Contents

[1. Introduction to Checkers 3](#_Toc16546566)

[1.1. Explanation of the board 3](#_Toc16546567)

[1.2. How to play? 3](#_Toc16546568)

[1.3. Objective of the game 3](#_Toc16546569)

[2. Representing checkers in solution 4](#_Toc16546570)

[2.1. Initial Draft 4](#_Toc16546571)

[2.2. Finalize Draft 5](#_Toc16546572)

[2.2.1. Representing pieces in terms of bit 5](#_Toc16546573)

[2.2.2. Initializing pieces 5](#_Toc16546574)

[2.2.3. Final board output 6](#_Toc16546575)

[Advantage of using bitboard 6](#_Toc16546576)

[3. Movement in checkers: 7](#_Toc16546577)

[3.1. Using array approach 7](#_Toc16546578)

[3.2. Using bitboard approach 8](#_Toc16546579)

[3.2.1. For basic movement: 8](#_Toc16546580)

[3.2.1.1. Table for Basic movement using bit: 8](#_Toc16546581)

[3.2.2. For jumping movement: 9](#_Toc16546582)

[3.2.2.1. Table for jumping movement using bit: 9](#_Toc16546583)

[4. Generating of pieces that can be moved 10](#_Toc16546584)

[4.1. Implementation of the functions in CPU 10](#_Toc16546585)

[4.1.1. Getting the pieces that can move in CPU 11](#_Toc16546586)

[4.1.2. Getting the pieces that can jump in CPU 11](#_Toc16546587)

[4.1.3. The mask used for verification in CPU 12](#_Toc16546588)

[4.2. Implementation of the function in GPU 12](#_Toc16546589)

[5. Extraction of moves 14](#_Toc16546590)

[5.1. In CPU implementation 14](#_Toc16546591)

[5.1.1. Getting the results 14](#_Toc16546592)

[5.1.1.1. Extracting of possible odd normal moves 15](#_Toc16546593)

[5.1.1.2. Extracting of possible even normal moves 16](#_Toc16546594)

[5.1.2. Extracting of jump moves 16](#_Toc16546595)

[5.1.2.1. Extracting of odd jump moves 17](#_Toc16546596)

[5.1.2.2. Extracting of even jump moves 18](#_Toc16546597)

[5.1.2.3. Extracting of additional jump moves 18](#_Toc16546608)

[5.2 In GPU implementation 19](#_Toc16546609)

[6. Getting the best move set available 24](#_Toc16546610)

[6.1. Using minimax algorithm 24](#_Toc16546611)

[6.2. Using CPU minimax 25](#_Toc16546612)

[6.2.1. Min function of minimax 26](#_Toc16546613)

[6.2.2. Max function of minimax 26](#_Toc16546614)

[6.3. Using GPU minimax 26](#_Toc16546615)

[6.4. Efficiency Concerns 30](#_Toc16546616)

[6.5. Attempted and Implemented Improvements 30](#_Toc16546617)

[7. Benchmarking CPU vs GPU 30](#_Toc16546618)

[8. Findings 30](#_Toc16546619)

[9. Conclusion 31](#_Toc16546620)

# Introduction to Checkers

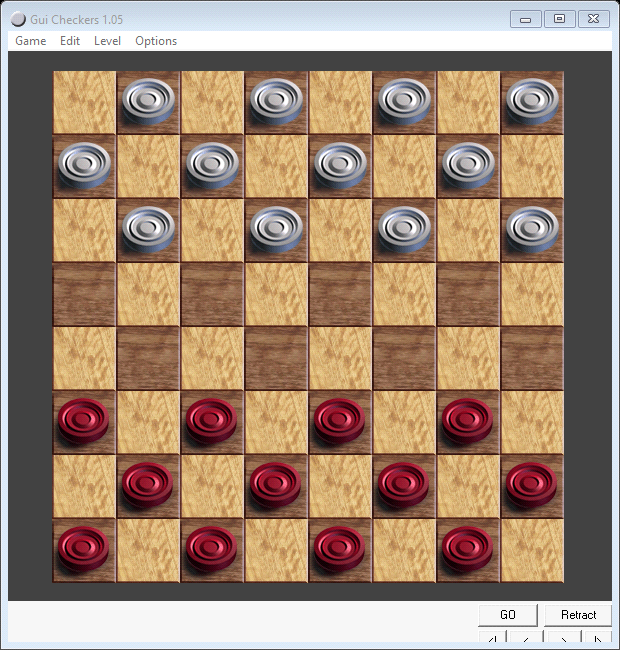


Figure 1. A normal 8 x 8 Checker board

## Explanation of the board

This is a standard 8x8 checker board with pieces on both side of the board, with black on the bottom and white on the top.

## How to play?

Each player is only allowed to move sideways on the color tiles and they must capture the opponent piece if they are close. Upon reaching the end of each of the opponent side, the piece will become a “King” in which that piece is allowed to move backwards.

## Objective of the game

The objective of this game is to capture all of your opponent pieces such that your opponent is unable to play the game. There can be a draw in the game if both players are unable to capture each other’s pieces within a certain amount of moves.

With the explanation of checkers, we will now proceed to represent the game into our project, starting with the data structure and the move generation.

# Representing checkers in solution

## Initial Draft

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

Figure 2a. Initial representation of the board

In this initial draft of the data structure, we planned to go with an array of 64 chars for both Cuda and CPU but we soon realized that this might cause a lot of unnecessary memory allocation on the Cuda side as there are some fields in the checker board that would not be accessed or use in the calculation of further moves, such that we decide that bit board representation for valid moves would be more preferred.

## Finalize Draft

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 28 |  | 29 |  | 30 |  | 31 |
| 24 |  | 25 |  | 26 |  | 27 |  |
|  | 20 |  | 21 |  | 22 |  | 23 |
| 16 |  | 17 |  | 18 |  | 19 |  |
|  | 12 |  | 13 |  | 14 |  | 15 |
| 8 |  | 9 |  | 10 |  | 11 |  |
|  | 4 |  | 5 |  | 6 |  | 7 |
| 0 |  | 1 |  | 2 |  | 3 |  |

Figure 2b. Bit representation of the board (where 0 to 11 = black pieces, 12 to 19 = free movement, 20 to 31 = white pieces)

As each piece in checkers will only touch the black titles, such that it would be better to store the relevant 32 moves as compared to using an array of 64 chars where most of the time only half of the array would be referenced to perform validations of move. Also, the data representation of each pieces would be different as the starting index would be beginning on the bottom right instead of the top right for array structure. Lastly as x64 and Cuda both are little endian so our bit board looks like this

## Representing pieces in terms of bit

As mentioned previously, translating the board into a 32 bit representation is slightly different as compared to array representation, rather than starting from 0 to 63 in a standard array, in the bit board version, it begins from the MSB to the LSB of each rows e.g. the piece at 4 starts at 0x0000 0000 0000 0000 0000 0000 1000 0000 where each corresponding 0000 representing a row

## Initializing pieces

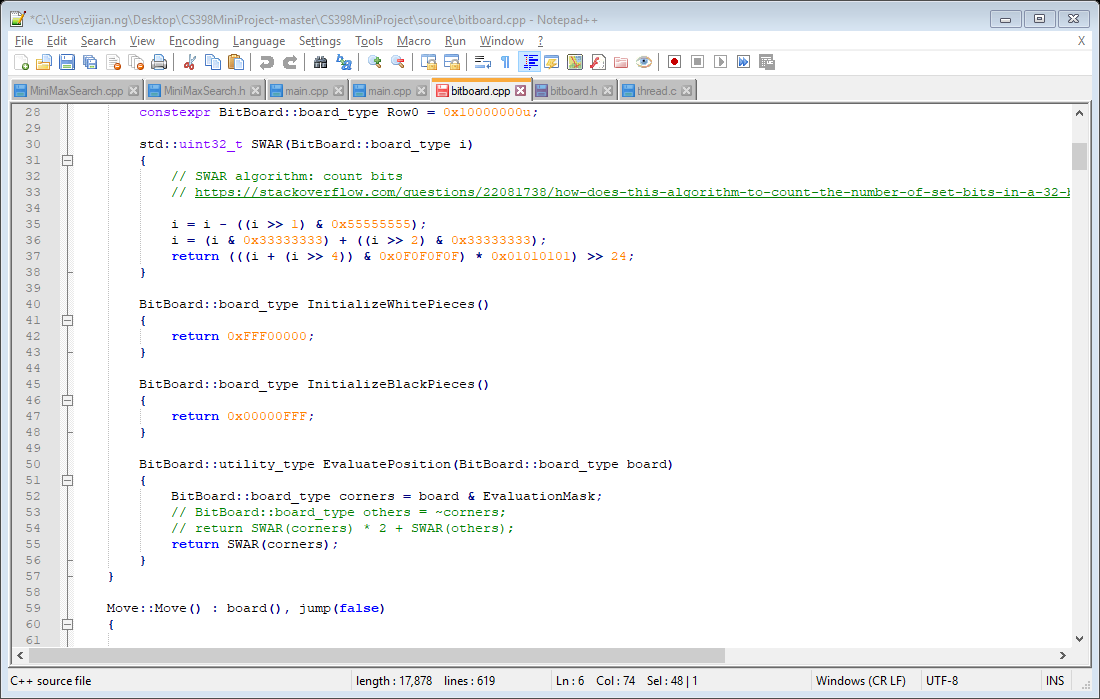


Figure 2c. Initialization of pieces.

In Figure 2c, we create two 32 bit binary to represent both black and white where white begins from the LSB to MSB while black begins from the MSB to LSB.

## Final board output

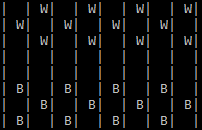


Figure 2d. A Checker board represented in console

This is the final representation of the checkers that will be seen when the program once starts up from the initialization of black and white pieces in figure 2c.

## Advantage of using bitboard

Performing bit wise operation would also be great in a sense that we do not need to perform countless of loops to get the list of possible moves. With the representation of the board done, let's move on to the processing of movement in checkers.

# Movement in checkers:

## Using array approach

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

Figure 3a. Array representation of the board

For movement in checkers, if we are using array of 64 chars then the movement will be much simpler as it will only be only dereferencing of the current pieces such as here for e.g. moving from a piece at 46 to 38 would be simply changing the piece in the array location of 46 to 38, this also applies for capturing of pieces.

## Using bitboard approach

### For basic movement:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 28 |  | 29 |  | 30 |  | 31 |
| 24 |  | 25 |  | 26 |  | 27 |  |
|  | 20 |  | 21 |  | 22 |  | 23 |
| 16 |  | 17 |  | 18 |  | 19 |  |
|  | 12 |  | 13 |  | 14 |  | 15 |
| 8 |  | 9 |  | 10 |  | 11 |  |
|  | 4 |  | 5 |  | 6 |  | 7 |
| 0 |  | 1 |  | 2 |  | 3 |  |

Figure 3b. Bit representation of the board

In the case of using bit representation where each bit above represents a valid move, in order to move from 11 to 14, the program first has to move from 11 to 12 to 13 to 14 which is a total of 3 shift such that we can apply bit shift operations to move pieces to the designated direction of upper right, upper left, lower right and lower left. However, do take note that there are certain restrictions of movement on both black and white unless they have become king pieces, as well as there is a type of shift for even and odd rows which now comes into the picture.

### Table for Basic movement using bit:

|  |  |  |
| --- | --- | --- |
|  | Odd rows | Even rows |
| Moving Upper right | <<5 (left shift by 5) | <<4 (left shift by 4) |
| Moving Upper left | <<4 (left shift by 4) | <<3 (left shift by 3) |
| Moving Lower right | >>4 (right shift by 4) | >>5 (right shift by 5) |
| Moving Lower left | >>3 (right shift by 3) | >>4 (right shift by 4) |

Table 3a. Possible basic movement

The table above is the concluded possible normal move available.

### For jumping movement:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  | W |  |  |  |
|  |  |  | W |  |  |  |  |
|  |  |  |  |  |  | W |  |
|  |  |  | B |  | B |  |  |
|  |  |  |  | B |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Figure 3c. An example of a possible scenario

For capturing of pieces in bit representation, it follows the same implementation of the array approach of checking if there are opponent pieces in the possible direction, in terms of bit represent it will be checking with the existing white location, while at it, there is also a consideration to be added such as jumping to the next possible cell with even and odd rows as well.

### Table for jumping movement using bit:

|  |  |  |
| --- | --- | --- |
|  | Odd rows to even rows | Even to odd rows |
| Moving Upper right | <<5 <<4 (left shift by 5 then left shift by 4) | <<4<<5 (left shift by 4 then left shift by 5) |
| Moving Upper left | <<4 <<3 (left shift by 4 then left shift by 3) | <<3 <<4 (left shift by 3 then left shift by 4) |
| Moving Lower right | >>4 >>5 (right shift by 4 then left shift by 5) | >>5 >>4 (right shift by 5 then right shift by 4) |
| Moving Lower left | >>3 >>4 (right shift by 3 then left shift by 4) | >>4 >>3 (right shift by 4 then right shift by 3) |

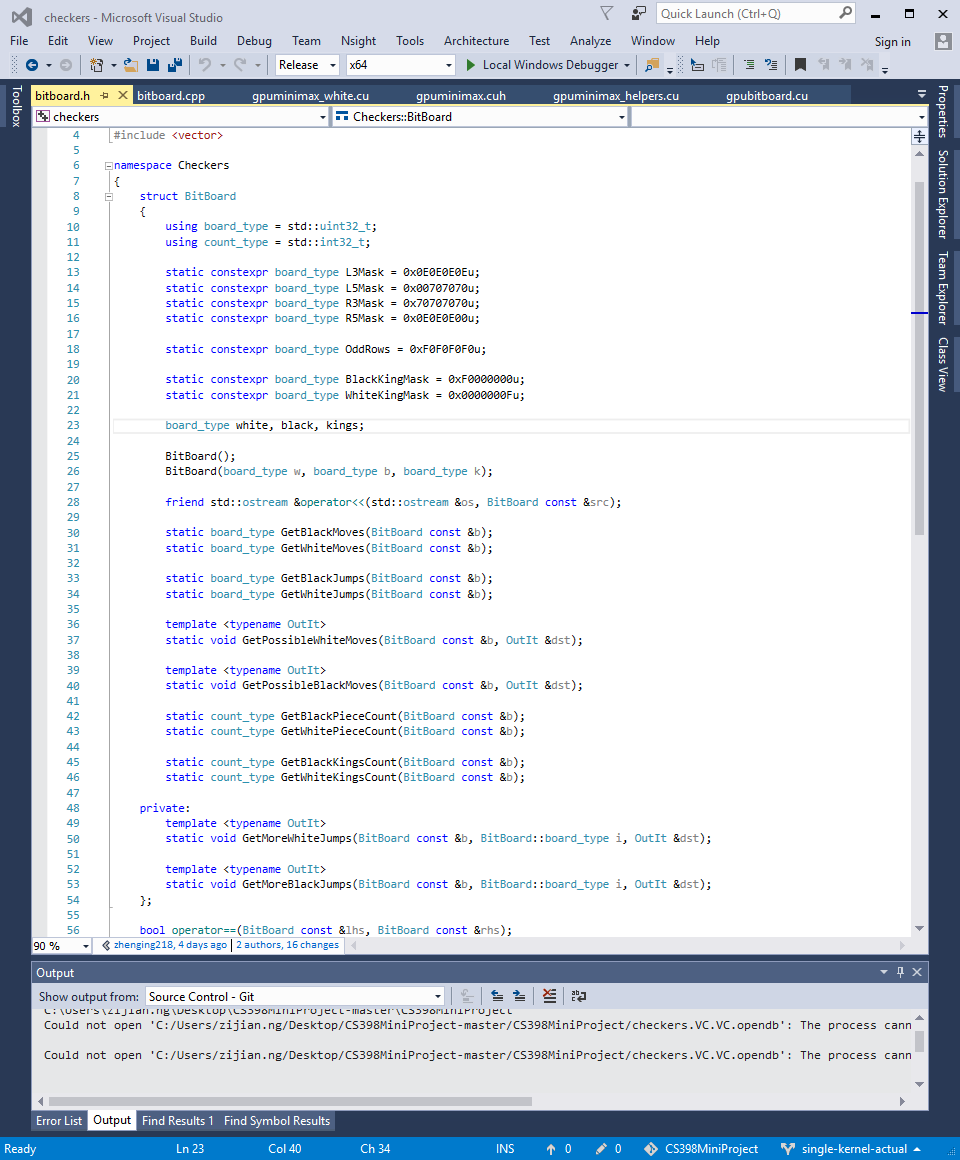
Table 3b. Possible jump movement

The table above is the concluded possible jump move available.

