# *From C to Blokus Duo with LegUp High-Level Synthesis*

*Jiu Cheng Cai, Ruolong Lian, Mengyao Wang, Andrew Canis, Jongsok Choi, Blair Fort, Eric Hart*[*], Emily Miao, Yanyan Zhang, Nazanin Calagar, Stephen Brown, Jason Anderson*
Department of Electrical and Computer Engineering, [*]Department of Mathematics
University of Toronto, Toronto, Canada
legup@eecg.toronto.edu

***Abstract*—We apply high-level synthesis (HLS) to generate Blokus Duo game-playing hardware for the FPT 2013 Design Competition [3]. Our design, written in C, is synthesized using the LegUp open-source HLS tool to Verilog, then subsequently mapped using vendor tools to an Altera Cyclone IV FPGA on DE2 board. Our software implementation is designed to be amenable to high-level synthesis, and includes a custom stack implementation, uses only integer arithmetic, and employs the use of bitwise logical operations to improve overall computational performance. The underlying AI decision making is based on alpha-beta pruning [2]. The performance of our synthesizable solution is gauged by playing against the Pentobi [8] – a "known good" C++ software implementation.**

## I. Introduction

High-level synthesis (HLS) is on the cusp of becoming a mainstream design methodology for field-programmable gate arrays (FPGAs). HLS raises the level of abstraction for hardware design nearer to that of software design, making it attractive to two constituencies: 1) hardware engineers as a means of reducing design time and shortening time-to-market, and 2) software engineers as a way to garner the speed and energy benefits of hardware – benefits that until recently, could only be leveraged using specialized hardware-design expertise. In this work, we apply the LegUp open-source HLS framework [1] to generate game-playing hardware for Blokus Duo.

HLS tools have not yet reached the level of maturity where they can accept *any* program as input, and produce high-quality hardware. Rather, to produce good results, today's HLS tools generally require that the high-level code be written in a specific style, perhaps with additional constraints in side files or within the code as pragmas. For example, a recent work [4] described how to tune C code to produce good results for Xilinx's Vivado HLS tool. The quality of hardware produced by our HLS tool, LegUp, is likewise sensitive to the style of C received as input – an interesting issue to which we devote a significant part of this paper.

Our Blokus game-playing engine uses alpha-beta pruning [2] to explore the solution space in an efficient manner. We incorporate a number of features to tailor the implementation to the HLS tool and the target FPGA technology. Namely, we represent the game state using a minimum number of bits (necessary given the limited amount of on-FPGA memory). We use only integer arithmetic, and make heavy use of bitwise logical operations to check move legality. Although recursion is used in standard tree traversal algorithms, the LegUp tool is unable to synthesize hardware for recursive software, necessitating that we design and implement a custom stack. Our implementation is parameterized to operate with any number of game-tree levels (at the expense of run-time), and with a variety of different heuristics we explored to "score" a game state. The implementation has been synthesized and tested on the Altera DE2-115 board [5].

With our competition entry, we intend to "challenge" both human-designed Blokus hardware implementations (an exciting "human-versus-machine" gaming match-up!), and also challenge implementations produced by other competing HLS tools. The rest of the paper is organized as follows: Section II briefly reviews the rules of Blokus Duo. Section III describes our implementation. Section IV discusses various heuristics we investigated to assess the quality of moves. An experimental assessment of heuristics is provided in Section V. Section VI discusses approaches to prune the solution space. Section VII gives the current status of our implementation comparing with a C++ software Blokus implementation (Pentobi [8]). Conclusions are offered in Section VIII.

## II. Blokus Duo

Blokus Duo is a 2-player game played on a board with 196 squares arranged in a 14×14 grid. Each player begins with 21 tiles, ranging in size from 1 to 5 squares, and each having a unique shape. A tile can be placed on the board in any orientation, by means of flipping/rotating the tile. The objective of the game is to place tiles on as many squares of the board as possible, and the game ends when neither player has any tile remaining that can be placed legally on the board. A "move" consists of placing a tile on the board. Players take turns and generally may only make one move per turn. In the case that one of the two players has no remaining valid moves, the other player can continue to place tiles until he/she has no valid moves remaining. A move (tile placement) is considered valid if three conditions are satisfied:

1. The tile does not overlap with existing pieces on board.
2. At least one of the tile's corners is directly adjacent to a corner of one of the player's already-placed tiles.
3. None of the edges of the tile are in contact with any of the edges of tiles previously played by the player on the board.

Rule #2 is not applied for the first move made by each player. In fact, the first move by each player must overlap a specially designated square on the board.

978-1-4799-2198-0/13/$31.00 ©2013 IEEE

## III. IMPLEMENTATION

We discuss our data structure to implement the game state, the search algorithm used, and the approach to prune the search space. In addition, we mention how we adapted our design to be more efficient in terms of run-time and area on the FPGA.

