On-Chip Neural Chess Analyzer (ONe-ChAn)

Haihan Wu and Muhammad Abdullah
abd880@mit.edu and haihanwu@mit.edu

I. Abstract

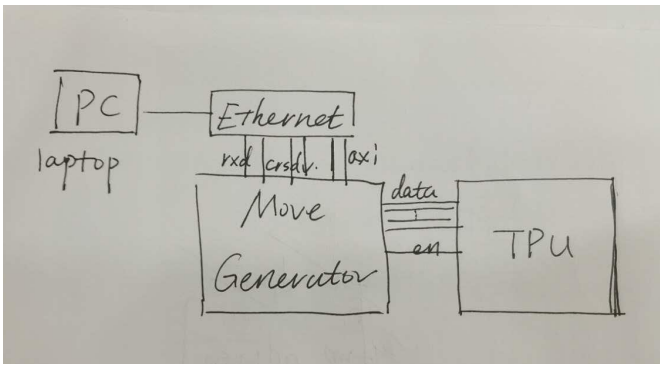
Our Project is going to be building a TPU that supplements a chess engine. A chess engine is made of two parts:

1. Position evaluator: Takes a board position and returns a numerical estimate of the advantage each side has, we will use a CNN on the TPU for this.
2. Move generator; Takes a board position and plays the possible moves, this is done at a shallow depth (3 or 4 moves into the future). Then we send the board to the position evaluator to find the value of playing that sequence of moves, so we can find the move with the highest advantage and return that.

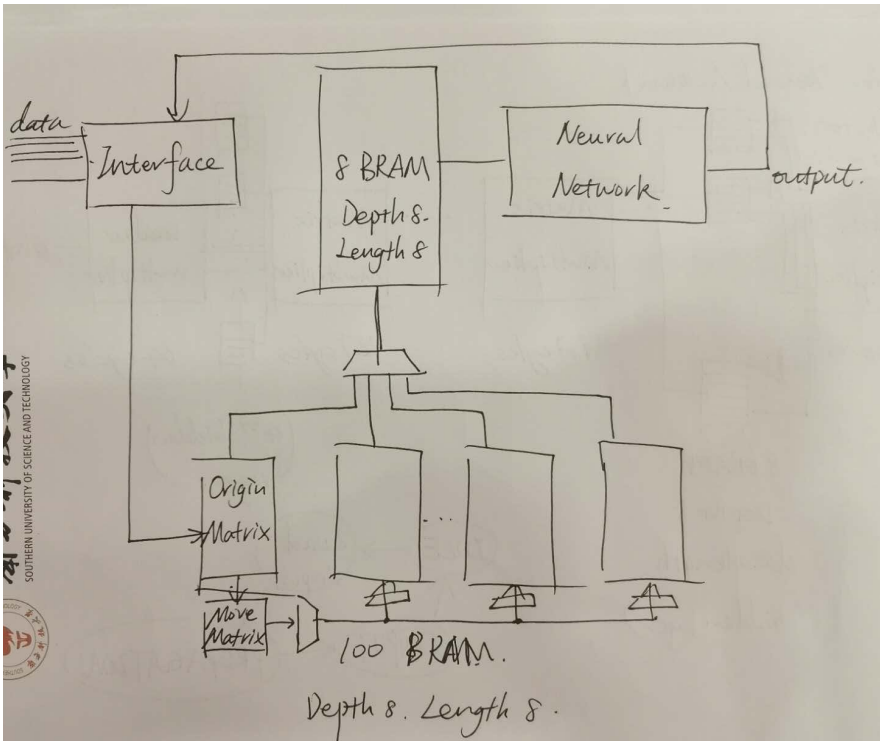
We will divide the workload on two FPGAs along the above lines, the Position evaluator will be a more general TPU. We train a CNN (or another matrix-based ML algorithm) locally and upload the weights and biases to the TPU. Muhammad Abdullah is planning to train the ML algorithm and implement an alpha-beta pruned tree traversal technique for the move generator, and Haihan Wu will implement the position evaluator (TPU) and the interface between two FPGAs.