# Generating of pieces that can be moved

With the list of possible moves derived in Section 3, it is time to start generating list of moves to be considered in Section 5.

## Implementation of the functions in CPU

Figure 4a. An implementation of creating moves in CPU

Above is the simple implementation of what is expected to be seen when implementing of the moves in CPU

## Getting the pieces that can move in CPU

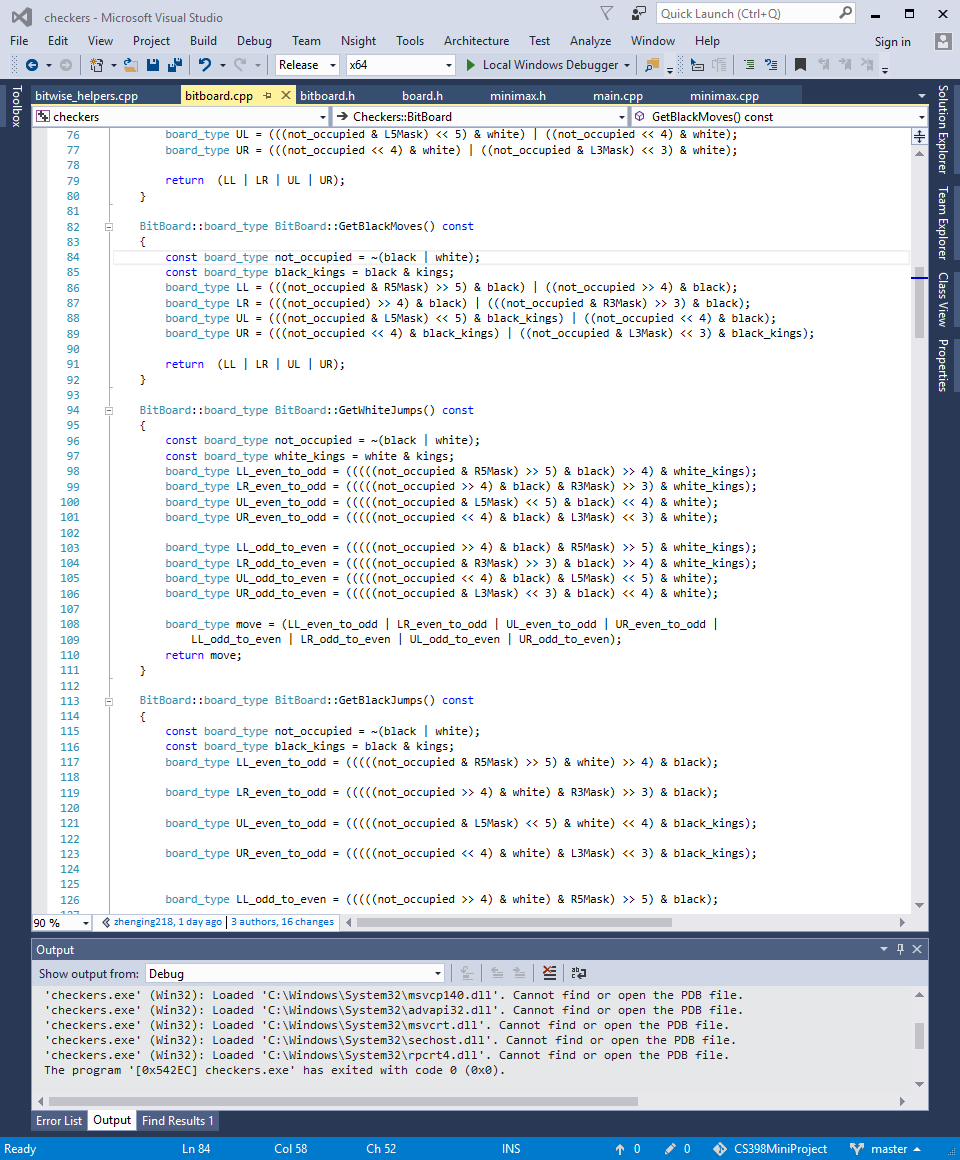


Figure 4b. An implementation of creating normal moves in CPU

As seen in Section 3, there is a possibility of a piece being able to move in 4 directions if there is nothing in the adjacent titles such that there will be lower left, lower right, upper left and upper right for both even and odd titles. Since we are going with bit implementation, we are constantly keep track of where the black and the white pieces are, to retrieve the empty cells for the moves considerations. While checking for the possible moves, it is also important to check if the moves are valid via the masks.

## Getting the pieces that can jump in CPU

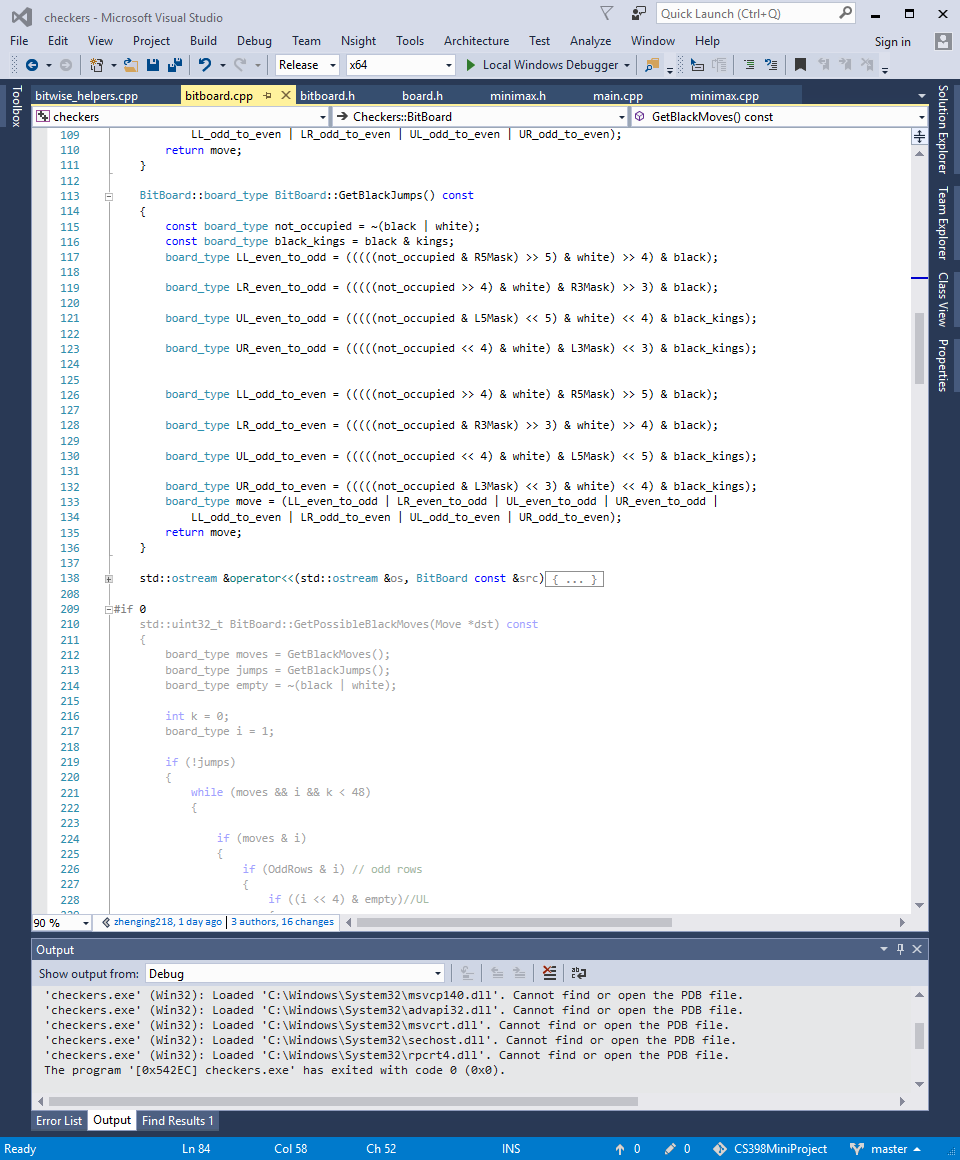


Figure 4c. An implementation of creating jump moves in CPU

With some implementation similar to creation of normal moves in Section 4.1, this time there is a need to check if the piece that is in the adjacent tiles are the opponent pieces. If the piece that is moving to any adjacent pieces has opponent tiles, the piece is then moved to the next tile after the opponent piece with their shifting based on the piece current position.

## The mask used for verification in CPU

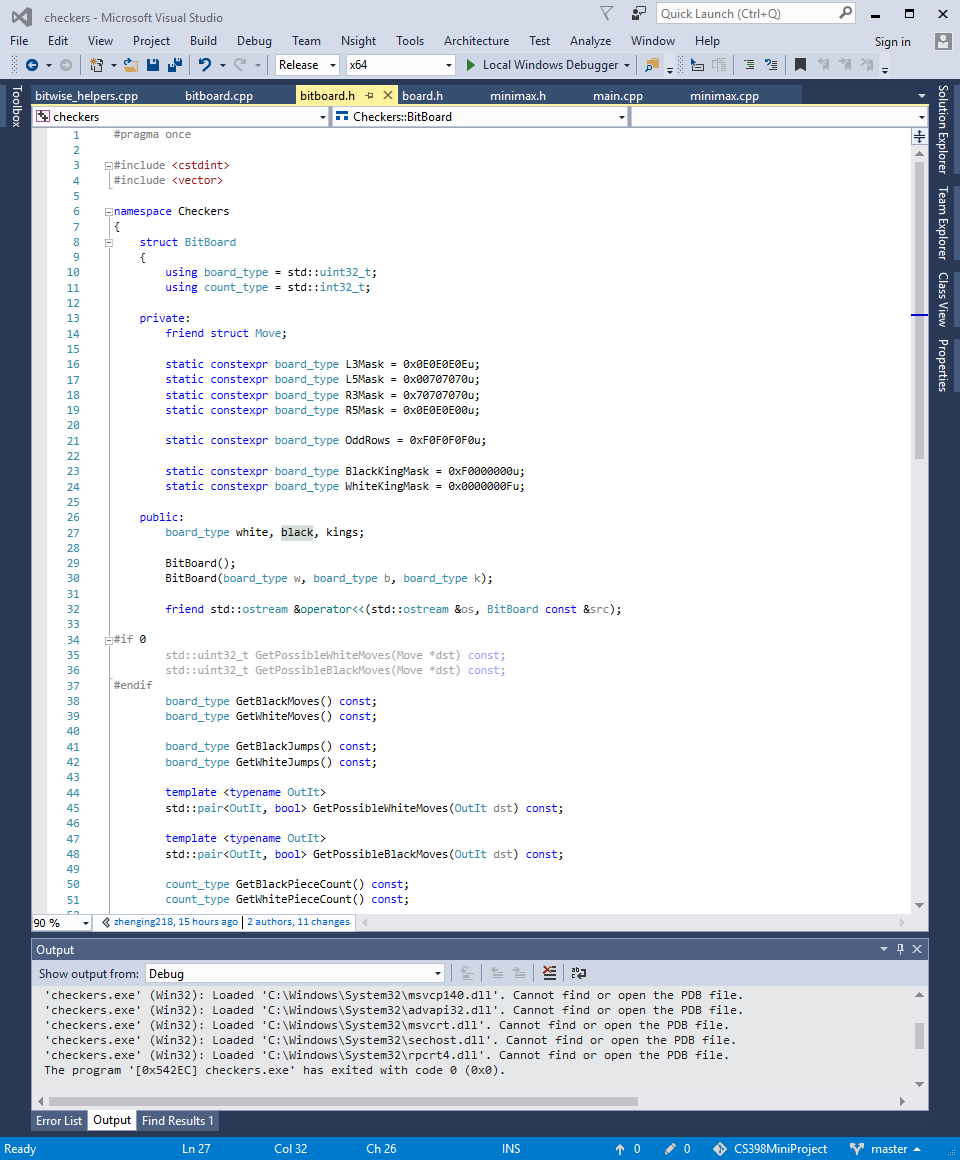


Figure 4d. The masks for checking valid moves

The L3,L5,R3,R5 masks are what is used to make a comparison with to see if the pieces can move to the corresponding even or odds tiles, as the king pieces in this case black could move additionally in the opposite direction thus the check of UL and UR is catered to the king pieces and for white pieces LL and LR. There is no need for if statements if we are using bit wise operations. Finally, after getting the possible moves in the 4 directions, all that is left is to return the bit results of those pieces that can be moved.

## Implementation of the function in GPU

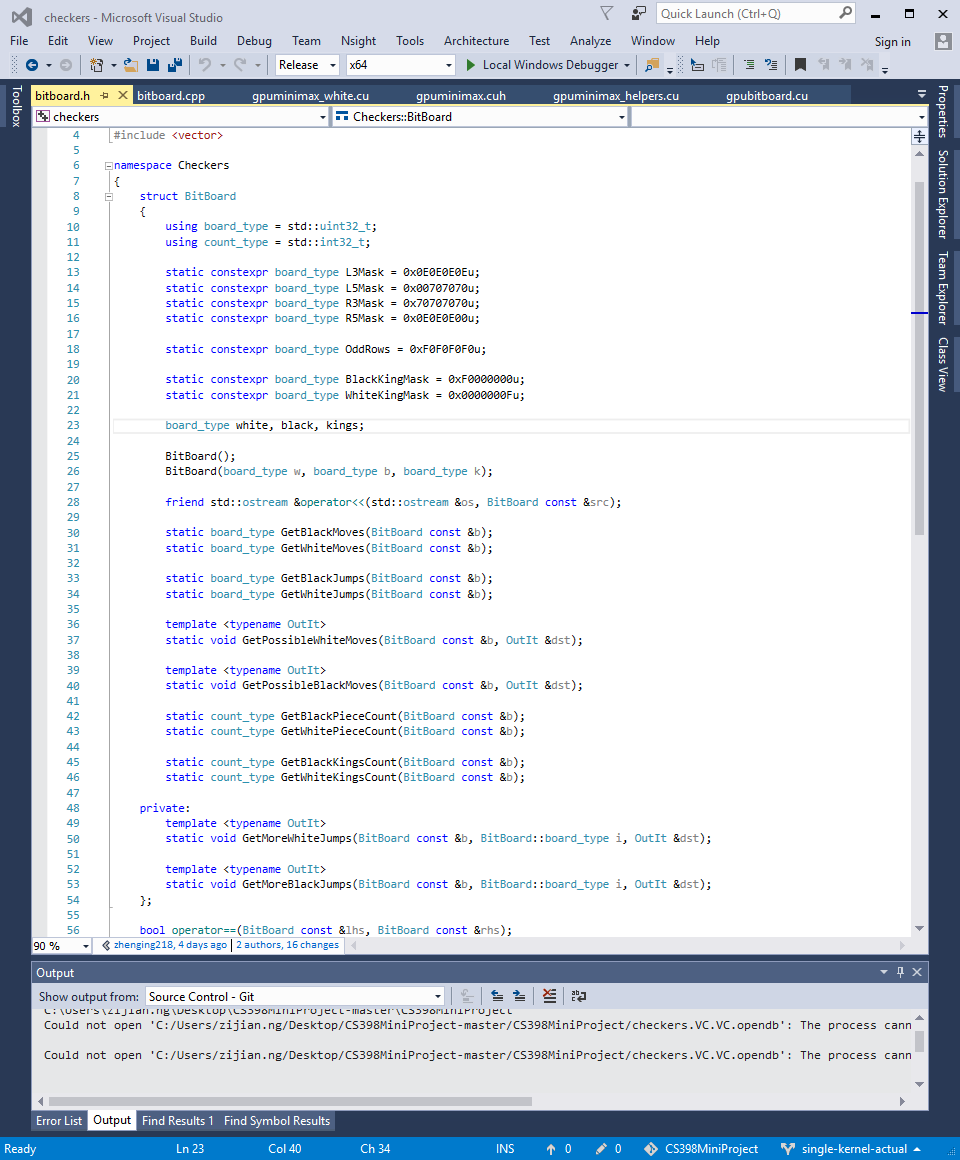


Figure 4e. An implementation of creating normal moves in GPU

Creating of moves in GPU is the same procedure in CPU except this time, there needs to be an indication of \_\_host\_\_ and \_\_device\_\_ derivatives such that both CPU and GPU can use it when it comes to the minimax portion of the GPU side.

