and check whether it is a playable position or not. If that move is

valid, we will copy the current board state and the 3D array recording 8 playable directions around that cell. We next simulate that move on the copied board and return a score we get after playing this move. After that, we save this cell position and its corresponding score into a structure named moves. Besides, the highest score value is continually updating, and the maximum score will be returned after we go through all cells on the board. The code snippet for this logic is shown in Figure 4.

```
// Think ahead 1 step
int get_easyAI_move_cpu(move* moves)
{
    int maxScore = 0;
    for (int i = 0; i < 8; ++i)
    {
        for (int j = 0; j < 8; ++j)
        {
            if (board[i][j] == PLAYABLE) {
                char copy_board[8][8];
                int copy_playable_direction[8][8][8];
                copy_board_mem_arr(copy_board, copy_playable_direction, board, playable_direction);
                int s = capture_potential_pieces_cpu(i, j, copy_board, current_player, copy_playable_direction);
                moves[i*8 + j].max = s;
                moves[i*8 + j].i = i;
                moves[i*8 + j].j = j;
                if (s >= maxScore) {
                    maxScore = s;
                }
            }
        }
    }
    return maxScore;
}
```

Figure 4. Code snippet for get sequential easy AI move

We will then go through the structure moves again to find the row and column of the move that give us the best score. We used the Reservoir Sampling idea to randomly choose a best move if there are multiple moves with max score returned. Afterwards, we play this move based on the row and column we get on the real board. The code snippet for this logic is shown in Figure 5. We can see that once the move is confirmed, the program will update the real game board, capture the pieces, and switch round to the opponent.

```
            if (ai == Easy) {
                max = get_easyAI_move_cpu(moves);
            }

            int sampling_count = 0;
            for (int i = 0; i < 64; i++)
            {
                if (moves[i].max == max)
                {
                    sampling_count++;
                    if (rand() % sampling_count == 0) {
                        row = moves[i].i;
                        column = moves[i].j;
                    }
                }
            }

    if (is_valid_position_cpu(row, column) && board[row][column] == PLAYABLE)
    {
        // printf("row: %d, column: %d is valid\n", row, column);
        board[row][column] = current_player;
        scores[current_player]++;
        capture_pieces_cpu(row, column);
        change_current_player_cpu();
    }
```

Figure 5. Code snippet for playing the sequential easy AI move

Parallel:

The logic of parallel is quite similar to that of sequential, the main difference is the way of implementation. Since the size of our board is 8x8 and there are 64 cells in total, we use only 64 threads and 1 block to do this parallel task. These 64 threads will perform the task mentioned in the flowchart above in parallel. In this global function, instead of using a 2d

array with 8 rows and 8 columns, the board is now a 1d array whose size is 64. We also use i = idx / 8 to get the current number of row, j = idx % 8 to get the current number of column and i * 8 + j to locate its corresponding position in the 1d array. Compared to using a 8x8 2d array to copy board and a 8x8x8 3d array to copy direction in sequential part, we only use 1d array with size 64 and 512 correspondingly to copy those two in parallel part. Consequently, all calculations related to 2d array should be converted to calculation in 1d array in the device function. Since the global function cannot return a result, we just save row, column and score into structure moves. The code snippet for this logic is shown in Figure 6.

```
__global__ void get_easyAI_move(char* board, int* playable_direction, int current_player, move* moves)
{
    int idx = threadIdx.x;
    if (idx < 64)
    {
        int i = idx / 8;
        int j = idx % 8;
        //printf("board, %d\n", board[i*8 + j]);
        if (board[i * 8 + j] == PLAYABLE) {
            char copy_board[64];
            int copy_playable_direction[512];
            memcpy(copy_board, board, sizeof(char) * 8 * 8);
            memcpy(copy_playable_direction, playable_direction, sizeof(int) * 8 * 8 * 8);
            int s = capture_potential_pieces(i, j, copy_board, current_player, copy_playable_direction);

            //printf("easy score: %d here, row: %d, column: %d\n", s, i, j);
            moves[idx].max = s;
            moves[idx].i = i;
            moves[idx].j = j;
        }
    }
}
```