### A. Game Representation Data Structure

The current game state is represented in a `struct` type called *GameState*. *GameState* has fields representing the current board, the tiles remaining for each player, which player's turn it is, and the turn number. Each instance of a *GameState* is 1004 bytes in size.

We represent the board using two arrays. Each array contains 16 elements of 16-bit variables. Array elements 0 and 15, as well as bits 0 and 15 in each element represent the "out of bounds" region of the board. Within the 16×16 bit matrix, a 1 represents an occupied space and 0 represents an empty space. The two arrays are named *our_board* and *opp_board*, where *our_board* contains a representation of the board with only the tiles our algorithm has placed up to the current state, and *opp_board* contains the opposite: only the opponent's tiles. To represent the tiles themselves, we use a 5-element array consisting of 16-bit variables; the reason for this is elaborated upon below. A Boolean variable tracks whether it is our turn, or the opponent's turn.

### B. Search Algorithm

The search algorithm used in our design is the *minimax search algorithm* [6]. The algorithm uses a tree representation, whose root node represents the current game state. The tree's depth corresponds to the number of future moves that the algorithm is capable of considering in deciding its move. We define the depth limit by the MAX_LEVEL constant. Each level of the tree represents all move possibilities for a particular turn. Such moves are considered for either us, or the opponent, depending on whose turn is at the specific level. The root node (level 1 of the tree) always represents our initial state after the opponent's move, and all nodes at the second level of the tree represent game states after we have played a legal move in response to the opponent's. A path from the root to a leaf node represents a particular potential move sequence, commencing from the current board state.

For each node in the tree, there is a *score* variable, which is initially either negative infinity (-1,000,000) or positive infinity (1,000,000), depending on whether it is our turn or the opponent's, respectively. The tree is traversed and on reaching a leaf node, score is calculated by set of heuristics (described below) and is returned one level up the tree – to the node's parent. The algorithm maximizes score for nodes corresponding to our turn, and minimizes it for nodes corresponding to the opponent's turn. Therefore, the calculated score for a node is compared with its parent's score and either the maximum or minimum is selected depending on whose turn the parent represents. The traversal and scoring process continues until all immediate children of the root node have been evaluated. The child with the highest score reflects the "best move" for us to take.

Intuitively, the minimax search algorithm looks ahead into the future to predict the set of likely moves and aims to reach the best possible state for us. Essentially, the search algorithm anticipates the opponent's probable moves and seeks to minimize the potential damage of the opponent's future moves, while at the same time maximizing the benefits of our own moves.

It is important to note that the current implementation of *LegUp* does not support dynamic memory (`malloc`/`free`). Also, as the depth of the search increases, the number of nodes in the tree grows exponentially, hence memory usage would grow as well. Taking these factors into account, we implemented the traversal algorithm with depth-first search, which is more memory efficient than breadth-first search. Since LegUp does not support recursion, we use a stack (with size *MAX_LEVEL*) to hold the current path being explored in the search tree. This way, the memory usage is limited to a small constant multiple of the size of the game state representation. Regarding speed, the minimax algorithm computation time peaks during the initial set of turns, decreasing drastically as the game progresses. Thus, we always begin by playing either the 'X' or the 'F' tile, and during the initial 7 turns, we consider solely the 5-square tiles (plus the 'Z4' tile) for moves.

### C. Alpha-Beta Pruning

As the depth of search increases, the number of nodes to be traversed increases exponentially. Pruning is a necessity to reduce the search space. We use *alpha-beta pruning* [2] to prune away portions of the tree that are provably non-optimal. Alpha-beta pruning allows us to prune away certain children of a node (and thereby the sub-trees rooted at those children), without affecting optimality. Consider, for example, a max node *n* with a current score of 5, and let *m* be a child of *n*. As described above, *m* must be a min node. As we traverse the tree, if node *m* ever receives a score less than 5, then it is guaranteed that *m* will not be selected as the "best child" of *n*. This is because for a min node like *m,* scores only decrease. Hence, *m*'s final score, is certain to be less than 5. Consequently, as soon as *m*'s score is reduced below 5, traversal of the sub-tree rooted at *m* can be terminated, without hindering our ability to find the best-possible solution.

### D. HLS Tuning

The discussion above centered primarily on the algorithms and data structures used in our implementation. To tune the algorithm specifically for HLS-generated hardware performance, we emphasize using logical operators, shifts and bitwise operators, while minimizing the use of area-intensive hardware elements such as multiplier operators. For example, as noted above, the game board and the tiles are represented by arrays of 16-bit variables, therefore updating moves on the game board can be done with a bitwise shift and logical `OR` operation of a tile's representation and the board array.