* + 1. Getting the pieces that can move in GPU

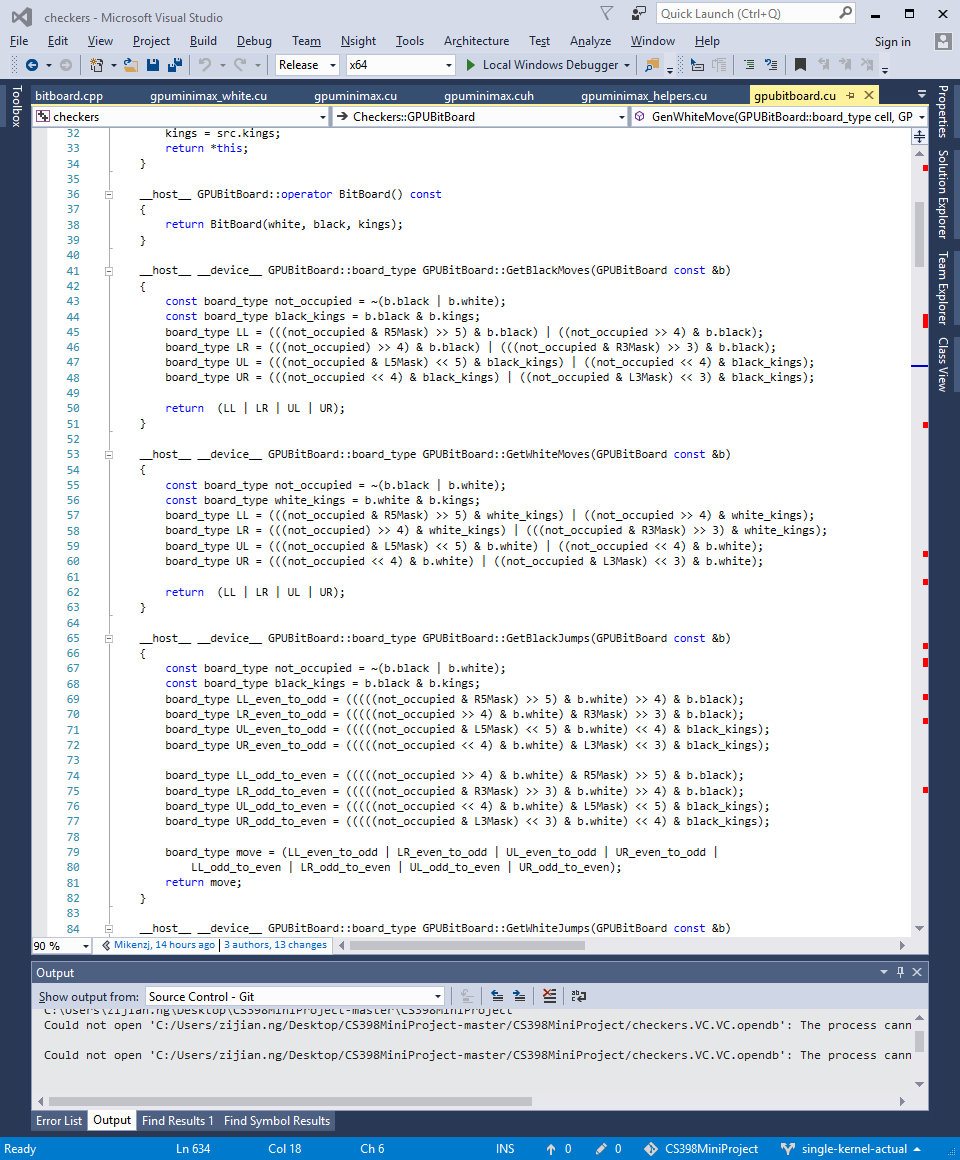


Figure 4f. An implementation of creating normal moves in GPU

In the GPU version of generating of moves, the implementation is almost identical to the version in CPU except this time, we need to input a parameter for GPU to be able to calculate.

* + 1. Getting the pieces that can jump in GPU

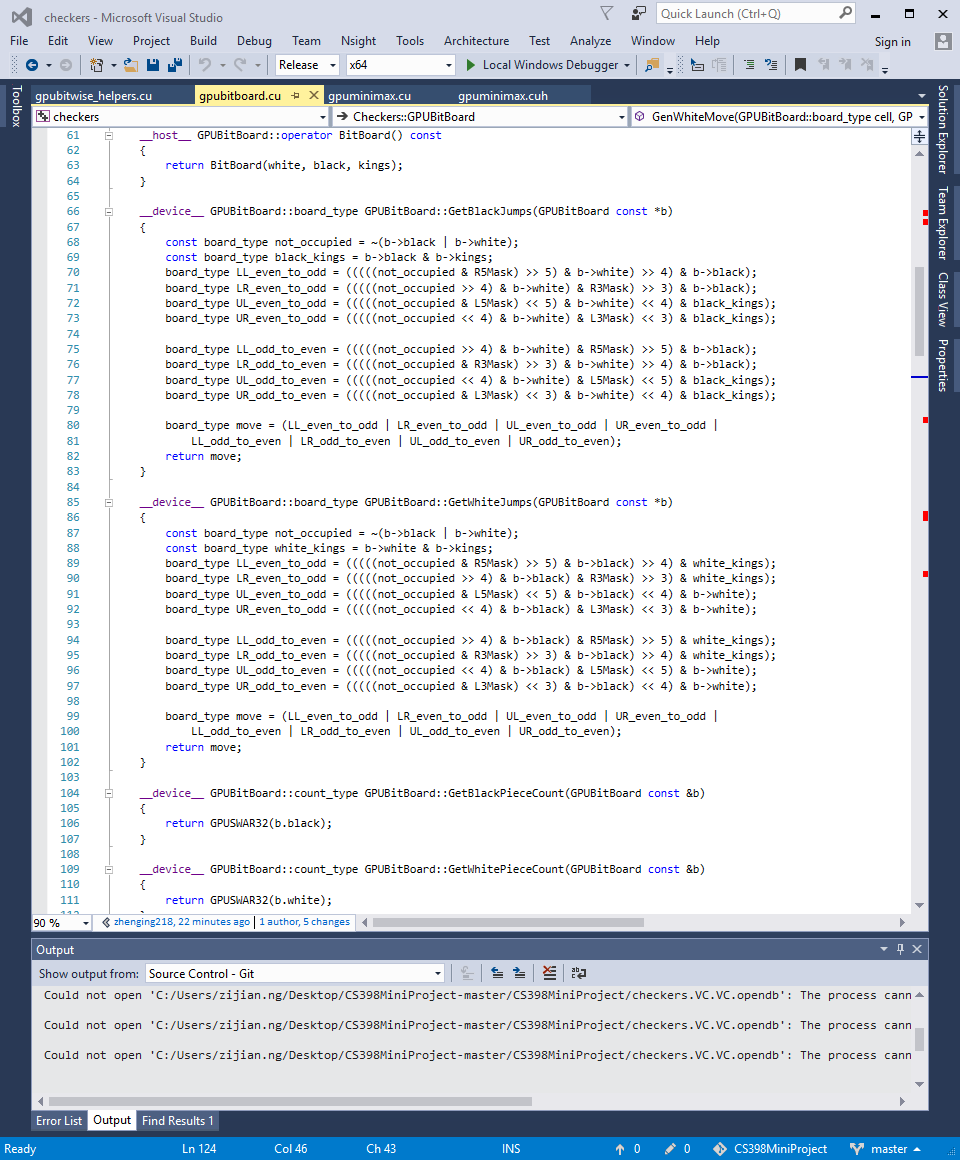


Figure 4g. An implementation of creating jump moves in CPU

In the GPU version of generating of moves, the implementation is almost identical to the version in CPU except this time, we need to input a parameter for GPU to be able to calculate.

* + 1. The mask used for verification in CPU

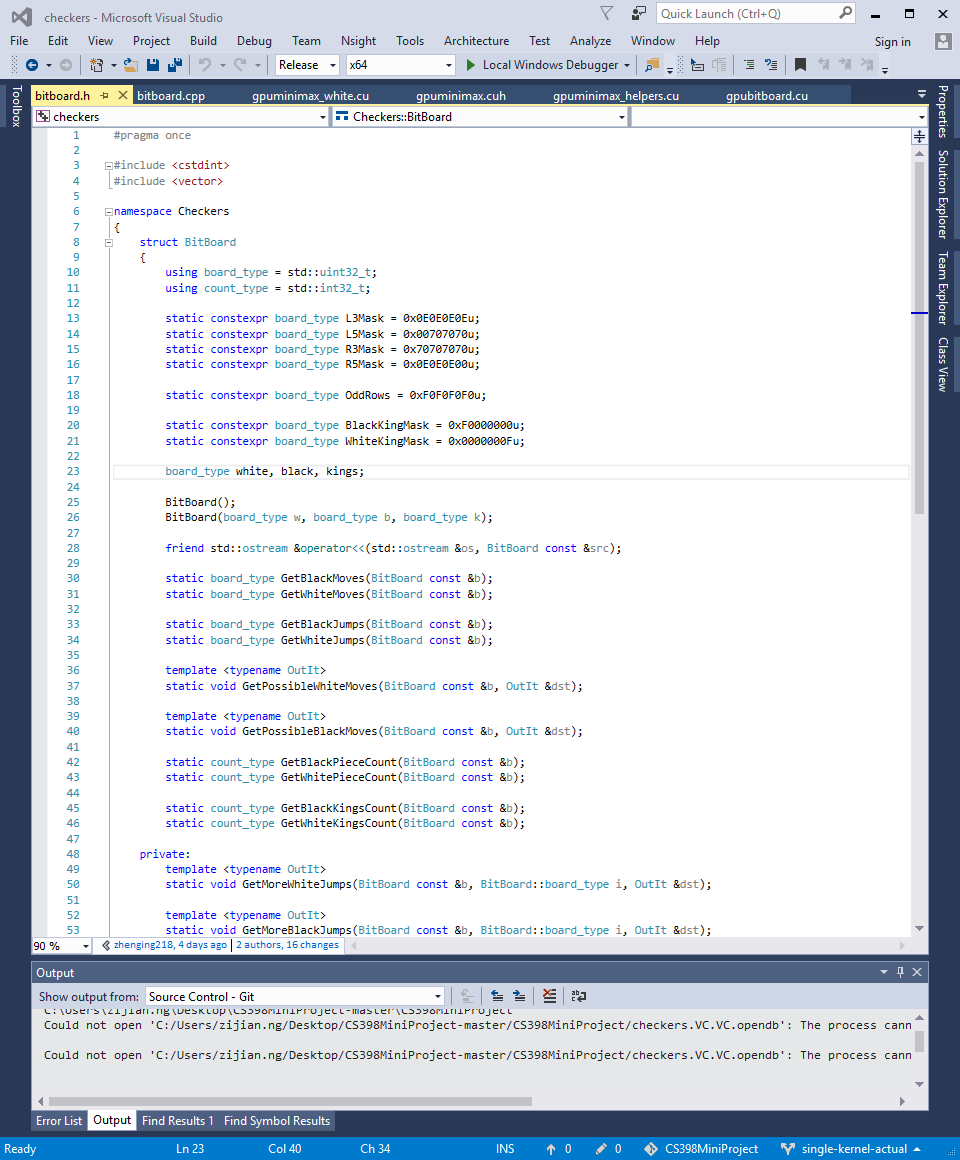


Figure 4h. The masks for checking valid moves

In the GPU version of the mask, the implementation is almost identical to the version in CPU

# Extraction of moves

## In CPU implementation

## Getting the results

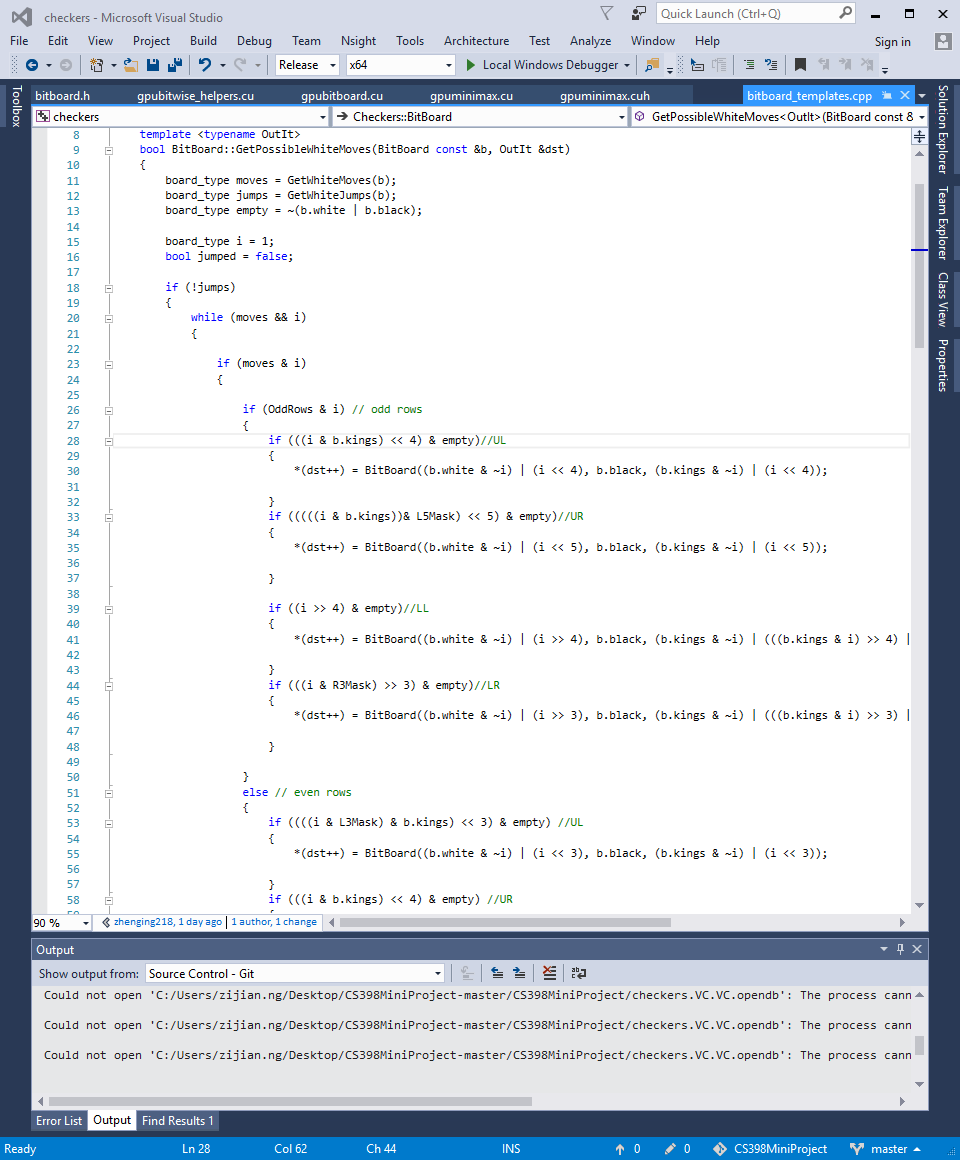


Figure 5a. Extraction of possible moves

In section 4, after the extraction of all the possible pieces that can move or jump, we would proceed to do some bitwise operation to create the list of possible move. In the game of checkers, the rule is that jump must be performed if there is two different color at adjacent tiles, such that jump movement would be performed immediately (refer to Section 5.1.2), the loop is done 32 as it is the list of possible cell that could be moved to and from.