Figure 6. Code snippet for get parallel easy AI move

In the function where we call kernel functions, we reshape the current board and its direction into 1d arrays. We then use cudaMalloc and cudaMemcpy to copy these two 1d arrays for further execution. The reason why we do not use cudaMemcpyDeviceToHost to copy those two arrays back is that we do not need them after the execution, the only thing we need is moves, which has necessary information about rows, columns and scores inside. Since this time the global function does not return the maximum value, we have to go through moves to find that value before we get its corresponding row and column. After all these steps are done, we play the move we just got on the real board, which should be the same as what we can get in the sequential part in the same situation. The code snippet for this logic is shown in Figure 7.

```
        devBoard = (char*)malloc(64 * sizeof(char));
        devplayDir = (int*)malloc(512 * sizeof(int));
        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 8; j++)
            {
                for (int k = 0; k < 8; k++)
                {
                    devplayDir[n] = playable_direction[i][j][k];
                    n++;
                }
                devBoard[m] = board[i][j];
                m++;
            }
        }
        char* devBoard2;
        int* devplayDir2;
        cudaMalloc(&devBoard2, 8 * 8 * sizeof(char));
        cudaMemcpy(devBoard2, devBoard, 8 * 8 * sizeof(char), cudaMemcpyHostToDevice);
        cudaMalloc(&devplayDir2, 8 * 8 * 8 * 8 * sizeof(int));
        cudaMemcpy(devplayDir2, devplayDir, 512 * sizeof(int), cudaMemcpyHostToDevice);
        cudaMallocManaged((void**)&moves, 64 * sizeof(move));

        if (ai == Easy_CUDA) {
            get_easyAI_move << < 1, 64 >> > (devBoard2, devplayDir2, current_player, moves);
        }
```

```
for (int i = 0; i < 64; i++)
{
    if (moves[i].max > max)
    {
        max = moves[i].max;
    }
}

int sampling_count = 0;
for (int i = 0; i < 64; i++)
{
    if (moves[i].max == max)
    {
        sampling_count++;
        if (rand() % sampling_count == 0) {
            row = moves[i].i;
            column = moves[i].j;
        }
    }
}
```

