Synthesis of LUT Networks for Random-Looking Dense Functions with Don't Cares — Towards Efficient FPGA Implementation of DNN

Yukio Miyasaka[1], Alan Mishchenko[1], John Wawrzynek[1], Nicholas J. Fraser[2]
[1]University of California, Berkeley, California, USA
[2]AMD Research and Advanced Development, Dublin, Ireland
yukio_miyasaka@berkeley.edu

***Abstract*— Many EDA applications deal with logic functions representing complex mathematical computations. Although in many cases, these functions depend on a small number of inputs, they often resemble random functions, making it hard to synthesize them using the traditional methods based on SOP minimization. This paper describes efficient synthesis and LUT mapping for this class of functions using a novel method that implements BDD-based minimization based on truth tables. The paper also investigates optimization with don't cares, when the outputs of a function are unspecified for some inputs, which is particularly useful in machine learning applications that trade accuracy for area. Compared to optimization and mapping used in academic and industrial tools, our method works faster and results in 1.5x smaller networks, while extra 20% area reduction was possible with don't cares at almost no accuracy cost.**

I. Introduction

Logic synthesis takes Boolean functions in the form of truth tables, sums-of-products (SOPs) or unoptimized circuits, and produces optimized circuits that are used to map the design into a target technology, typically FPGAs or standard cells. Improving logic synthesis methods remains an important challenge for the developers of modern design automation tools, especially given that design sizes keep growing while users expect tools to get faster.

In practice, different types of functions call for different logic synthesis methods. For example, Boolean functions appearing in control logic blocks (such as state-machines) are amenable to synthesis by algebraic methods [1] applied to minimized SOPs [2]. These functions having compact SOP representations can be described as *sparse* because their primes tend to have relatively few literals and cover large areas of Boolean space.

In contrast, *dense* Boolean functions are those that do not have compact SOPs. A class of dense functions can be found in router designs, where this type of logic is often expressed almost exclusively using large multiplexers and one-hot-encoded selectors. An effective way to handle these circuits is to recognize the multiplexers (or avoid bit-blasting them during elaboration) and perform restructuring, followed by specialized logic sharing extraction. Another class of dense functions are those rich in XOR gates, appearing in CRC checkers and cryptographic applications.

Despite decades of research, a "universal" synthesis method has not been discovered. Attempts to apply one synthesis method to all types of logic leads to mediocre results, prohibitive runtime, or both. For example, applying algebraic methods to XOR-rich logic leads to poor quality, while applying them to multiplexer-rich designs often gives good results but only after many synthesis iterations, rendering such an approach impractical due to long runtime.

In this paper, we develop a novel synthesis method targeting a class of logic functions, which can be characterized as *random-looking dense functions*, especially for LUT mapping. *Random-looking* implies that these functions are hard for synthesis but they are not random, because truly random functions can be compacted only by brute-force cofactoring [3]. On the other hand, the fact that the functions are *dense* means that they do not have compact SOPs.

Random-looking dense functions arise in several applications. One of these applications is deep neural networks (DNNs). For example, the Boolean functions of quantized neurons computed in LogicNets [4] are random-looking and dense. Synthesis and mapping of Boolean functions arising in the LogicNets project has served as a primary motivation for this work, while other arithmetic functions such as exponentiation, sigmoid, or trigonometric functions are also our potential target.

In some applications, especially DNNs, the functions do not always have to be implemented exactly. Approximate logic synthesis has been playing a roll for these applications by trading accuracy for area [5]. Our method, in contrast, can take an incompletely specified function as input, where the acceptable errors are specified as external don't cares by the user before synthesis. In our experiment, we assigned the patterns that are rarely observed in the training set to don't care. The results show that it can reduce the number of LUTs about 20% with little accuracy drop.

We developed a method based on binary decision diagrams (BDDs) [6], while our implementation uses truth tables as a primary data structure. Since we are interested only in the number of BDD nodes for area optimization, we do not need to maintain BDDs explicitly with several auxiliary data structures. Instead, we count unique cofactors on each level in a top-down manner in truth tables, where a truth table is seen as an expanded decision diagram. This way we can get the exact number of nodes in the BDDs without actually constructing them. Next, we perform variable reordering in the truth table, as we would have done in the BDDs while trying to minimize the number of BDD nodes, and

use the don't cares if they are available. The don't-care-based minimization is similar to the known methods on BDD minimization [7], as will be discussed in the background section.

We note several advantages of not explicitly using BDDs:

- the runtime with truth tables is about 1.5x faster
- no need to develop or integrate a BDD package
- the code can be simplified by avoiding recursion

The contributions of the paper are:

- a novel truth-table-based method to perform logic synthesis for LUT mapping with or without don't cares
- isolating a practical class of functions, which allows for an efficient solution using the proposed method
- experimental evaluation demonstrating 1.5x reduction in area and 10x reduction in runtime, compared to the methods implemented in the existing CAD tools
- with don't cares, our method achieved extra 20% area reduction with little accuracy drop for DNNs

The rest of the paper is organized as follows: Section II gives some background on BDD. Section III describes the proposed algorithm. Section IV lists experimental results. Section V concludes the paper.