## Extracting of possible odd normal moves

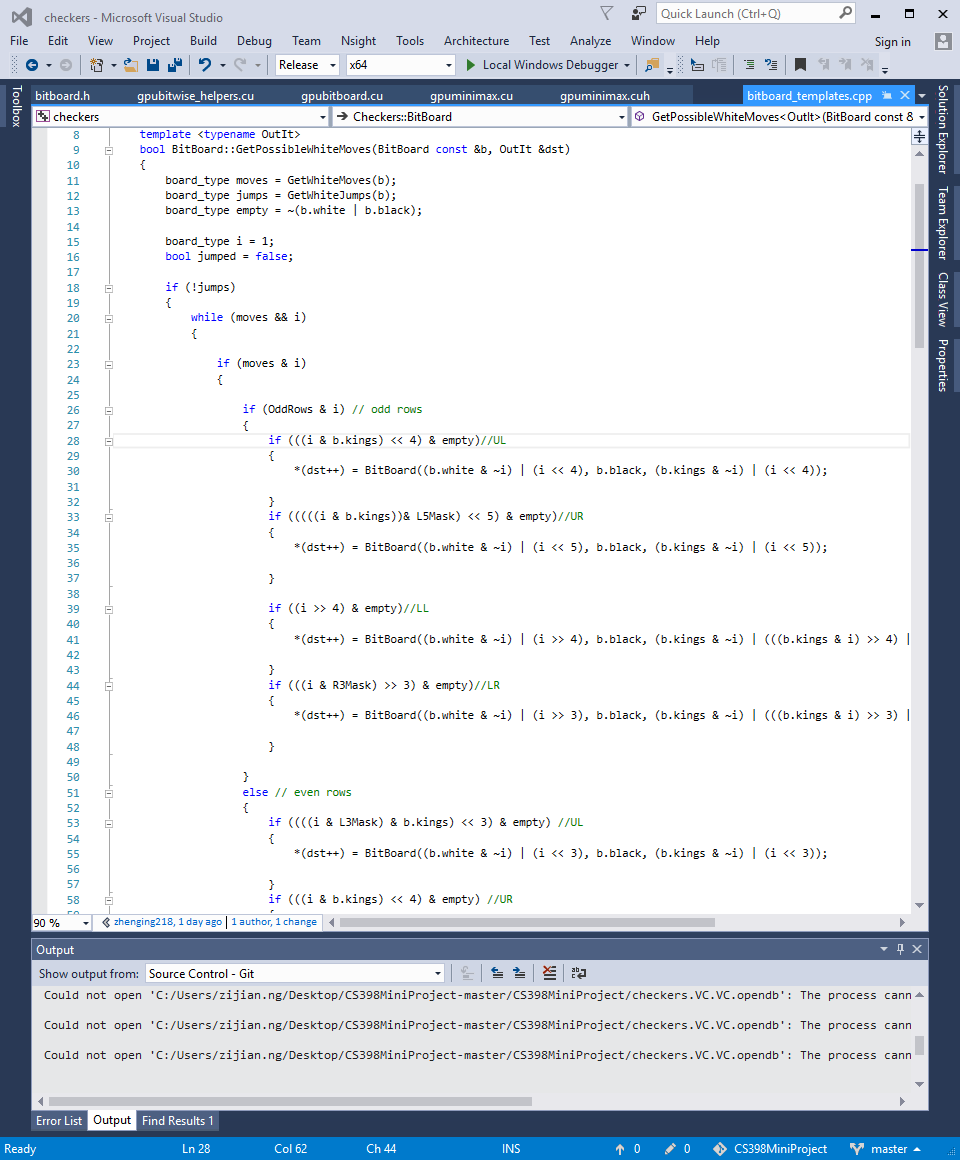


Figure 5b. An implementation of extracting odd moves in CPU

Since there could be 2 possible direction and 4 for a king, there needs to be a check for each possible direction. For the UL and UR moves, this is only exclusive to the king pieces. In the sample of the UL of king, we check that if the king piece could backtrack or is becoming a king, if it could, we would call for Bit Board constructor to make that piece into a king if it isn’t at the end of the other end. As both normal pieces and king share the LL and LR, moving in those tiles shares the same implementation as UR and UL. Explanation of such moveset is to remove the current position of that piece and moving it to the next possible move using the or operator and lastly this implementation here is to check if that piece is a king and thus move the king position.

## Extracting of possible even normal moves

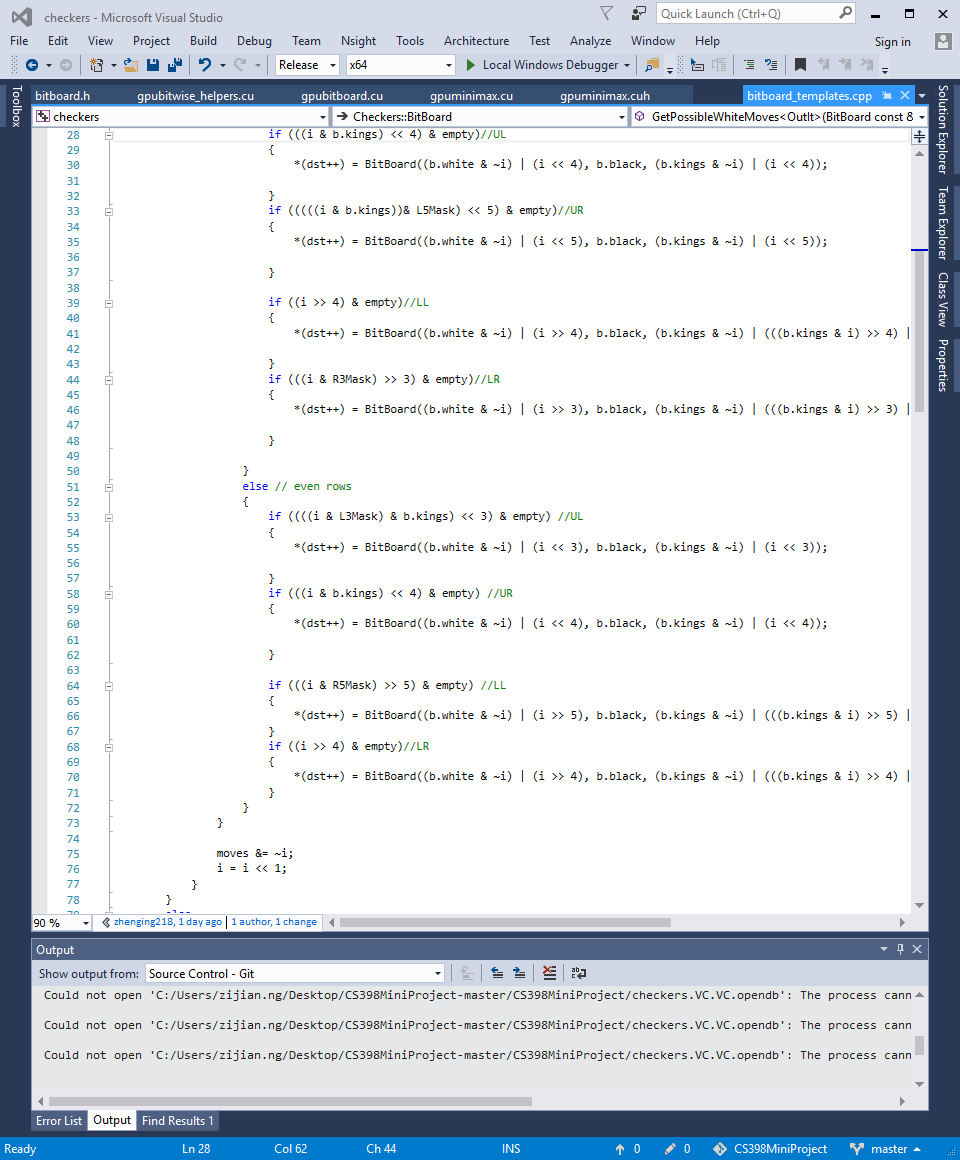


Figure 5c. An implementation of extracting even moves in CPU

Extraction of possible even normal moves follows the same implementation as the one in 5.1.1.1 except all the checks are now used with even movements.

## Extracting of jump moves

Extraction of jump moves now factors in if the adjacent pieces could be captured and if there is no piece on the next diagonal tile.

## Extracting of odd jump moves

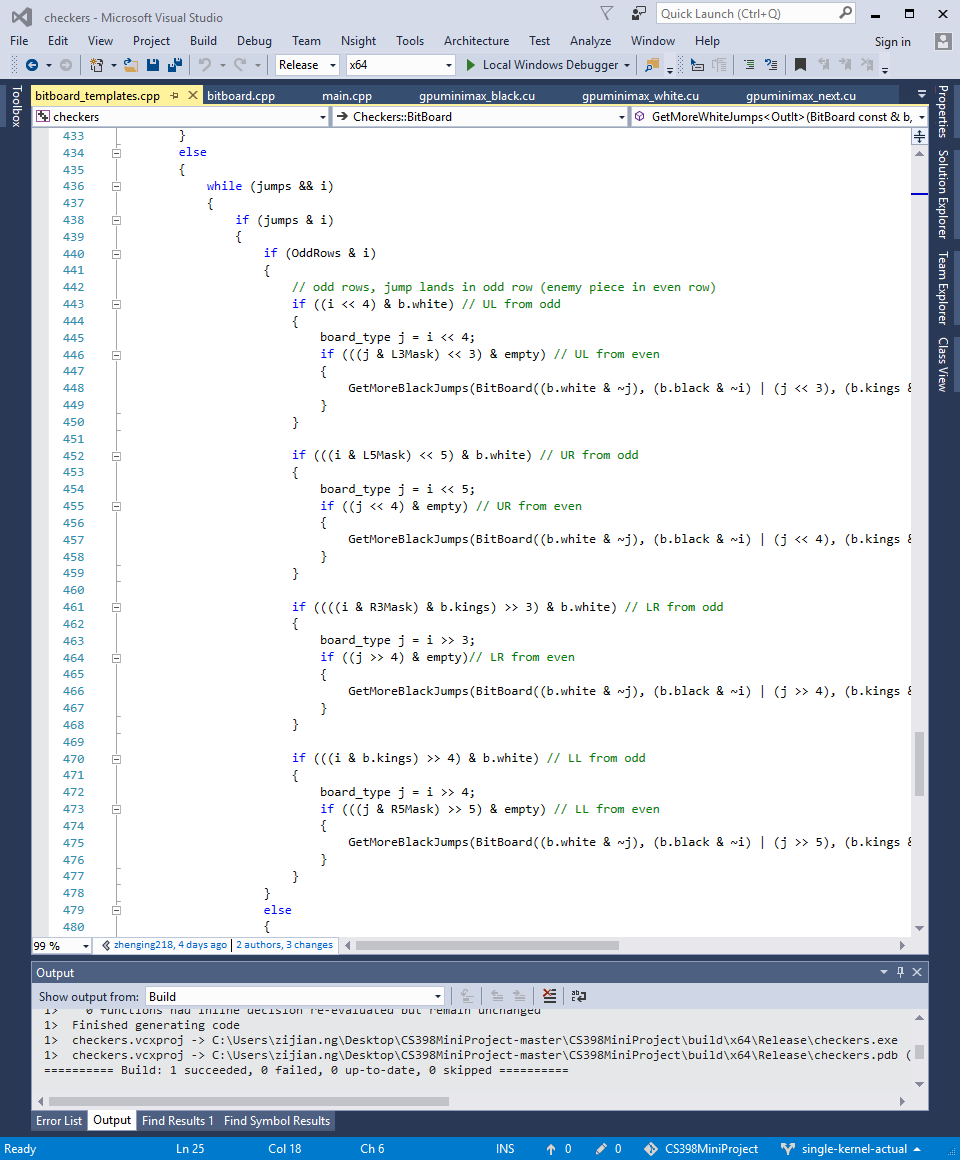


Figure 5d. An implementation of extracting odd jump moves

Taking an example in UL, the check in the move in << 4, gets whether is there any opponent piece and the operation (j & L3Mask) <<3) & empty, checks if the current piece could jump to the next diagonal tile. This function also calls to check for additional jump moves as checkers needs you to perform a jump move until you cannot perform it.

## Extracting of even jump moves

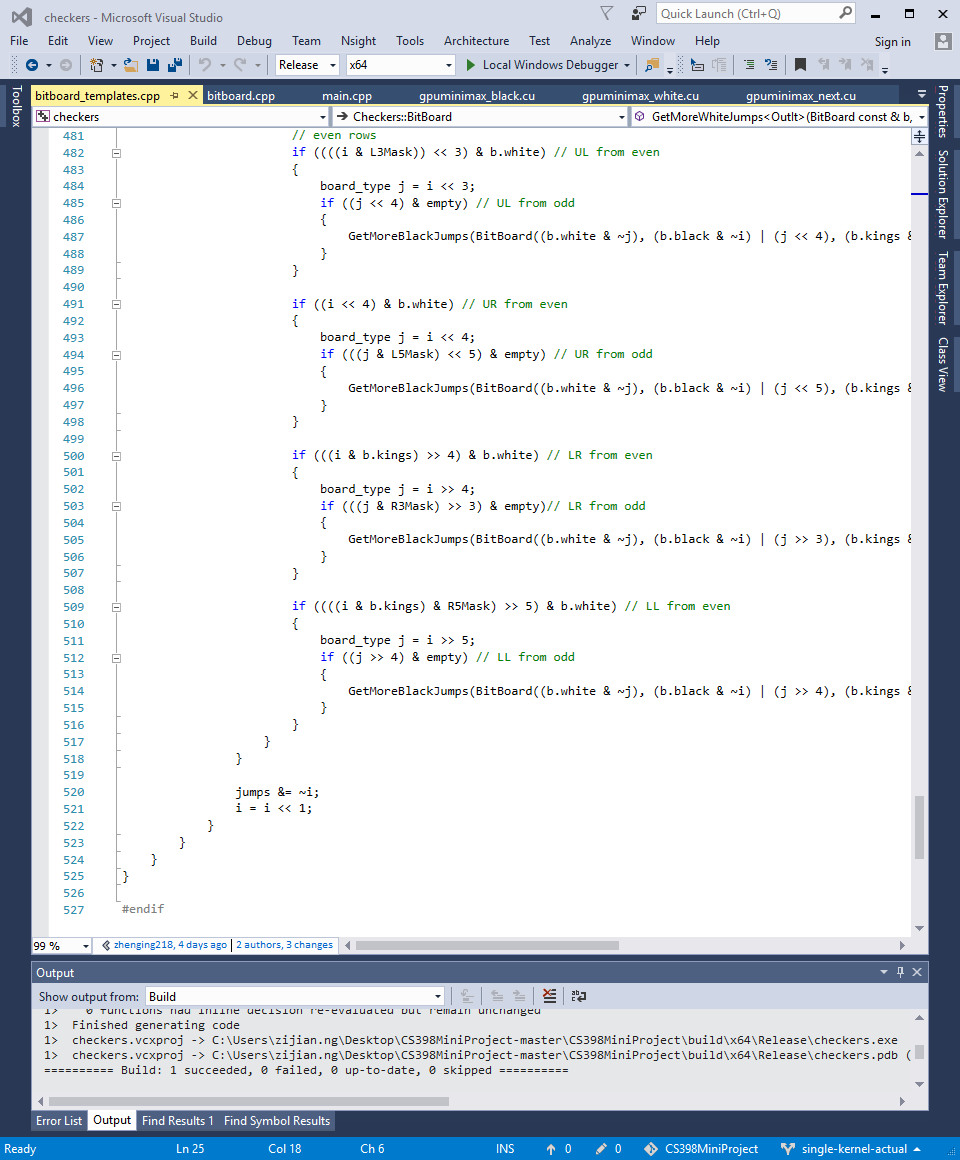


Figure 5e. An implementation of extracting even jump moves

Extraction of possible even jump moves follows the same implementation as the one in 5.1.2.1 except all the checks are now used with even movements.