Figure 7. Code snippet for playing the parallel easy AI move

Medium AI

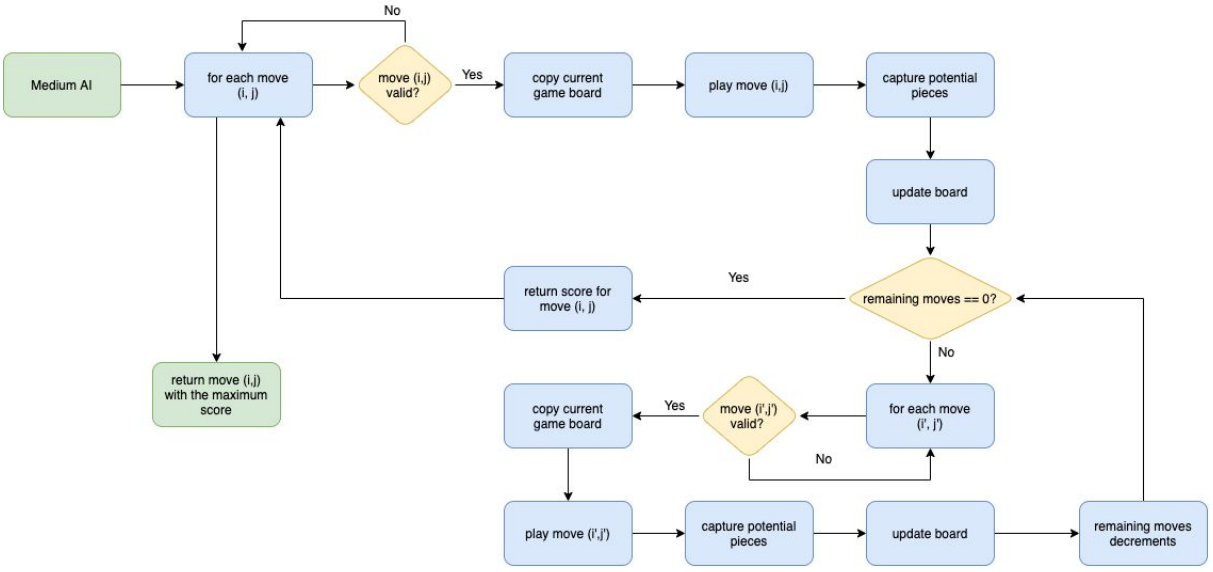

Figure 8. Medium AI logic flowchart

Sequential:

The code snippet below and flow chart above shows the process of the medium AI determining the next move. The sequential part is similar with the easy AI but instead of easy AI is just able to see the next best move. The medium AI is able to think 3 steps ahead. After the medium AI checks whether this move is a valid or invalid move, the medium AI will create one 8x8x8 virtual board. The 8x8 is the board itself and each cell on the board has 8 potential moves (either valid or invalid). The medium AI would play the potential move on the virtual board and get the score for this potential move, the medium AI then would create another 8x8x8 virtual board for this potential move and get this step's score. Figure 10 shows how the medium AI creates another virtual board and performs the same calculation on the board to get the potential best moves. Note here, the second virtual board is simulating the moves for the player against medium AI. In the second virtual board the medium AI will create the virtual board again to perform the simulation of the third step. After the medium AI simulates all these 3 steps the medium AI will compute the best score and play the move. Essentially, at every step, each potential playable move will result in another "for" loop to simulate the next step.

```
// Think ahead 3 steps
int get_mediumAI_move_cpu(move *moves) {
  int maxScore = -1000;
  for (int i = 0; i < 8; ++i) {
    for (int j = 0; j < 8; ++j) {
      if (board[i][j] == PLAYABLE) {
        char copy_board[8][8];
        int copy_playable_direction[8][8][8];
        copy_board_mem_arr(copy_board, copy_playable_direction, board,
                           playable_direction);
        copy_board[i][j] = current_player;
        int s = capture_potential_pieces_cpu(i, j, copy_board, current_player,
                                             copy_playable_direction);
        mark_playable_positions_simulate_cpu(
            copy_board, copy_playable_direction, (current_player + 1) % 2);
        int res = predict_next_move_cpu(copy_board, copy_playable_direction,
                                        (current_player + 1) % 2,
                                        STEPS_THINK_AHEAD - 1);

        s = s - res;
        // printf("medium score: %d here, row: %d, column: %d\n", diff, i, j);
        moves[i * 8 + j].max = s;
        moves[i * 8 + j].i = i;
        moves[i * 8 + j].j = j;
        if (s >= maxScore) {
          maxScore = s;
        }
      }
    }
  }
  return maxScore;
}
```

Figure 9. Code Snippet for sequential programming medium AI Part1

```
int predict_next_move_cpu(char cboard[8][8], int cplayable_direction[8][8][8],
                          int tmp_current_player, int count) {
  if (count == 0)
    return 0;
  int maxScore = -100;
  for (int i = 0; i < 8; ++i) {
    for (int j = 0; j < 8; ++j) {
      if (cboard[i][j] == PLAYABLE) {
        char copy_board[8][8];
        int copy_playable_direction[8][8][8];
        copy_board_mem_arr(copy_board, copy_playable_direction, cboard,
                           cplayable_direction);
        copy_board[i][j] = tmp_current_player;
        int s = capture_potential_pieces_cpu(
            i, j, copy_board, tmp_current_player, copy_playable_direction);
        mark_playable_positions_simulate_cpu(
            copy_board, copy_playable_direction, (tmp_current_player + 1) % 2);
        int res =
            predict_next_move_cpu(copy_board, copy_playable_direction,
                                  (tmp_current_player + 1) % 2, count - 1);

        s = s - res;
        if (s >= maxScore) {
          maxScore = s;
        }
      }
    }
  }
  return maxScore;
}
```