To improve the time consumed in checking the validity of a move, two additional fields are added to the *GameState* `struct`: *our_shifted_board* and *opp_shifted_board*. These fields are 16-element arrays of 16-bit variables – the same size and dimensions as the game board representations (*our_board* and *opp_board*) discussed in Section III-A. The *our_shifted_board* field is populated with the bitwise logical `OR` of *our_board* shifted in all four directions: north, south, east and west. That is, *our_shifted_board* is a logical OR of four shifted versions of *our_board*. And, *opp_shifted_board* is populated in an analogous fashion from *opp_board*.

We use these shifted board fields to ease checking the "edge constraint" and "overlap constraint" for tile placement (see rules #1 and #3 in Section II). Specifically, given a candidate tile position that meets the "corner constraint" (rule #2), we can perform a logical `AND` between the candidate tile's representation and the four-direction-shifted board. If 0 is returned, the candidate position is a valid move. Otherwise, the move is illegal. Move legality checking is thus reduced to primitive shift and logical operations that are natural in the target FPGA hardware. Thus, our data structures offer improved performance and area of the circuit generated by HLS.

## IV. Heuristics

A key aspect of any game-playing algorithm is a way of "scoring" a game state. The scoring mechanism is used in the minimax search algorithm, and alpha-beta pruning approach described above. The following four factors are considered to evaluate a game state: number of squares placed on the board, number of corners adjacent to an opponent's square, "influence area" and "weighted reachability". These factors are considered from both our own, and the opponent's perspective (i.e. the final evaluation of the game state is determined by our score minus the opponent's score). The factors can be weighted with different coefficients to reflect their importance/effectiveness. The first criterion directly corresponds to the game's objective: covering as many board squares as possible. The reason for considering corners adjacent to an opponent's squares is that 1) a corner is a necessity for moves, 2) the opponent cannot block a corner location if the corner location is directly in contact with an opponent's square (by rule #3).

*Influence area* is a concept we borrowed from the *Blockem* software implementation [7]. This metric measures the area around the player's placed tiles on the board, where squares may potentially be placed. The score is based on how much of the area is empty. The more vacant area leads to larger number of move possibilities, thus the game state is deemed as more beneficial to the player. Influence area metric *includes* the number of corners, thus the "corners count" is not directly included as a heuristic scoring factor.

The last heuristic, *weighted reachability*, is computed as follows: we perform a breadth-first search (BFS) of the vacant squares of the game board starting from each vacant square adjacent to a corner. The BFS is performed considering only north/south/east/west adjacency, not diagonal adjacency. Each vacant square visited in the search is assigned a "cost" in proportion to its distance from a (starting) corner square. Vacant squares that cannot be reached (due to opponent blockages) are assigned a high cost. The weighted reachability metric is the cumulative cost of all vacant squares visited in the BFS, and also the costs of vacant unreachable squares. The intuition behind this metric is to gauge the amount of "reachable" board state, with farther-away squares gauged as "less reachable" than closer squares.

**TABLE I.** Evaluation of heuristics.

| Heuristic 1 | Heuristic 2 | Results | | |
|---|---|---|---|---|
| | | Heuristic 1 Wins | Heuristic 2 Wins | Ties |
| Corners Touching | Influence Area | 4 | 12 | 4 |
| Corners Touching | Weighted Reachability | 0 | 20 | 0 |
| Corners Touching | Squares | 6 | 12 | 2 |
| Influence Area | Weighted Reachability | 7 | 13 | 0 |
| Influence Area | Squares | 6 | 14 | 0 |
| Weighted Reachability | Squares | 18 | 2 | 0 |

## V. Heuristic Experimental Results

To measure the effectiveness of different heuristics, we developed a framework to allow our implementation of the Blokus to play against a duplicate of itself but with different heuristics in use in the two versions. In essence, this is akin to running our Blokus implementation vs. a control implementation. The approach allowed us to evaluate the effectiveness of different heuristic weighting styles.

We considered the four heuristics in isolation, facing them off against one another. For each pair of heuristics (e.g. squares vs. influence area), we executed our implementation 20 times; i.e. we played 20 games. To ensure fairness, we alternated which heuristic had the opportunity to play the first tile. Across the 20 games played for each pair of heuristics, we tracked the number of times one heuristic "beat" the other, vs. the two tied. This experiment was conducted with *MAX_LEVEL* set to 4 (a max depth of 4 in the search tree) and alpha-beta pruning active.

Table I shows the experimental results. The results demonstrate that weighted reachability is the most important factor to consider. The number of squares and influence area

appear to be the next most important factors. Our current implementation scores the board state using a composite function of the heuristics, from both our's (contribute positively) and the opponent's (contribute negatively) perspective.