## Extracting of additional jump moves

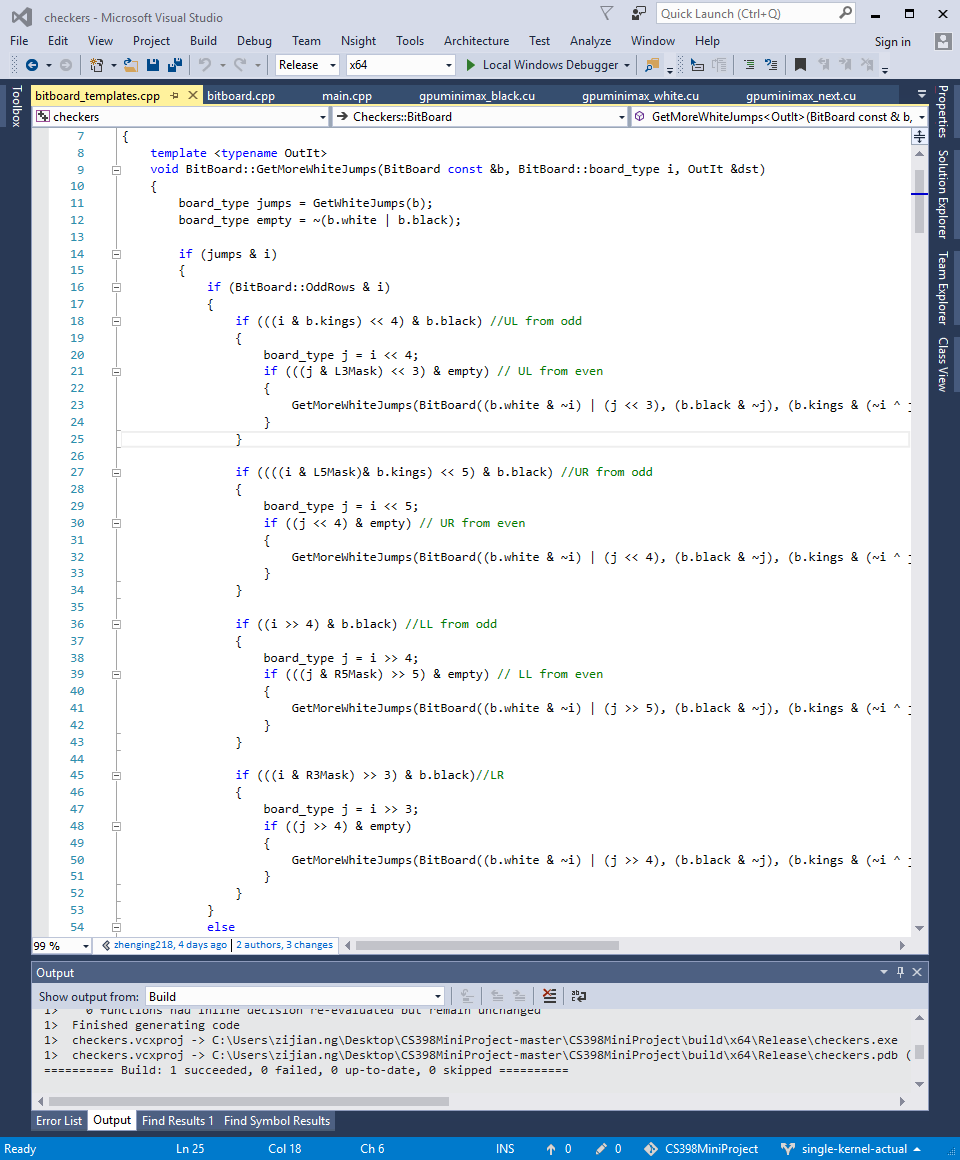


Figure 5f. An implementation of extracting even jump moves

In this function of extracting more jump, a recursive call is made such that it will keep getting possible list of moves until there isn’t a possible jump available.

## In GPU implementation

In the GPU implementation, most of the codes that were in CPU could be reused into GPU with a slight change.

## Extracting of possible odd normal moves

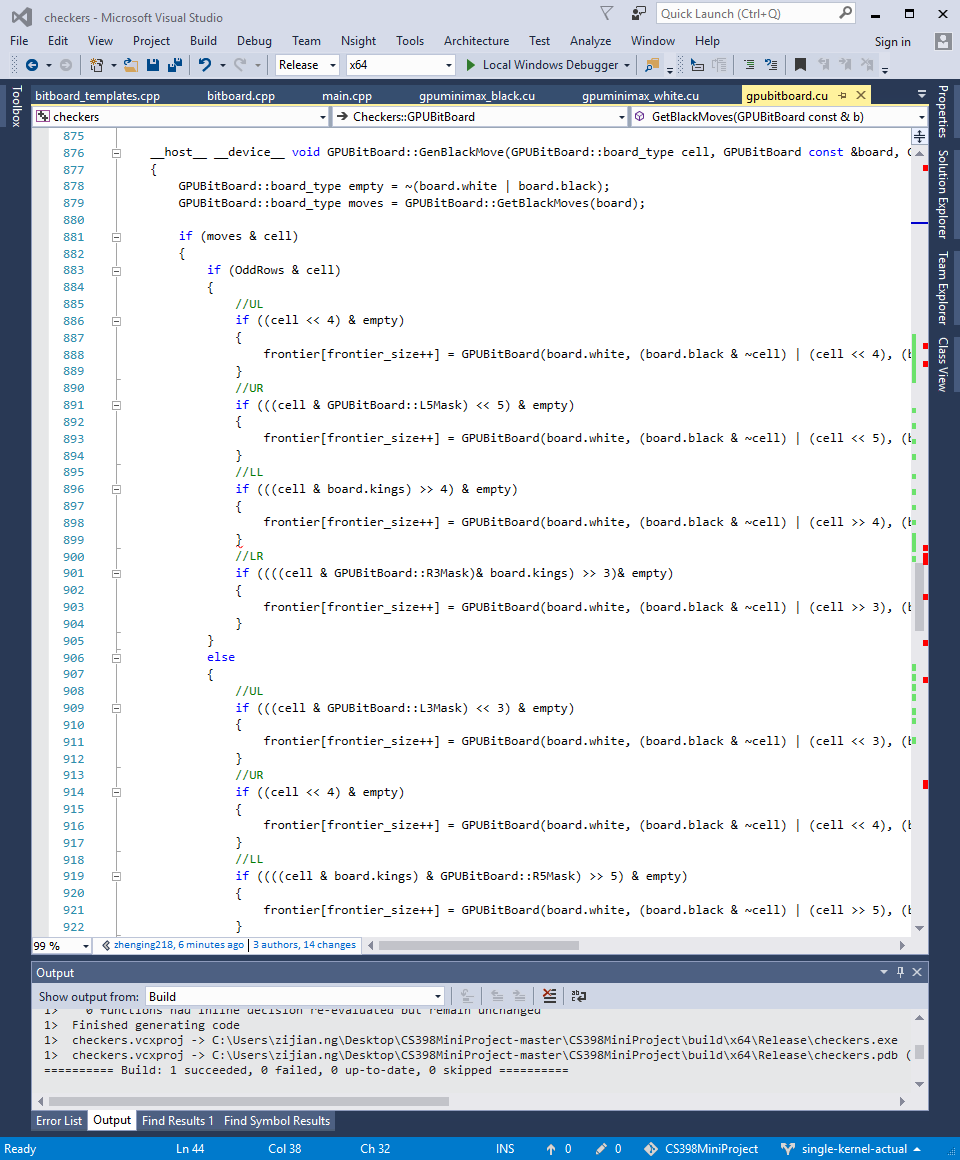


Figure 5g. An implementation of extracting odd moves in GPU

Implementation is the same as CPU in Figure 5b except now it outputs a list of possible moves into a frontier param

## Extracting of possible odd normal moves

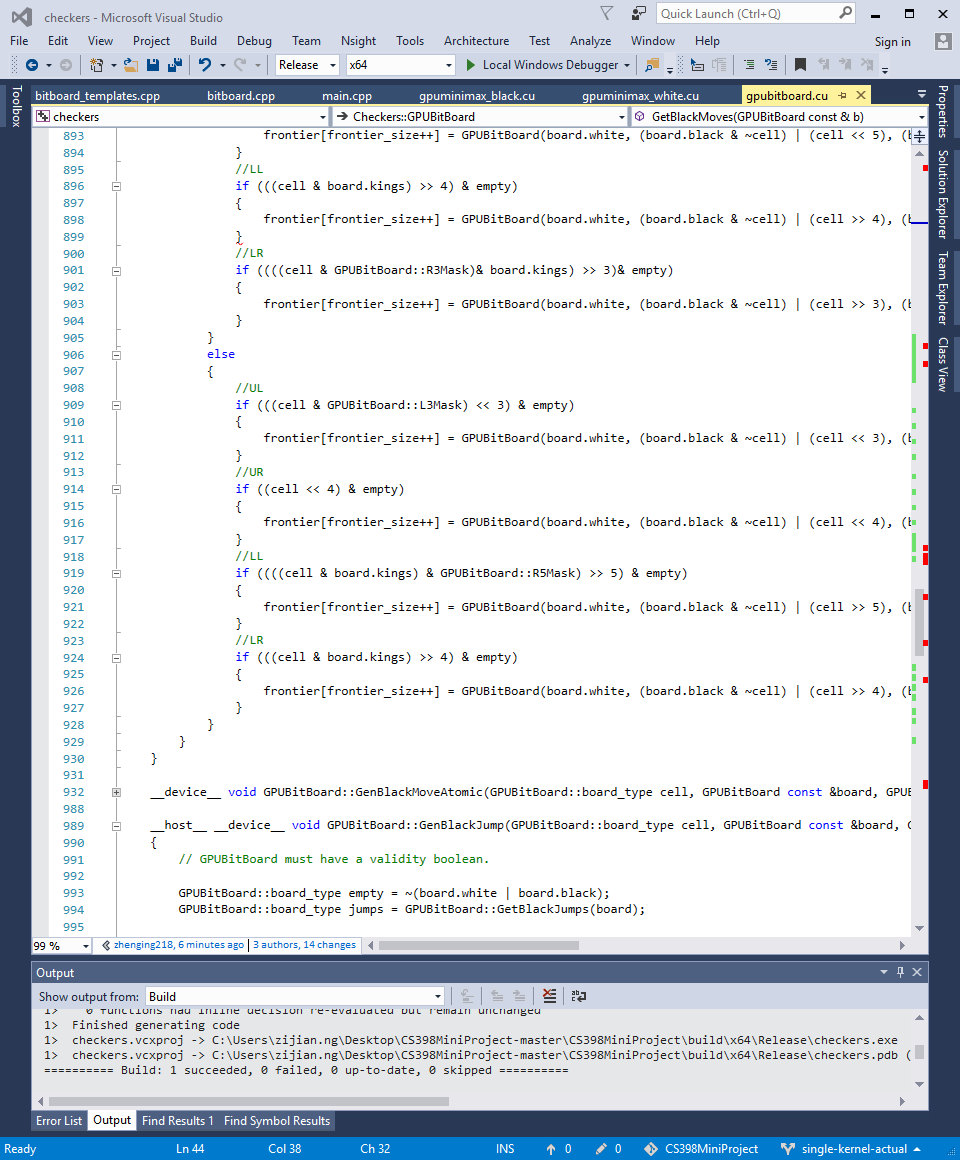


Figure 5h. An implementation of extracting odd moves in GPU

Implementation is the same as CPU in Figure 5c except now it outputs a list of possible moves into a frontier param

## Extracting of possible odd jumps moves

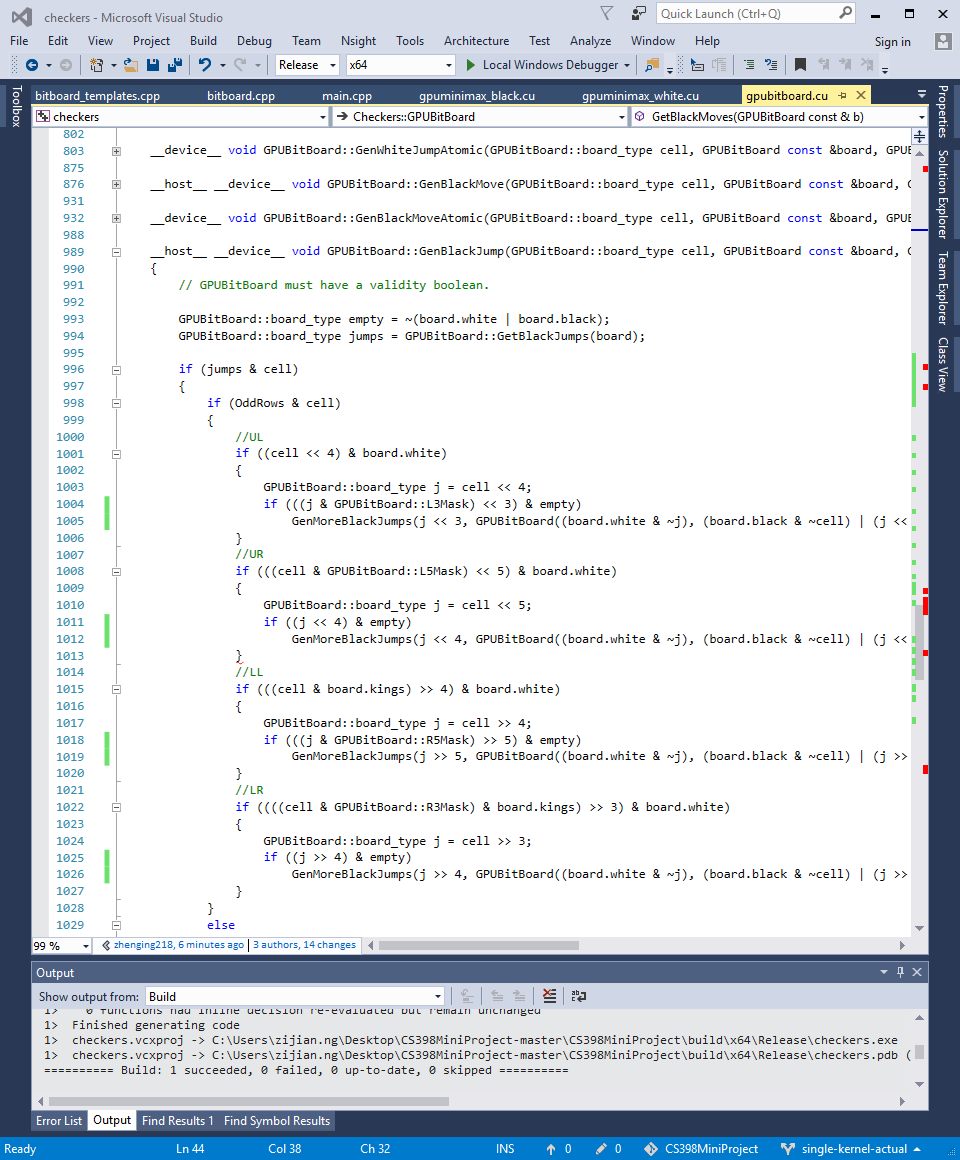


Figure 5i. An implementation of extracting odd jumps in GPU

Implementation is the same as CPU in Figure 5d except now it outputs a list of possible moves into a frontier param