II. Block Diagram

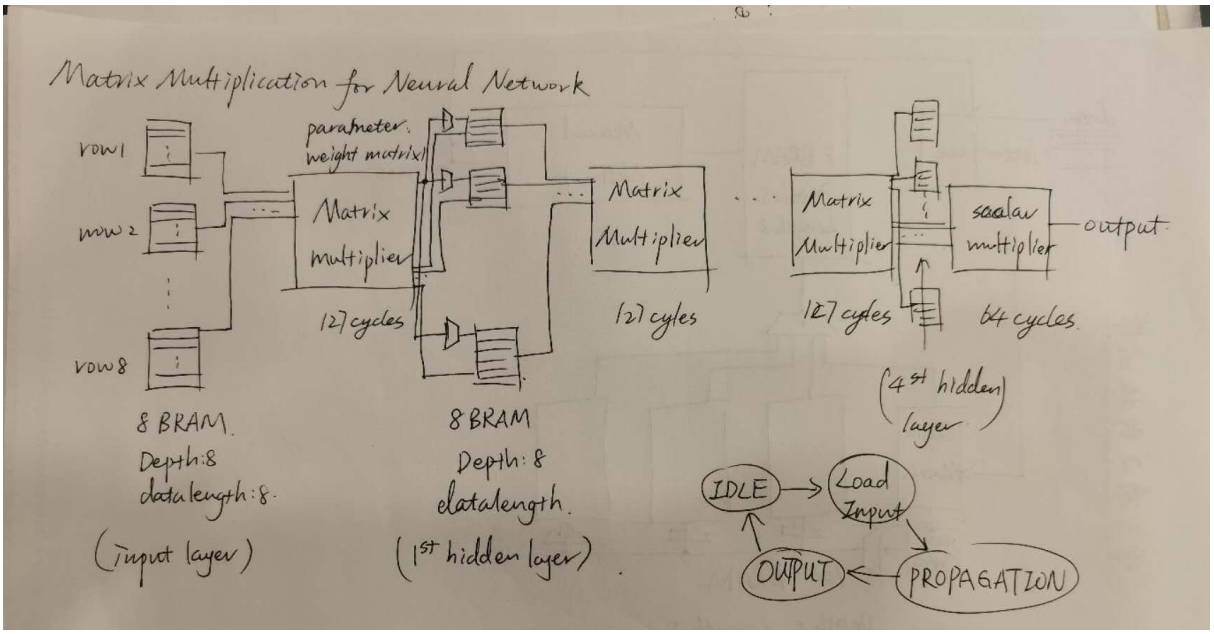
A. Overview of the two FPGAs



B. TPU (position evaluator)



C. Neural network



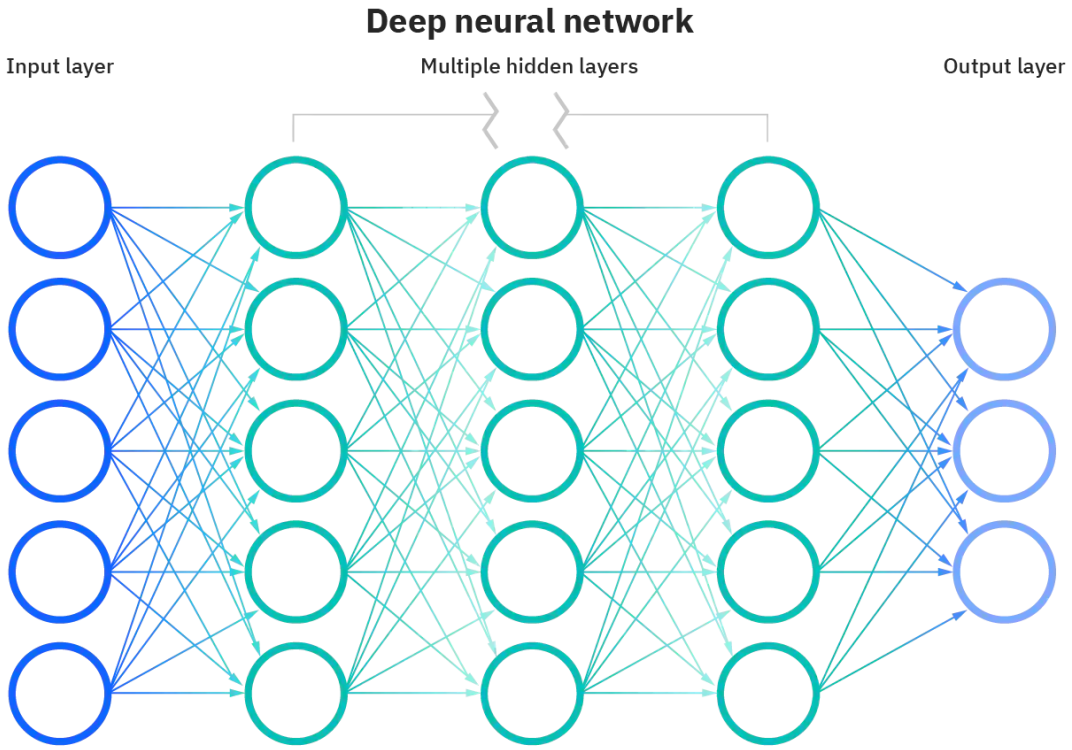
D. Move generator (Provided in title 4)