**TABLE II.** RESULTS AGAINST PENTOBI (OUR IMPLEMENTATION PLAYS FIRST HALF OF THE TIME).

| Pentobi Level | Results | | |
|---|---|---|---|
| | Wins | Losses | Ties |
| 1 | 16 | 4 | 0 |
| 2 | 8 | 12 | 0 |
| 3 | 4 | 16 | 0 |

## VI. ADDITIONAL PRUNING METHODS

Our minimax implementation with alpha-beta pruning finds the *optimal* solution for a given scoring function and tree depth. To improve execution time, we have implemented additional pruning methods that are heuristic.

### A. Random Monte Carlo Pruning

We implemented "Monte Carlo"-style pruning of the search tree, where, based on a user-specified percentage, we randomly eliminate certain nodes from the tree traversal. With this approach, we can traverse a *portion* of the tree to a deeper level, while maintaining a reasonable run-time. In essence, we are "sampling" the search tree.

### B. Pruning Offset

In its optimal form, the alpha-beta pruning algorithm only prunes a child node (and its descendants) when it provably cannot deliver the "best" score to its parent. That is, for a parent max node with a score of Q, if one of its child min nodes has a score less than Q, we can safely prune the child and its descendants, as the child's score will only decrease in the minimax algorithm. We implemented a heuristic "offset" to allow further pruning, possibly at the expense of optimality. In our heuristic, the user supplies a non-negative integer offset *T*. Returning to the example, where a parent max node has a score of Q, in the heuristic, we would prune any child min node with a score less than Q+T. With T = 0, the heuristic reduces to standard (optimal) alpha-beta pruning. In an analogous fashion, for a parent min node with a score of Q, we would prune any child max node of the parent if the child had a score more than Q-T.

The rationale for this pruning strategy is the assumption that moves lacking an appreciable difference in scores do not have a significant impact on the game's outcome. By pruning away moves that are "close to" moves already discovered (from the quality perspective), we eliminate more possibilities and reduce computation time, slightly reducing the accuracy of the search algorithm. In our implementation, we selectively apply this approach only at lower levels of the search tree.

## VII. CURRENT STATUS

At the time of writing this paper, when our Blokus implementation is synthesized to hardware through LegUp, the design uses 83,675 Cyclone IV logic cells, 28 DSP blocks, and 238K memory bits (area results include the RS-232 interface that meets the FPT Design Competition requirements). We verified our HLS-generated hardware by having it play against a human player, through RS-232 communication.

To evaluate our algorithm, we developed a framework to allow for automated playing against a well-known Blokus implementation called *Pentobi* (version 1.0, Linux release) [8]. Table II shows the results of our algorithm vs. Pentobi's level 1, level 2 and level 3 algorithm (at the time of writing the paper). As shown, our implementation beats Pentobi level 1 most of the time, is partially successful against level 2, and loses 75% of the time vs. level 3. For these results, we set *MAX_LEVEL* to 4, and use the heuristics mentioned in the previous section, with alpha-beta pruning active.

## VIII. CONCLUSIONS

High-level synthesis (HLS) holds significant promise as a design methodology for FPGAs. However, there remains a significant gap between HLS-produced hardware and human-designed hardware. In this paper, we describe a game-playing Blokus hardware implementation for the FPT 2013 Design Competition that we submit as a candidate to "take on" the human-designed submissions. Our solution has been synthesized using the LegUp open-source HLS tool. Our C code has been written with HLS and FPGA hardware in mind. We make use of bitwise logical and shift operations to check move legality, incur a small memory footprint, and use solely fixed-point integer arithmetic. Further improvements in the AI and run-time of our implementation will be made in the months leading up to the competition.

## REFERENCES


[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. Brown, J. Anderson "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems." In *ACM Trans. On Embedded Computing Systems*, Vol. 13, No. 2, 2013.

[2] D.E. Knuth, R.W. Moore. "An analysis of alpha-beta pruning." *Artificial Intelligence 6, no. 4, pp. 293-326. 1976.*

[3] The 2013 International Conference on Field-Programmable Technology, *ICFPT2013 design competition* [Online]. Available: http://lut.eee.u-ryukyu.ac.jp/dc13/index.html.

[4] M. Janarbek, P. Meng, L. Wu, B. Weals, and R. Kastner. "Designing a hardware in the loop wireless digital channel emulator for software defined radio." In *IEEE FPT*, pp. 206-214, 2012.

[5] Altera, Corp., San Jose, CA. *DE2-115 Data Sheet*, 2010.

[6] M. S. Campbell, and T. A. Marsland. "A comparison of minimax tree search algorithms." *Artificial Intelligence 20,* no. 4, pp. 347-367. 1983.

[7] Faustino Frechilla. "Block'em", 2011 [Online]. Available: http://blockem.sourceforge.net/index.html.

[8] Enz. "Pentobi", 2011 [Online]. Available: http://pentobi.sourceforge.net.