## Extracting of possible even jumps moves

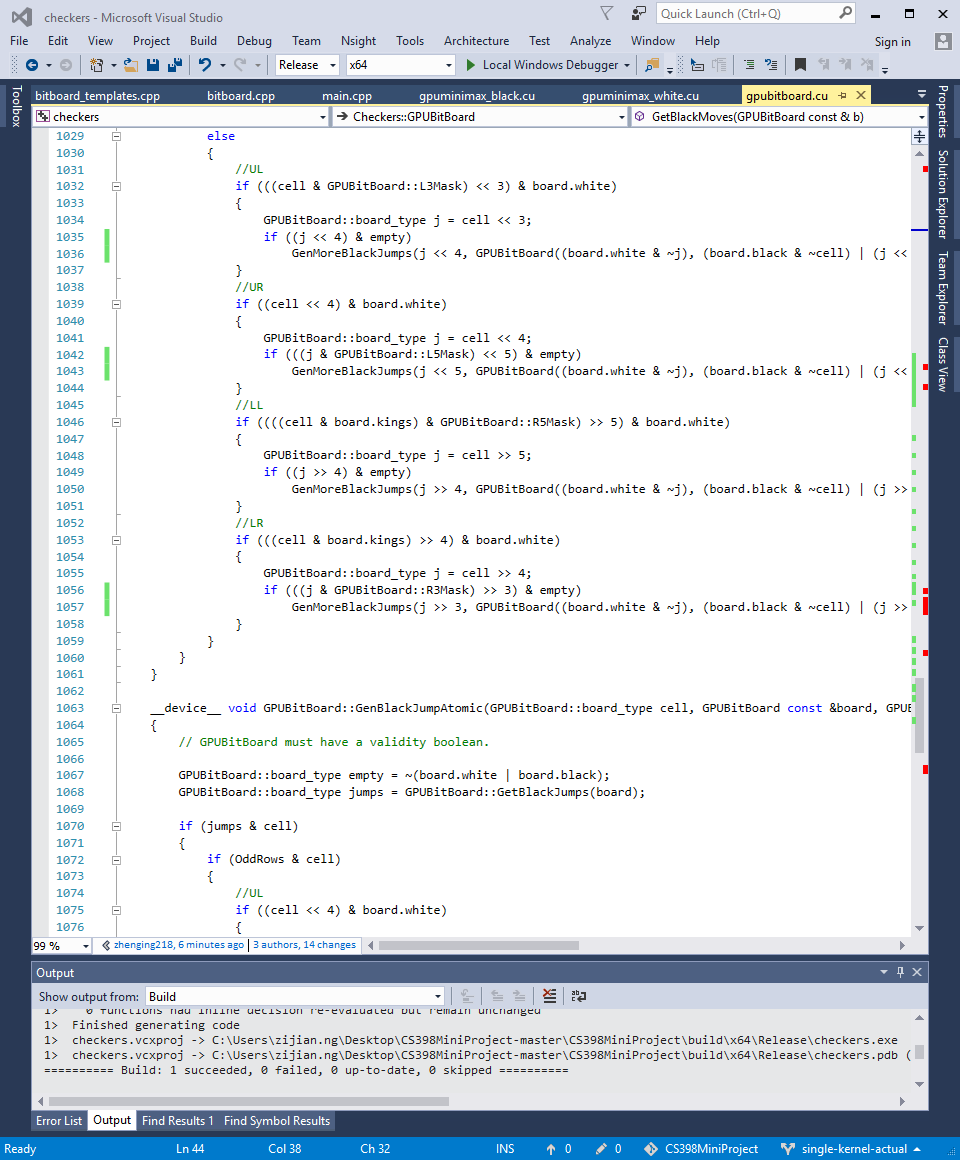


Figure 5j. An implementation of extracting even jumps in GPU

Implementation is the same as CPU in Figure 5e except now it outputs a list of possible moves into a frontier param

## 5.2.5 Extracting of possible even jumps moves

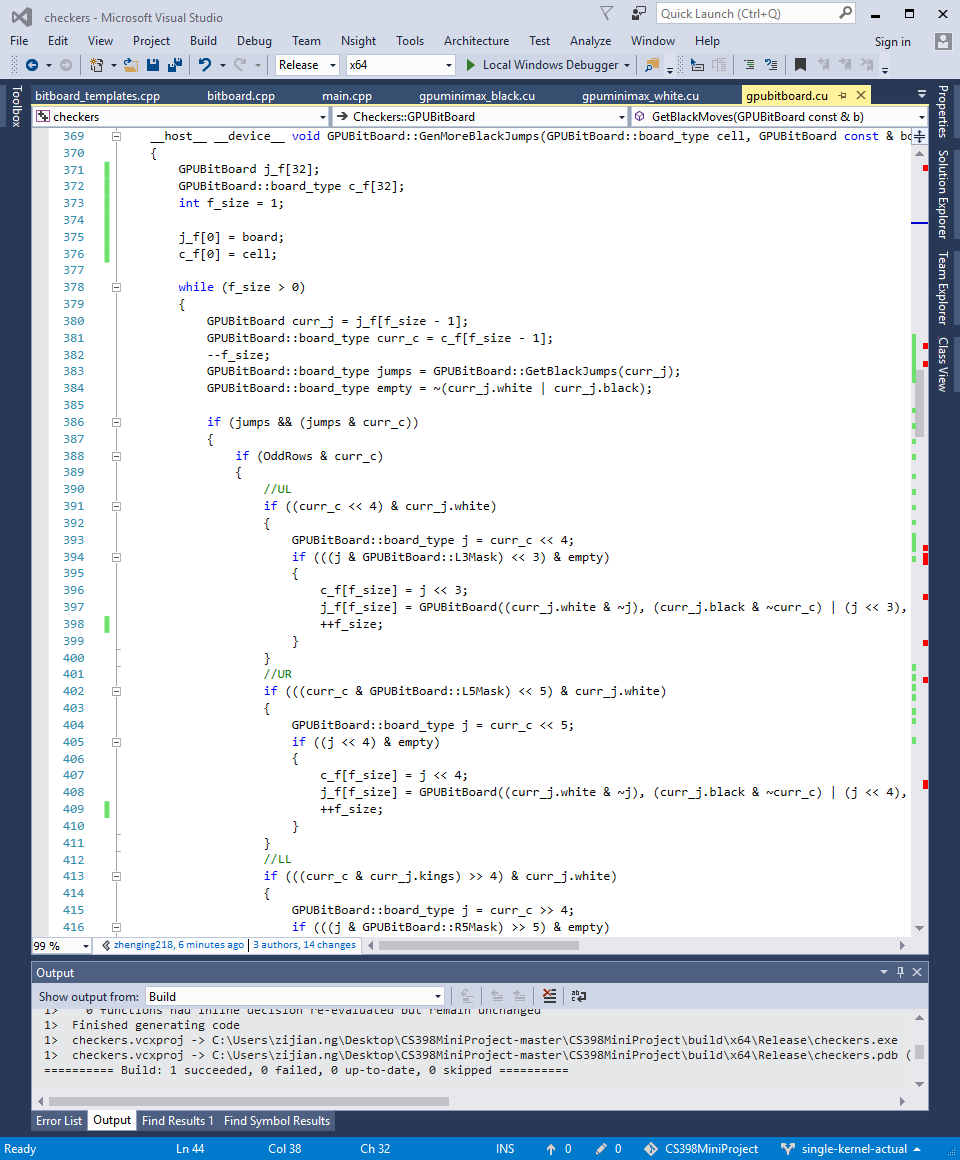


Figure 5k. An implementation of extracting additional jumps in GPU

In the jump implementation of GPU, it is slightly different as there is a consideration to the stack size that Cuda has by default, thus there is a need to declare a standard array size and then storing the list of possible jump moves into j\_f after which it will then be transferred over to the frontier at the end in Figure 5m.

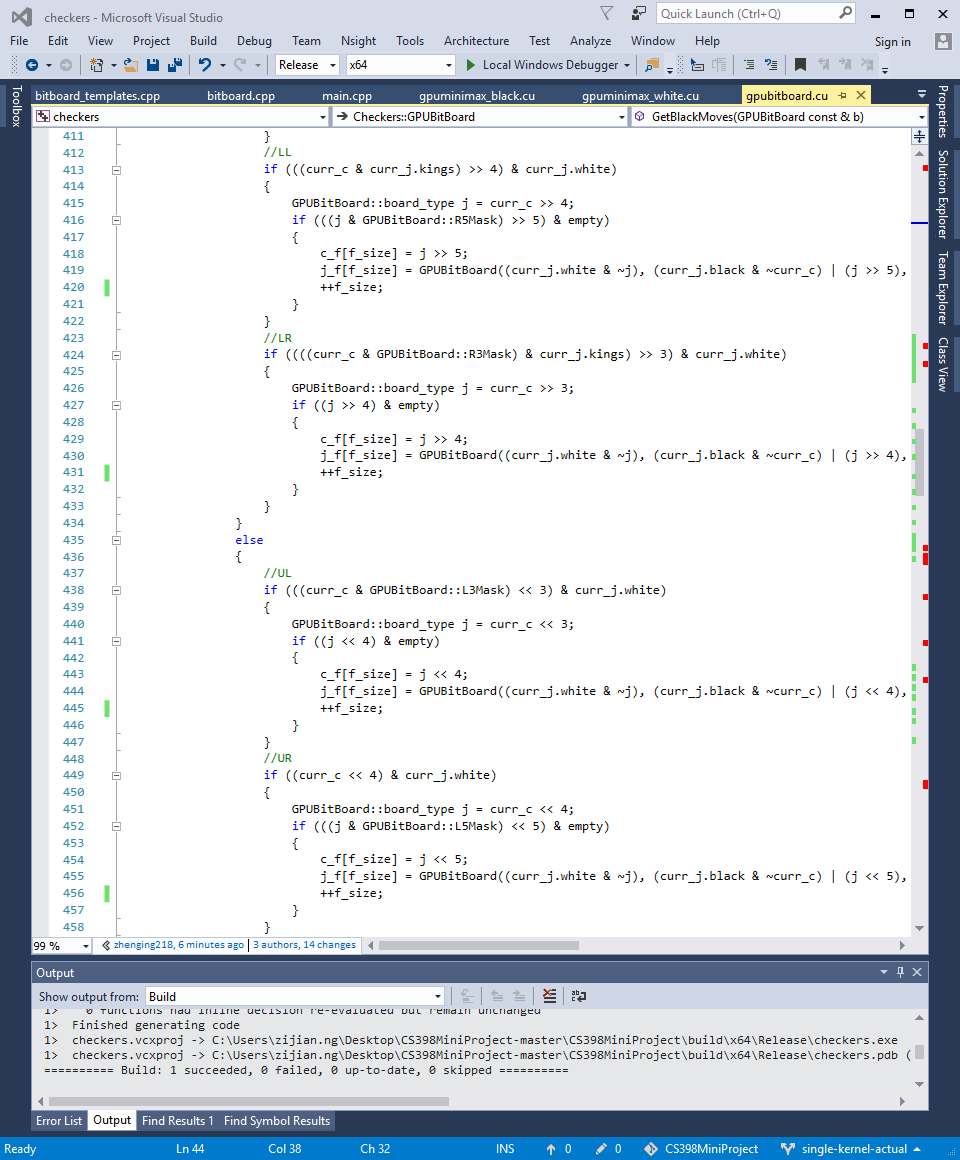
+

Figure 5l. Continual of 5k.

In this function, it follows the implementation of getting odd and even possible jump with checks.

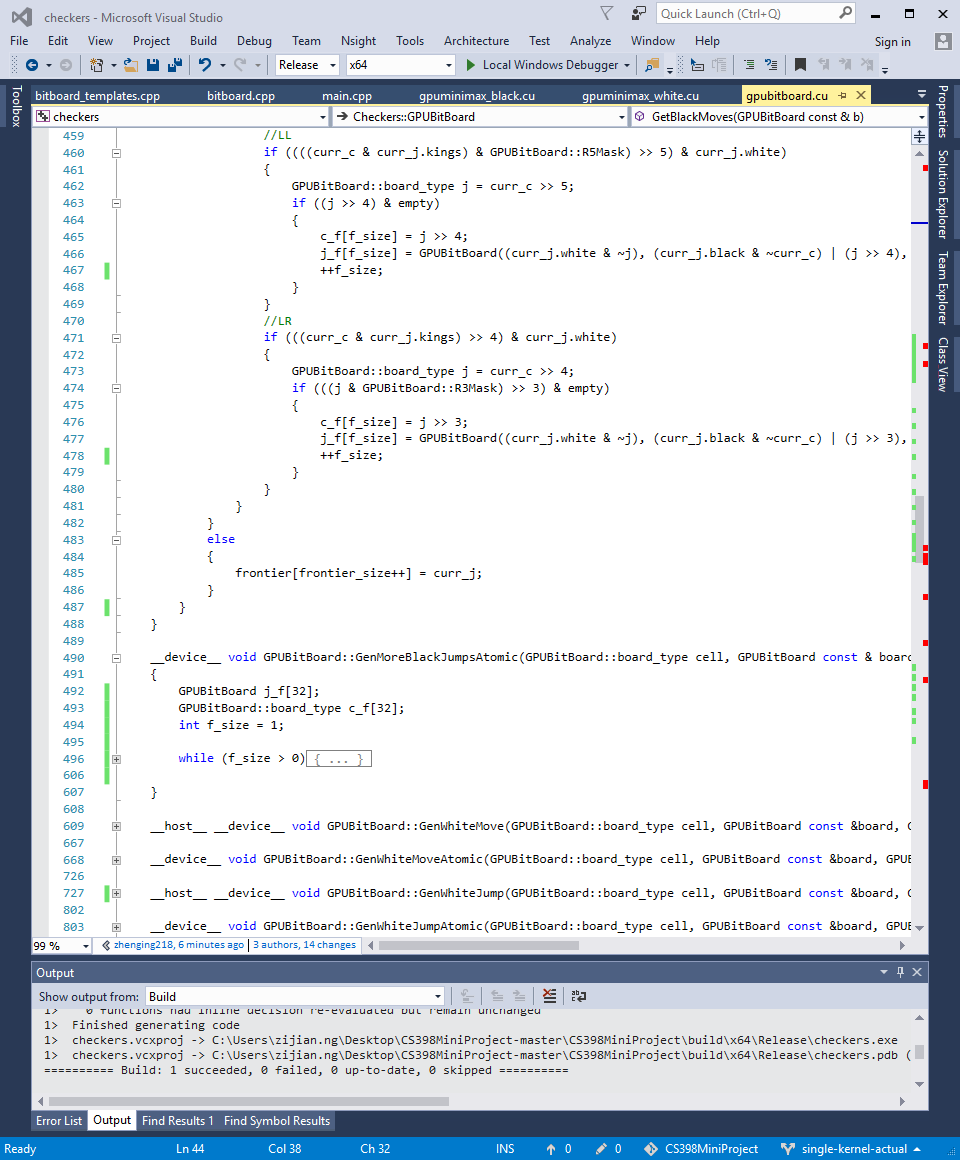


Figure 5m. Last of the implementation.

# Getting the best move set available

## 6.1. Using minimax algorithm

We are using minimax algorithm to determine the best moves that is derived from section 4 in both CPU and GPU. What minimax does is to get the best possible moves from a series of nodes that might have been created from each tree.



Figure 5a. An illustration of minimax algorithm

How minimax works, it perform an expansion of the main node to the terminal node or the desired depth and works its way up the node while comparing with its siblings’ nodes applying the condition of who has min or max value on that specific depth and it keeps comparing until it has reached the top node.