Figure 10. Code Snippet for sequential programming medium AI Part2

Parallel:

The code snippet below shows the medium AI implementing in parallel programming. The logic and methods of how medium AI determines the best next move is the same as the sequential part. What is different is that the parallel implementation will allocate one thread for each cell on the board to perform the computation. This would increase the speed of the process. First, the AI would copy the board and create a 8x8x8 virtual board which requires a size of 8x8x8 bytes of dynamic memory in GPU and play the next move on the virtual board and calculate the score that it can get if it plays this move. Next, for each playable move, the medium AI would then create another 8x8x8 board and call another kernel function which has 8x8=64 threads and it is for simulating the next step of which is for the player against the medium AI. Recall that medium AI would think 3 steps ahead. In the kernel function predict_next_move shown in Figure 12, the AI would copy the board and create the 8x8x8 virtual board again, allocate 8x8x8 bytes memory in GPU and simulate the next move for the player against the AI. It will then call the kernel function allocating 8x8

threads for it again for the final step simulation. After these 3 steps simulation, the AI would get the best move. Essentially, at every step, each potential playable move will result in another kernel call with 64 threads to simulate the next step.

```
// Think ahead 3 steps
__global__ void get_mediumAI_move(char *cboard, int *cplayable_direction,
                                  int current_player, move *moves) {
  int idx = threadIdx.x;
  if (idx < 64) {
    int i = idx / 8;
    int j = idx % 8;
    // printf("board, %d\n", board[i*8 + j]);
    if (cboard[i * 8 + j] == 3) {
      char *copy_board2;
      int *copy_playable_direction2;
      cudaMalloc((char **)&copy_board2, 64 * sizeof(char));
      cudaMalloc((int **)&copy_playable_direction2, 512 * sizeof(int));
      memcpy(copy_board2, cboard, sizeof(char) * 8 * 8);
      memcpy(copy_playable_direction2, cplayable_direction,
             sizeof(int) * 8 * 8 * 8);
      copy_board2[i * 8 + j] = current_player;
      int s = capture_potential_pieces(i, j, copy_board2, current_player,
                                       copy_playable_direction2);
      mark_playable_positions_simulate(copy_board2, copy_playable_direction2,
                                       (current_player + 1) % 2);
      int *res2;
      cudaMalloc((int **)&res2, 64 * sizeof(int));
      for (int k = 0; k < 64; k++) {
        res2[k] = -1000;
      }
      predict_next_move<<<1, 64>>>(copy_board2, copy_playable_direction2,
                                   (current_player + 1) % 2,
                                   STEPS_THINK_AHEAD - 1, res2);
      cudaDeviceSynchronize();

      int max = -1000;
      for (int k = 0; k < 64; k++) {
        if (res2[k] >= max) {
          max = res2[k];
        }
      }
      if (max == -1000)
        max = 0;
      moves[idx].max = s - max;
      // printf("res2: %d\n", s-max);
      moves[idx].i = i;
      // printf("i: %d\n", i);
      moves[idx].j = j;
      // printf("j: %d\n", j);

      cudaFree(copy_board2);
      cudaFree(copy_playable_direction2);
      cudaFree(res2);
    }
  }
}

__global__ void get_easyAI_move(char *board, int *playable_direction,
```