II. Background

A BDD is a binary tree that represents a single-output logic function [6]. Each non-leaf node is associated with an input variable, and depending on the value of the variable, one of its two child nodes is selected to determine the output value. BDDs are *ordered* if the variable associated with a node always precedes the variable associated with its child node in a given variable order. BDDs are *reduced* if every node has a unique function and is not *redundant*. A redundant node is a node with a function that has identical Shannon cofactors with respect to the associated variable. Therefore, every node has two unique child nodes in a reduced BDD. A child node may be shared by multiple nodes. For a multi-output function, a BDD is built for each output. The BDDs are *shared* if they are ordered using the same variable order and reduced together. EDA applications use ordered, reduced, and shared BDDs in order to achieve compact representation and efficient manipulation. In this paper, BDDs are always assumed to be ordered, reduced, and shared unless otherwise stated.

To further compress the size of a BDD, *complemented edges* are frequently used [8]. When a child node is connected by a complemented edge, the function of the child node is negated. With complemented edges, we need only one leaf (constant) node. When evaluating the output, we count the number of complemented edges passed from the root node, and flip the output value accordingly. We assume the use of complemented edges in this paper.

Since the size of a BDD depends on the variable order, there have been many studies to find a good variable order that makes the BDD small. Variable reordering by sifting [9] is one of the successful approaches. After building a BDD, this method picks up a variable with the largest number of nodes and iteratively swaps its position with the adjacent variable in the current variable order. The best position where the smallest BDD was observed is remembered during the iteration and restored at the end. The same procedure is repeated for the rest of the variables. This approach is efficient because each variable swap only affects the nodes associated with the swapped variables [10].

BDD minimization using don't cares was studied in [7]. In principle, BDDs can be minimized by merging nodes while keeping their function unchanged on the care set—the complement of the don't-care set. They proposed three matching criteria for merging nodes: one-sided don't care match (OSDM), one sided match (OSM), and two-sided match (TSM), with two heuristics on the order of nodes to compare as follows:

- Merging criteria
  - OSDM: $c_j = 0$
  - OSM: $(c_j \implies c_i) \wedge (c_j \implies (f_i = f_j))$
  - TSM: $(c_i \wedge c_j) \implies (f_i = f_j)$
- Comparison order
  - Sibling: child nodes of each node (depth-first)
  - Level: set of nodes on a given level

where (f_i, c_i) and (f_j, c_j) are the pairs, containing the function and the care set of the nodes to be compared. The OSDM and OSM conditions are not symmetric, so the comparison is performed both ways, and if it holds, (f_i, c_i) replaces (f_j, c_j). If the TSM condition holds, both of the nodes are replaced with a new node (f_k, c_k), which satisfies

$$(c_i \implies (f_i = f_k)) \wedge (c_j \implies (f_j = f_k)),$$
$$c_k = c_i \vee c_j.$$

For OSM and TSM, a *complemented match* is also considered, where f_j in the condition is negated, and the merged node will be pointed to by a complemented edge.

The sibling order starts the comparison from the root node and recurs on each child node. If the child nodes have been merged, it recurs only on the merged node. The level order performs the comparison only for the pairs of nodes on the same level, but including redundant nodes. Finding the optimal order to pick up a pair is NP-complete, so they use a heuristic where once the nodes are merged, the merged node is compared with the rest of nodes before proceeding to another pair. In the end, they proposed applying the sibling order OSM first and then the level order TSM in a partitioned BDD.

III. Proposed method

A. Truth-table-based node counting

Our method is as simple as constructing and reordering BDDs, but we implemented it in a novel way. The advantage of our implementation comes from small memory usage. Instead of storing the structure of BDDs as pointer-connected nodes, we operate on the given truth table and its indices.

The proposed top-down procedure to count the number of BDD nodes in the function represented by the truth

```
// number of inputs
int nInput;
// number of outputs
int nOutput;
// vector of unique indices for each level
vector<vector<int>> vvIdx(nInput);
// vector of redundant indices for each level
vector<vector<int>> vvRedIdx(nInput);

int FindOrAdd(int idx, int lev) {
  if(IsConst0(idx, lev))
    return -2;
  if(IsConst1(idx, lev))
    return -1;
  for(int loc = 0; loc < vvIdx[lev].size(); loc++) {
    if(IsEq(idx, vvIdx[lev][loc], lev))
      return loc << 1;
    if(IsComplEq(idx, vvIdx[lev][loc], lev))
      return (loc << 1) | 1;
  }
  int loc = vvIdx[lev].size();
  vvIdx[lev].push_back(idx);
  return loc << 1;
}

int CountNodes() {
  for(int idx = 0; idx < nOutput; idx++)
    FindOrAdd(idx, 0);
  for(int lev = 1; lev < nInput; lev++)
    for(int idx: vvIdx[lev - 1]) {
      int cof0 = FindOrAdd(idx * 2, lev);
      int cof1 = FindOrAdd(idx * 2 + 1, lev);
      if(cof0 == cof1)
        vvRedIdx[lev - 1].push_back(idx);
    }
  int count = 1; // constant node
  for(int lev = 0; lev < nInput; lev++)
    count += vvIdx[lev].size() - vvRedIdx[lev].size();
  return count;
}
```