III. Convolution Neural Network and TPU

TPU is a dedicated processor for matrix operation, especially matrix multiplications. Convolution neural network(CNN) consists of input layers, several hidden layers, and output layers, each layer is computed by the weighted sum of the previous layers. If the chessboard message is popped into the network, the output will reveal the advantages. The sign of output reveals the advantageous side, and the value indicates how advantageous it is. For

example, if the output is -8, it's more favorable for black; if the output is +3, it's more favorable for white.

Our neural network model consists of 1 input layer and 4 hidden layers with 64 neurons each, 1 output layer with 1 output of bits. All the neurons' values are signed numbers. To get a precise prediction, we need to train the network so that we can obtain the optimal combination of weights. Muhammad will train the network in python with the backpropagation method. The weights are matrices between layers.



The Chess has an 8-times-8 board, two players, and 6 types of chess pieces(The Pawn, The Bishop, The Knight, The Rook, The Queen, and The King). Since the neural network only accepts binary input on hardware, the chess information should be encoded in *One-hot encoding*. We will use an 8-by-8 matrix with 8-bit elements to contain information on the board. The row and column of the matrix represent a position on the board, and the element reveals the player and chess piece. All the encodings are listed as follows.

8-bit element	meaning
MSB	1 for black
6th-bit	1 for white
5th-bit	The Pawn
4th-bit	The Bishop

3rd-bit	The Knight
2nd-bit	The Rook
1st-bit	The Queen
LSB	The King

And 4 64-by-64 matrices, 1 64-by-1 weight matrix, and 5 biases are required. Weights are 8-bit signed float binary between -1 to 1, and biases are 8-bit signed binary.

The main operation is matrix multiplication, so in TPU, we utilize a high-efficient matrix multiplication method to get the inner product of the corresponding vectors. The move generator will send the board information and possible move of chess pieces. Board information is a signed-8-bit 8-by-8 matrix, and chosen piece of information and the moving option will be encoded properly to generate the board after moving at TPU. Those messages are transmitted 1 byte each cycle. Move generator will send matrix elements row by row.

Byte number	1-64	65	66	67-76
Content	Board matrix	Chosen piece	Position	Possible move

bit number of move	1-4	5-8
Content	Vertical move	Horizontal move

The TPU will instant Matrix_multiplier module, which performs 64-by-64 matrix multiplying 64-by-64 or 64-by-1 matrix. Once the data is received at the interface, it will be loaded in BRAM with depth 64 and start generating moved matrices from moving information, and they will also be stored in other BRAMs. Then Matrix_multiplier will pop in value from BRAM into 8 BRAMs with depth 8 and start performing matrix multiplication 5 times.