Figure 11. Code Snippet for parallel programming medium AI Part1

```
__global__ void predict_next_move(char *cboard, int *cplayable_direction,
                                  int current_player, int count,
                                  int *maxScore) {
  if (count == 0) {
    return;
  }
  int idx = threadIdx.x;
  if (idx < 64) {
    int i = idx / 8;
    int j = idx % 8;
    // printf("board, %d\n", board[i*8 + j]);
    if (cboard[i * 8 + j] == 3) {
      char *copy_board2;
      int *copy_playable_direction2;
      cudaMalloc((char **)&copy_board2, 64 * sizeof(char));
      cudaMalloc((int **)&copy_playable_direction2, 512 * sizeof(int));
      memcpy(copy_board2, cboard, sizeof(char) * 8 * 8);
      memcpy(copy_playable_direction2, cplayable_direction,
             sizeof(int) * 8 * 8 * 8);
      copy_board2[i * 8 + j] = current_player;
      int s = capture_potential_pieces(i, j, copy_board2, current_player,
                                       copy_playable_direction2);
      mark_playable_positions_simulate(copy_board2, copy_playable_direction2,
                                       (current_player + 1) % 2);
      int *res2;
      cudaMalloc((int **)&res2, 64 * sizeof(int));
      for (int k = 0; k < 64; k++) {
        res2[k] = -100 * count;
      }
      predict_next_move<<<1, 64>>>(copy_board2, copy_playable_direction2,
                                   (current_player + 1) % 2, count - 1, res2);
      cudaDeviceSynchronize();

      int max = -100 * count;
      for (int k = 0; k < 64; k++) {
        if (res2[k] >= max) {
          max = res2[k];
        }
      }
      if (max == -100 * count)
        max = 0;
      maxScore[idx] = s - max;

      cudaFree(copy_board2);
      cudaFree(copy_playable_direction2);
      cudaFree(res2);
    }
  }
}
```

Figure 12. Code Snippet for parallel programming medium AI Part2

Program Analysis

AI player	Win	Loss	Draw	Winning rate
Easy (Sequential)	60	34	6	60%
Easy (CUDA parallel)	61	36	3	61%
Medium (Sequential)	78	22	0	78%
Medium (CUDA parallel)	76	23	1	76%

Table 1. Results of AI player plays against random player for 100 games

As table 1 showed, against the random player, both sequential and parallel easy level AI can achieve a winning rate of 60%. The winning rate is higher 50% infers that AI takes more effective strategies towards conquering more tiles than the random player. Sequential medium level AI has a 78% winning rate, similar to its parallel counterpart. It shows that medium AIs have more advanced tactics to win the game, validating the effectiveness of looking ahead 3 steps with the minimax algorithm. Meanwhile, the close winning rates within Easy level and Medium level AIs respectively proves that we correctly implement the parallel scheme for both levels of AI.

Game Play	Win	Loss	Draw	First player winning rate

Easy vs Easy_CUDA	45	52	3	45%
Easy vs Medium	19	77	4	19%
Easy vs Medium_CUDA	17	82	1	17%
Easy_CUDA vs Medium_CUDA	24	75	1	24%
Easy_CUDA vs Medium	20	78	2	20%
Medium vs Medium_CUDA	50	48	2	50%

Table 2. Results of AI players play against each other for 100 games

In addition, we simulated more games in between our AI players, see Table 2. It again proves the correctness of parallelism, seeing approximately a 50% winning rate for either Easy or Medium AI playing against their parallel counterpart. Besides, Easy AIs can only secure 20% of the games when battling with Medium AIs, indicating the superiority of Medium AI's algorithm.

AI player	Total time of step calculations in 100 games
Easy (Sequential)	33ms
Easy (CUDA parallel)	739ms
Medium (Sequential)	13360ms
Medium (CUDA parallel)	54320ms

Table 3. Step calculation time of AI players in 100 games

In Table 3, in 100 games, we counted the total time that each AI player spends for the next step calculation when facing the random player. Even though the sequential implementation of AIs take less time than their parallel counterparts, we still see the advantages and potentials of using CUDA parallel programming to speed up the step calculation. The sequential program iteratively checks every tile on the board looking for a playable move. As the AI looks ahead more steps, more layers of iterative checks will happen. We see that from Easy to Medium, sequential program runtime scaled up 404 times, while the parallel program runtime only scaled up 73 times. We predict the sequential step calculation time will exceed parallel calculation if the AI looks ahead more than 5 steps or the game is played on 16x16 board. On the other hand, we suspect the reason for the current slow runtime on the parallel side is due to CUDA and GPU setup overhead. We need time to pass the simulated steps and board to the kernel while each kernel only takes on 64 threads of calculation.