## Using CPU minimax

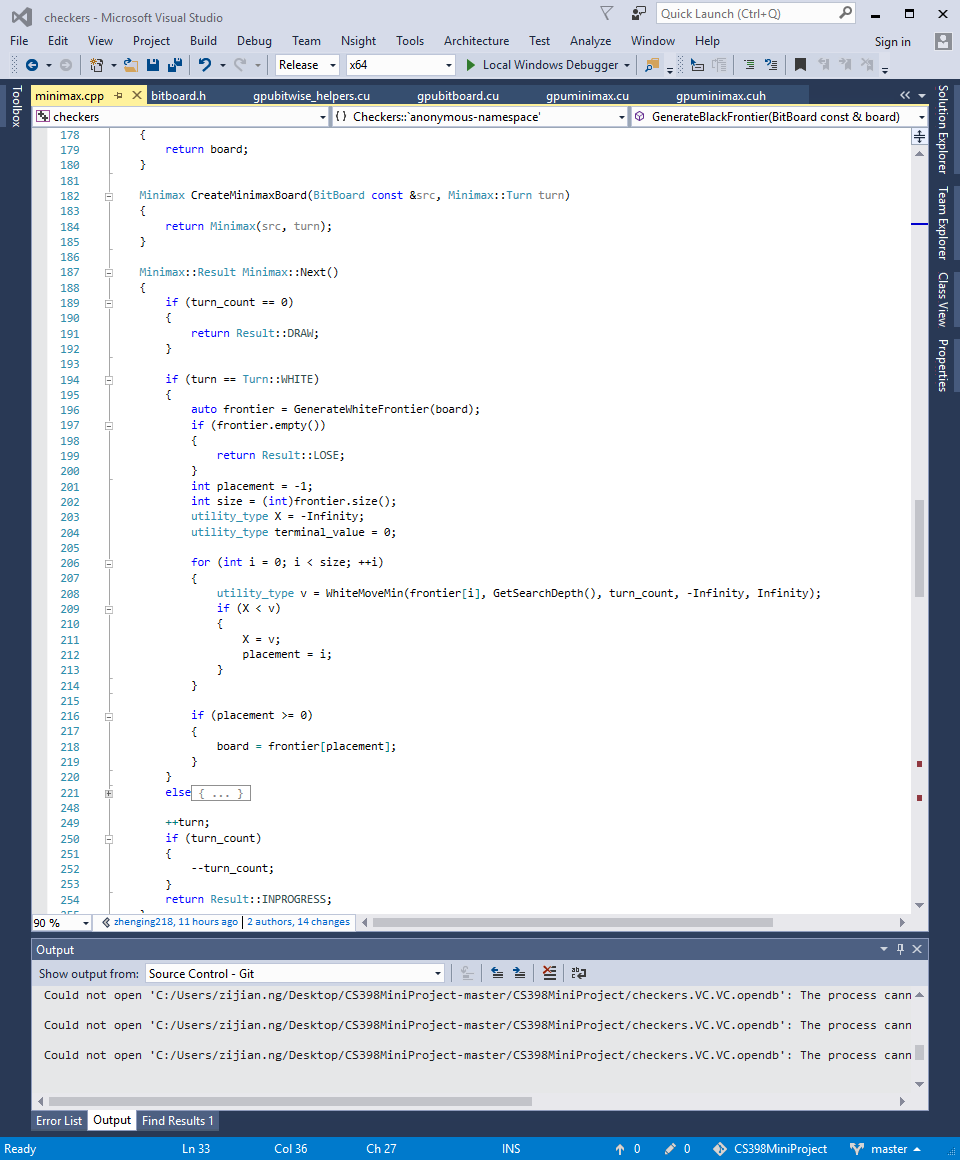
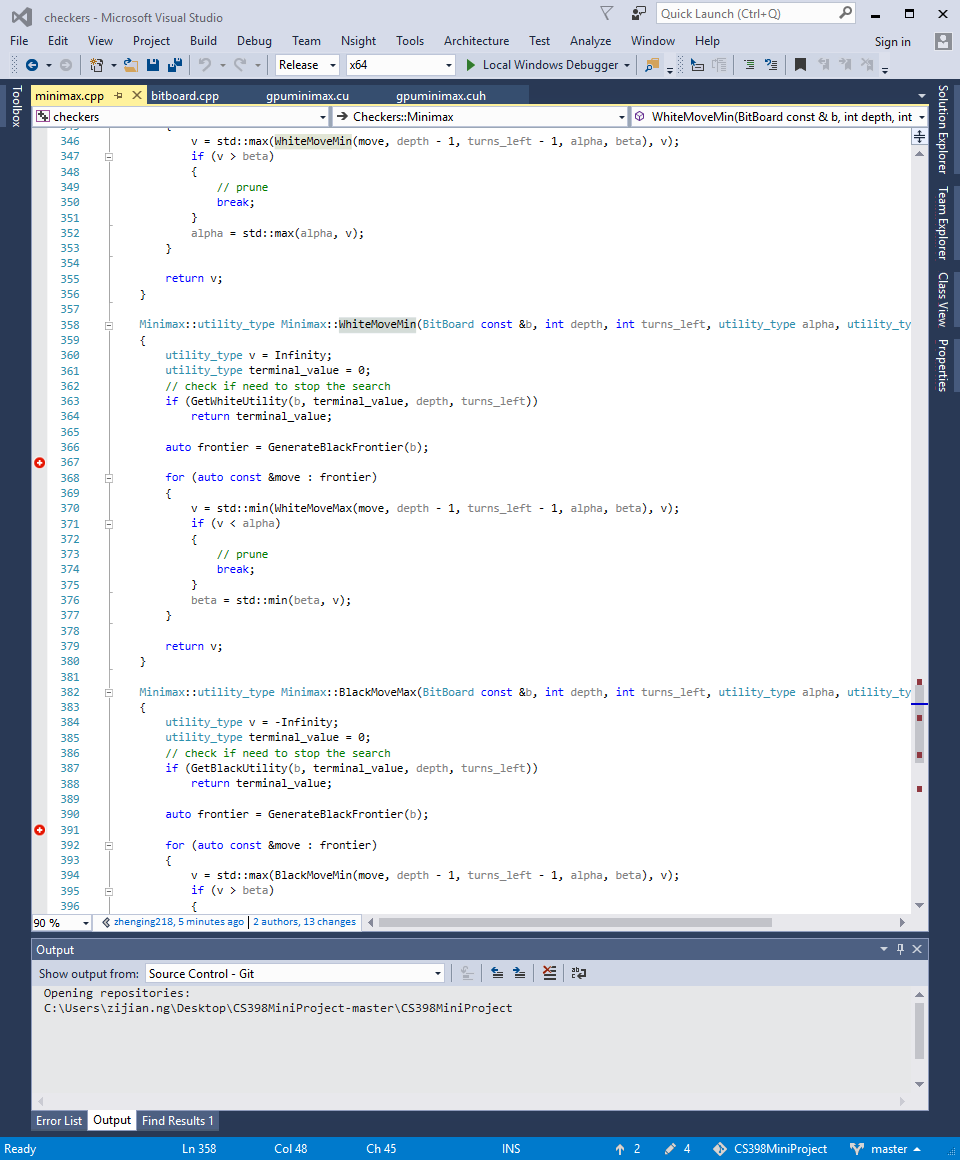


Figure 5b. Main function of minimax

In this main function of minimax is where it calls for the creation of possible moves in Section 5 and evaluates them to see which of the following frontier that was generated, has the best value out of all the frontier available. It commences by calling the min function of each corresponding function of the pieces. After the nodes are explored, it then makes a comparison of the best utility of the nodes and get the best move set.

## Min function of minimax

Figure 5c. A min function of minimax algorithm

In this min function of the minimax, we start by initializing the value as infinity and slowly, it will be filled with values as it traverse down.by calling the max function until it has reached the specified depth. The values that is derived from these function is then passed back to the previous function to be compared for a better frontier to explore or there isn’t a need to expanding into the siblings frontier

## Max function of minimax

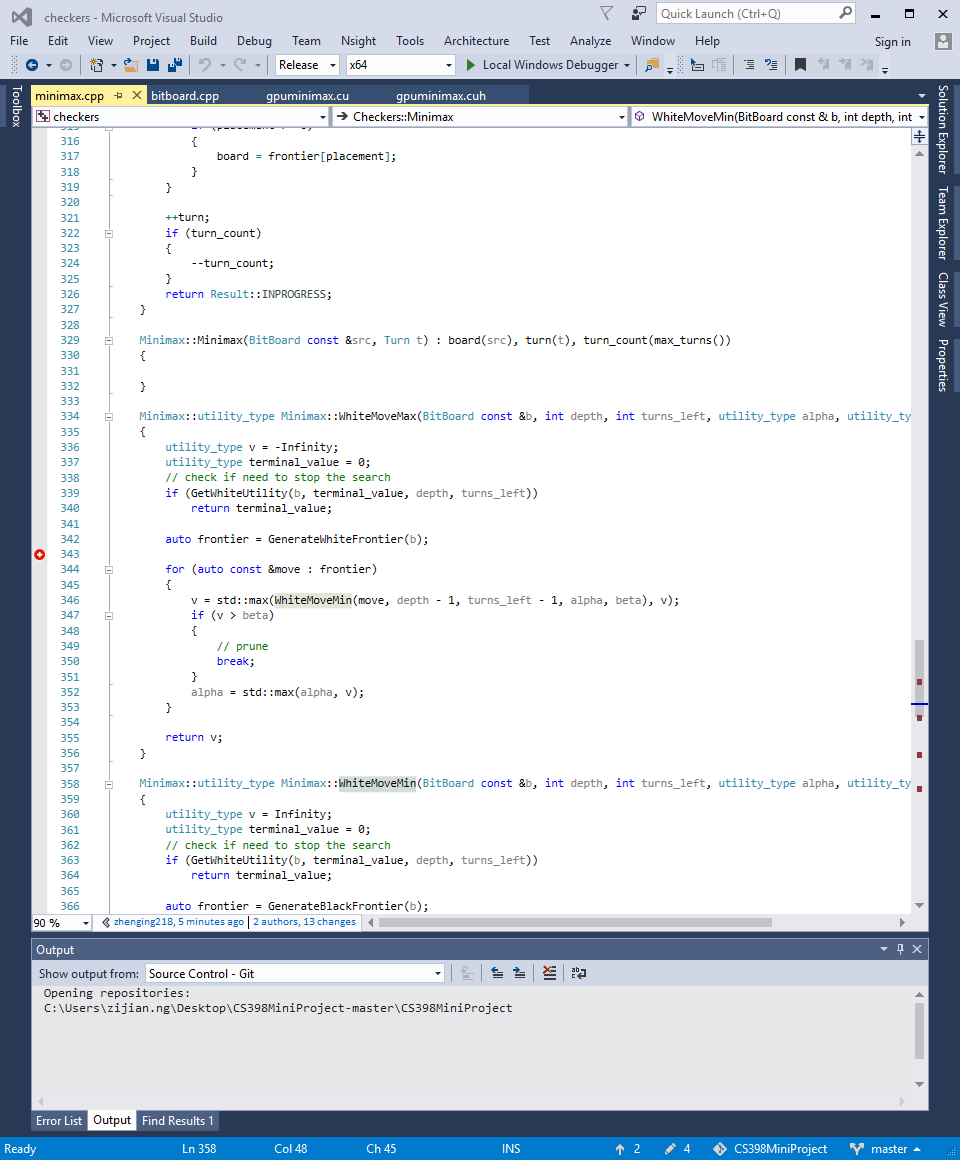


Figure 5d. A max function of minimax algorithm

## Using GPU minimax

To implement a parallelized version of the minimax algorithm, we look towards a particular method called the Principal-Variation Splitting (PV-S).

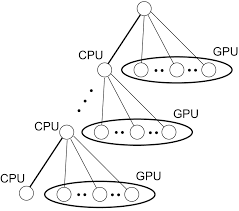


Figure 5e. An illustration of PV-split

The rationale of the PV-S is that most of the information needed for the pruning of nodes are determined by the depth-first traversal of the left-most game sub-trees i.e. the principle variation. As such, the left most node is computed on the CPU trivially and then have its alpha/beta values passed onto the sibling nodes of the explored child node. These sibling nodes will be sent to the GPU and processed in parallel.

We try to maximize our utilization of the GPU by sending the sibling nodes (frontier) to the GPU, which runs the corresponding minimax node in the next depth. With the bitboard being 8x8 in dimension, we can utilize the entire block worth of 32 threads to generate each frontier node’s corresponding frontier.