IV. Move Generator and PC interaction

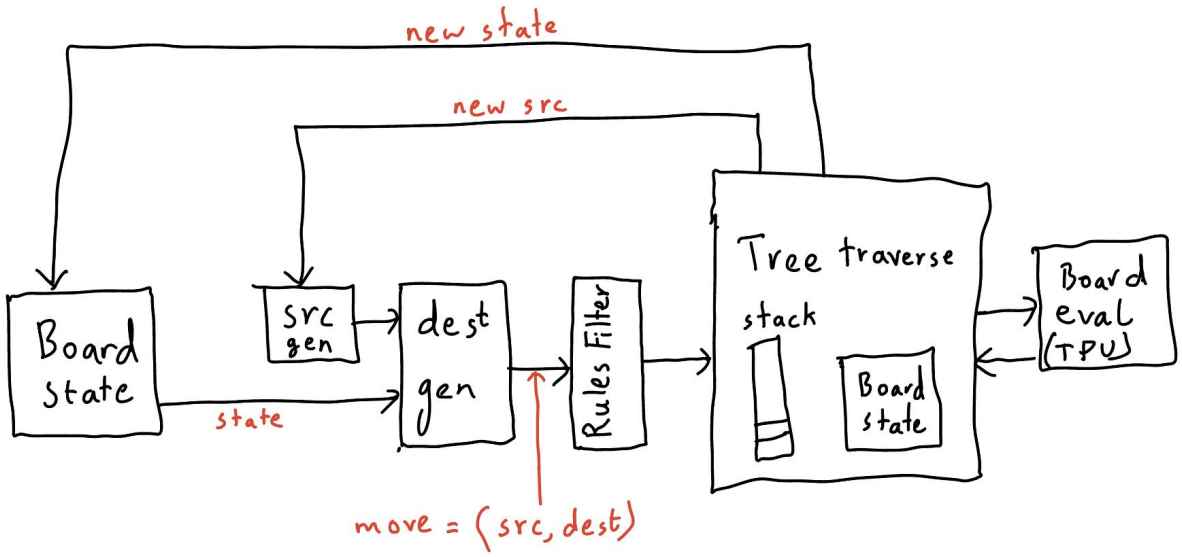
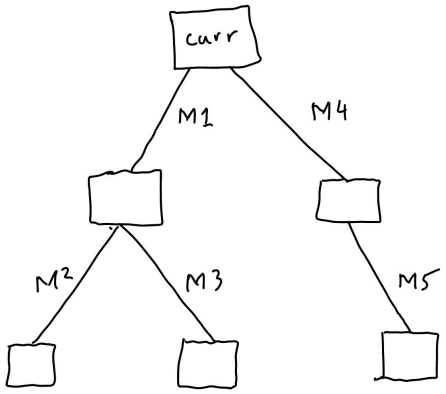


Figure 1: Move Generator

We use the Universal Chess Interface(UCI) to play around with moves on a chess board. This notation takes the form *source_col,source_row,dest_col,dest_row* where the source and destination are the movements of the piece, eg **e4e5.** This also allows us the possibility to use a laptop to interface with FPGA and play chess using the UCI format. So we can fight other game engines and players and determine the ELO (rating) of our engine. Initially, we will use switches to communicate which move the opponent played (12 switches are sufficient for most UCI moves).

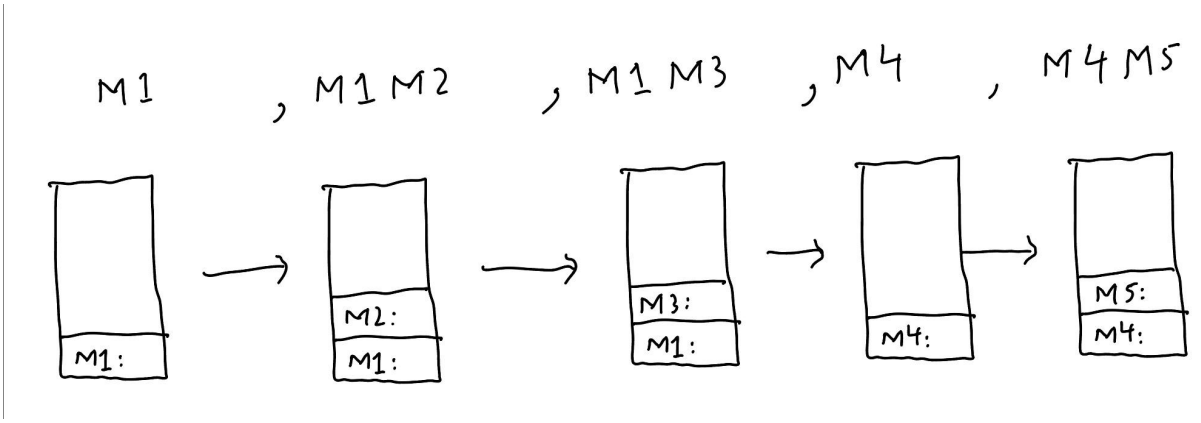
Figure 1 shows the Move Generator Block Diagram. Given a chess board, we generate a set of possible moves. These moves are iterated over by the *dest_gen* module which outputs them in a UCI-like format: *(src, dest).* This module's block diagram is in Figure 2. The moves are then filtered through the *rules_filter* module which enforces whether the current move is allowed or not, it takes care of checks and piece pinning. We can iteratively add more rules to approach the entire ruleset of chess, like en-passant and castling.



Once a valid move has been generated, we pass it into the *tree_traversal* module which performs a complicated brute force. The idea is; from a starting position, we can have a possible set of future moves the player can play (*M1* and *M4* in the diagram). And in response to those, the opponent can play another set of moves (*M2*, *M3,* or *M5*). So we can imagine the future moves as the tree shown above.

We can use the stack as shown in the block diagram to perform a depth-first search of the moves. We take the current board and send it to the TPU along with the stack to perform moves on the TPU side, this decreases the amount of data we need to transfer FPGA-to-FPGA. We also store some metadata on the stack (like if a piece was taken, what

type was it) which would allow us to backtrack by popping off from the stack and undoing the move.



The chess engine will keep track of the **current best** sequence of moves that result in the highest advantage at the leaves. When it has gone the depth it needs to (preliminarily we will go 4 layers deep, but the goal is to go 8), it returns the best move.

So the tree traversal module would take care of the following points:

1. Use the stack to keep track of the moves played
2. Update the *board_state* and *src_gen* to produce the next move it looks for.
3. Update its own internal board and forward the board and all moves to the TPU
4. Keep track of the current best move.

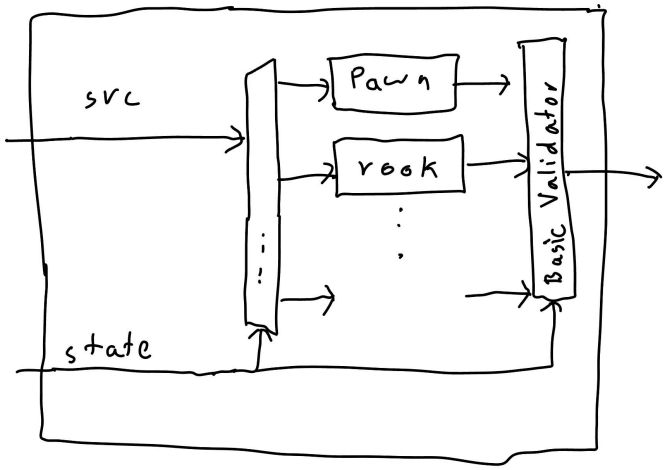


Figure 2: *dest_gen* block diagram, we generate the moves for the piece on the *src* position and output through a basic move validator which checks for board edges and friendly pieces blocking the move.

V. Timeline

WEEK	Haihan WU	Muhammad Abdullah
Oct 28 ~ Nov 6	High-efficient 64-by-64 matrix multiplication realization	Move generator Detailed design and simulation

Nov 7 ~ Nov 13	Interface building	
Nov 14 ~ Nov 20	Ethernet interface with PC	Training NN and get optimal weights and biases
Nov 21 ~ Nov 23	Test chess engine with online agent	
Nov 23	**Preliminary Report Submission**	
Nov 24 ~ Nov 28	Increase efficiency of the engine	