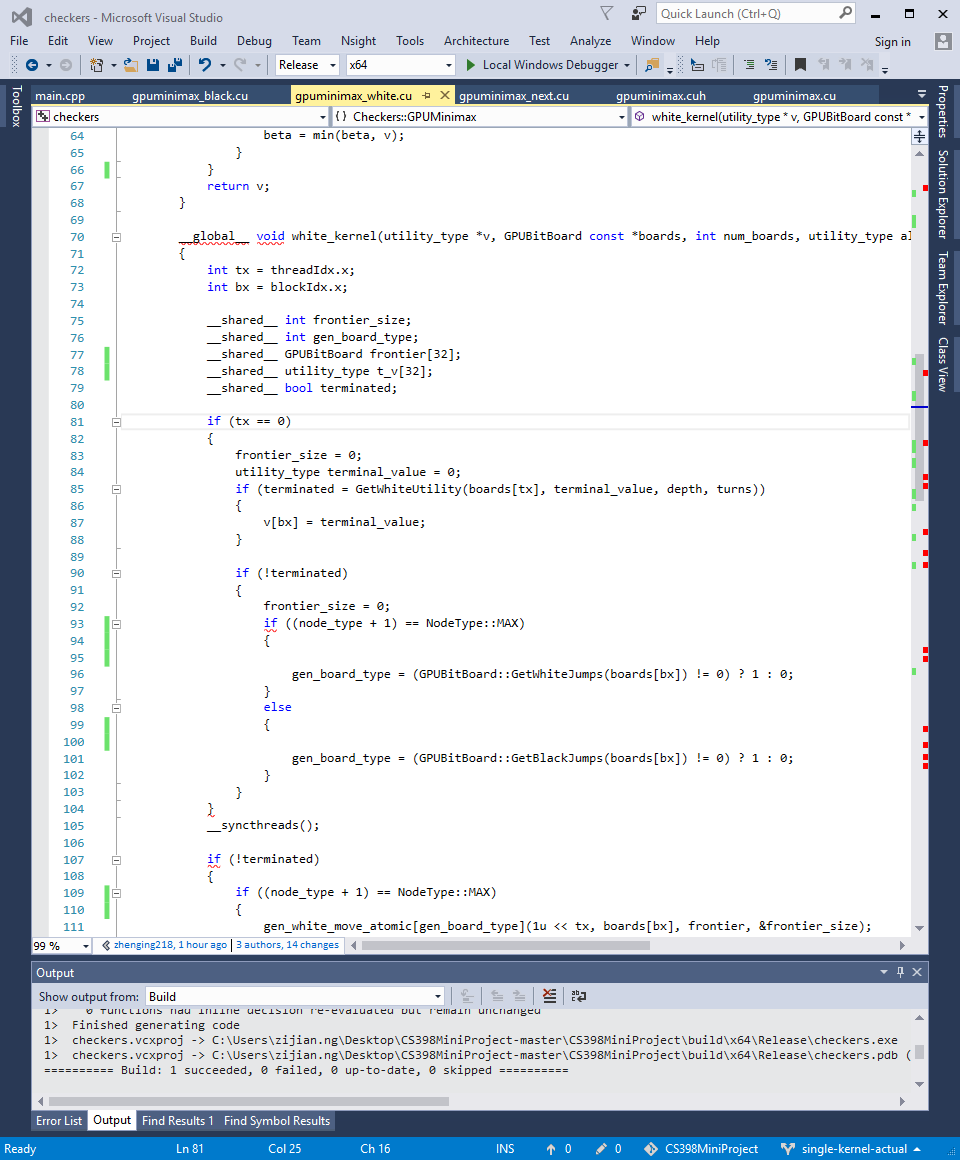


Figure 5f. An implementation of getting the best moves in GPU

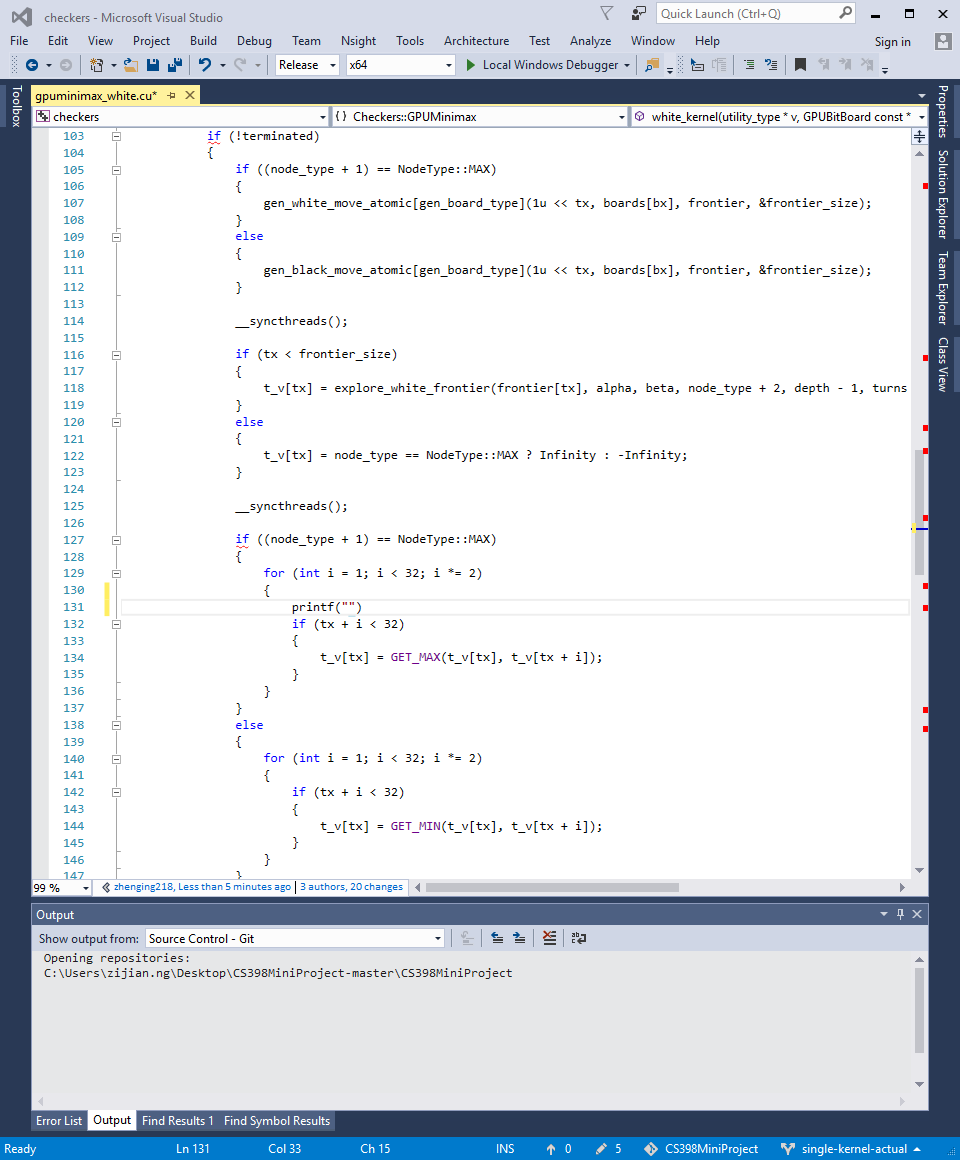
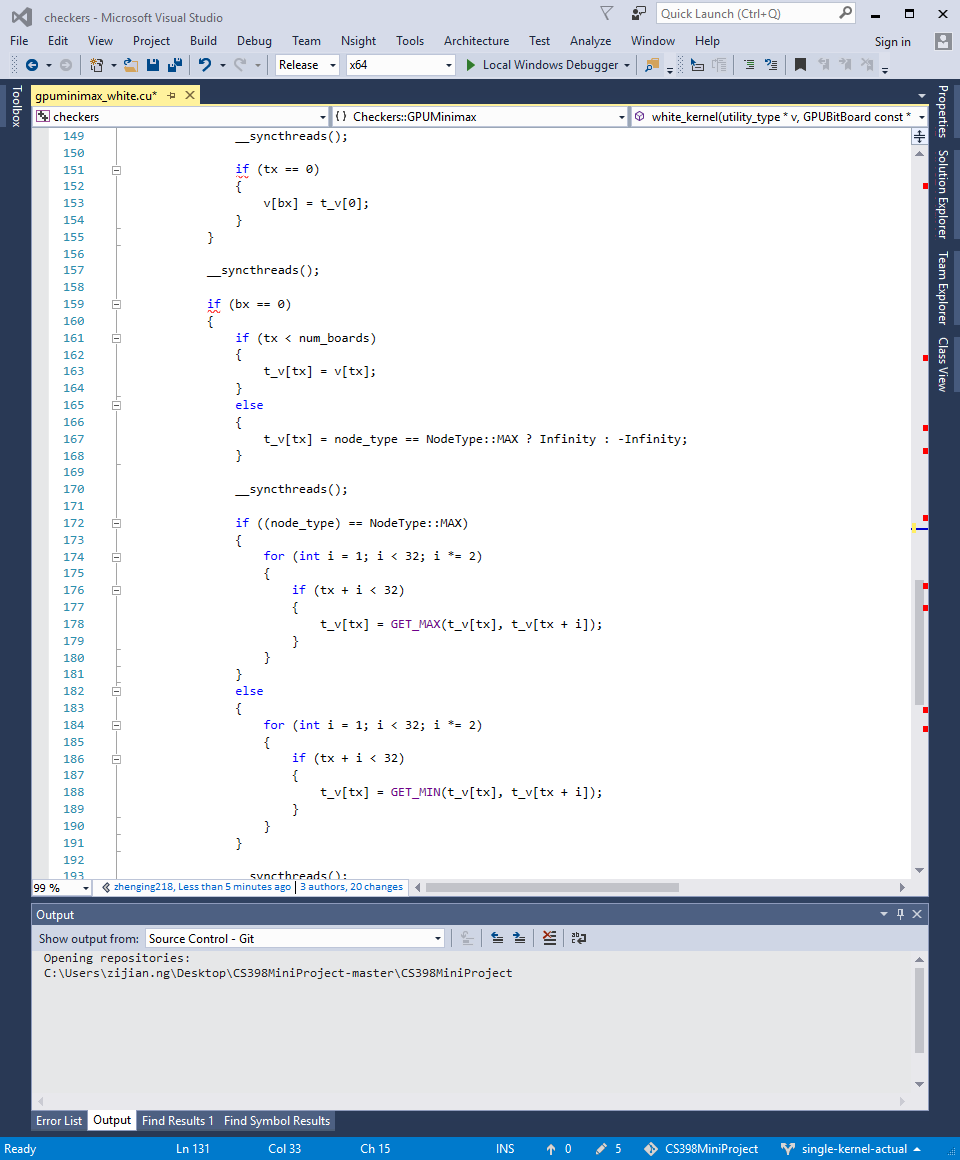


Figure 5f. A continual implementation of getting the best moves in GPU



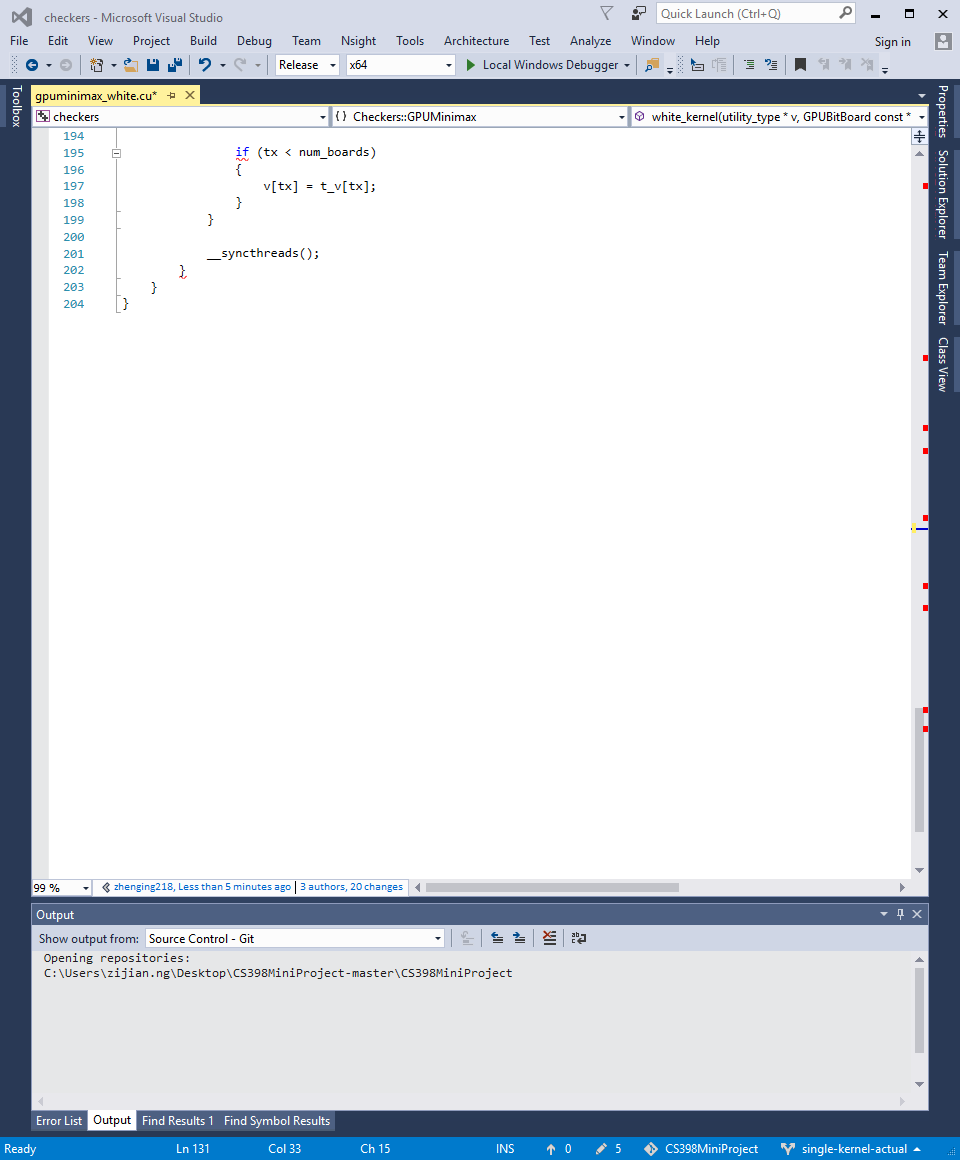


Figure 5g. A continual implementation of getting the best moves in GPU

We then make use of the threads in the block to sequentially explore the new frontier nodes, which will run similarly to the CPU version of the minimax. In this implementation of GPU, we are using the reduction method to get the best max and min values from the nodes that we have explored.

As such, a typical decision tree execution can look like this:

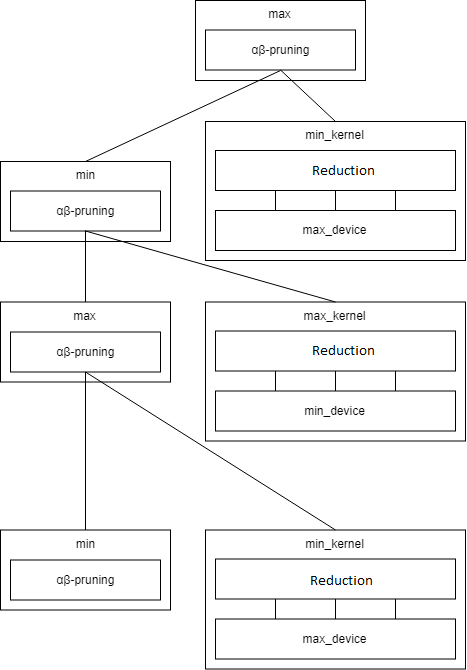


Figure 5h. An illustration of how it works

The αβ-pruning process is done once per expanded depth in this case. Therefore, the GPU would perform the reduction of the frontiers of each child node, and then perform the remaining of the child nodes sent from the CPU to the kernel. We will then have just one utility value remaining that the CPU can trivially perform the decision computation.

## Efficiency Concerns

As discussed in the previous section, a main concern with efficiency is with the expanding of grand-child nodes on the CPU, which is done serially per thread in the block. Although we have achieved some parallelism on the frontier generation on the child node, we are now trading off parallelism for a simpler implementation.

Another concern with regards to efficiency is the size of the data the GPU is working with. On average, most nodes will expand into a frontier of around 4 – 12 child nodes, which is a small number. Although we are exploring each child node in parallel, the overhead of moving small amounts of data to the GPU will most likely not outweigh the parallelism achieved, especially with the serial nature of grand-children node exploration.

## Attempted and Implemented Improvements

Dynamic Parallelism was considered to alleviate the serial nature of grand-children node exploration. However, it also comes with its own limitations and overhead. Firstly, dynamic parallelism is limited by the synchronization depth of the GPU, which means that an implementation of minimax will be restricted to a certain depth of searching through the decision tree. Although the synchronization depth can be increased, the overhead cost of the increased depth will also consume more memory and computation resources, which may not offer much of a trade-off, if any at all.

However, we did implement an increased stack size for the GPU execution. As each expansion of the grand-children nodes require more memory on the stack, we were initially unable to run the minimax on the GPU due to stack overflow. We then attempted to use heap memory through cudaMalloc, but this method brings with it latency during global memory access, which is now present every time a frontier node is expanded further. As such, we opted to increase the stack size for device-side execution using cudaDeviceSetLimit.

Lastly, the use of the bit-board representation allowed us to not increase the stack limit by too much, although it was still a concern due to the recursive nature of the decision tree searching.

# Benchmarking CPU vs GPU (Only for

Below are the comparison between CPU and GPU

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Depth | CPU(in ms) | | GPU(in ms) | |
| Slowest | Fastest | Slowest | Fastest |
| 1 | 0.015832 | 0.001173 | 2.31432 | 0.000293 |
| 5 | 9.70037 | 0.010264 | 101.005 | 0.000587 |
| 8 | 1636.97 | 0.003812 | 24909 | 0.001466 |
| 10 | - | - | - | - |

# Findings

As expected, the running of the minimax algorithm on the GPU produced significantly slower results as compared to the sequential CPU version, which is most likely due to the small data size and serial nature of the exploration of grand-children nodes. The efficiency of the GPU version of the minimax may potentially improve if used on larger data such as bigger boards, but it may incur large amounts of data usage. It will also require a different representation for each bigger board size, since the bit board implementation is specifically tailored to the 8x8 checker board.

As we initially opted for the bit-board representation with memory constraints and the ease of bit-wise arithmetic on the computation in mind, we believe that this particular optimization increases the timing differences between CPU and GPU, since memory space and access is now a negligible factor to both versions. As such, the CPU now has the advantage of locality, while the GPU is hampered by memory transfer from host to device as well as under-utilization of its parallelism capabilities.

We also note that the GPU minimax seem to be non-deterministic with regards to the move generation as well, which we do not seem to be able to pinpoint the problem. We suspect that this could be due to the depth-first-search nature of the minimax, which causes a batch-based execution on the GPU to yield non-deterministic results.

# Conclusion

Perhaps the use of the minimax on conventional checkers AI will not yield positive responses with regards to efficiency due to the small board size. As we focused mostly on the use of a bit-board representation of the checkers board, there could be potentially unforeseen speed up of the execution of the minimax on larger boards, allowing better trade-off between host-to-device memory transfer and the parallelism of execution on the GPU.