Code 1. Counting the number of nodes

table is shown in Code 1. We use 0-based-indexing for all potential (non-reduced) BDD nodes for each level as shown in Fig. 1. For multi-output functions, there are as many nodes as outputs on the 0th (top) level. The function of a node is represented by a segment of the truth table. For example, each node corresponds to a segment of 2^N rows on the 0th level, where N is the number of inputs. On the k-th level, each node corresponds to a segment of 2^{N-k} rows, while the top k variables are not used in their functions.

For each level, our proposed procedure creates a vector of *unique indices*, whose functions are unique and non-constant (checked in the sub-procedure in Code 1). After the 0th level, it only needs to check the indices that are the cofactors of the unique indices on the previous level. Meanwhile, if the cofactors are identical (confirmed by comparing their locations in the vector, returned by the sub-procedure), the unique index is redundant, so it is stored in another vector to be exclude from the final count. To realize the complemented edges, the function of each index is also compared with the complemented functions of the unique indices already existing in the vector. This complementary information is returned by the sub-procedure along with the location, while constants are expressed by negative locations.

For the function in Fig. 1, all indices are unique on the 0th and 1st levels. On the other hand, index 0 is the only unique index on the 2nd level: index 1 has the same function as

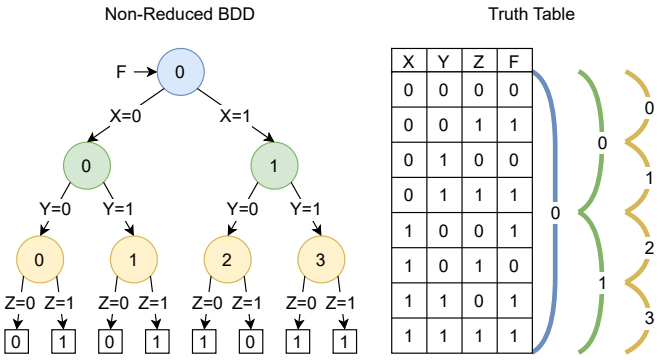


Fig. 1. Indices of potential BDD nodes and the corresponding segments of the truth table for a single output function F with input (X, Y, Z). The constant node is duplicated without complemented edges for visibility sake.

index 0, index 2 has its complement, and index 3 is constant. Since index 0 and 1 are the same on the 2nd level, index 0 on the 1st level is redundant. Therefore, this function takes 4 BDD nodes: 4 unique indices minus 1 redundant index plus 1 constant.

B. Variable reordering

We can perform variable swap in truth table simply by sorting its entries. However, although sorting is simple, it is not as efficient as variable swap in the form of BDD. Reconstructing BDDs after variable swap takes a linear time over the number of nodes in the level below. On the other hand, sorting takes a linear time over the number of truth table segments therein, which is an upper bound of the former number.

The proposed implementation adopts a hybrid approach. When counting the number of nodes for the first time, we also memorize the cofactors of the unique indices. Then, we perform variable swap in a traditional way while instantiating a unique table. We reconstruct the lower level based on 2-level cofactors using the unique table; refer to [10] for more details. This unique table is a bit simpler than usual since it uses only a pair of cofactors as a key. The unique indices in the upper level remain the same, while their redundancy will be reevaluated using the new cofactors.

C. BDD minimization using don't cares

We implemented BDD minimization using don't cares on top of our truth-table-based algorithm. We adopted the level order TSM, as presented in Section II. The sibling order does not match our top-down algorithm. The reason why we chose TSM is that, in the case of OSDM and OSM, we have to check redundancy of nodes afterwards in a bottom-up manner. Even if the child nodes look different when they are checked, their functions could be modified by node merging later. For example, let us assume that the 00-cofactor and the 11-cofactor have the same function and no don't care, while the 01-cofactor and the 10-cofactor have different functions but are entirely don't care. Since the 0-cofactor and 1-cofactor do not satisfy the merging condition, they are regarded as different unique indices. However, when

checking their level, the 01-cofactor will be merged with the 00-cofactor, and the 10-cofactor will be merged with the 11-cofactor, having the same function. Then, not only the 0-cofactor and 1-cofactor become redundant, their parent node also becomes redundant. This does not happen in TSM because, if a node can be redundant, its child nodes must satisfy the condition and be merged immediately.

We can perform the minimization simply by replacing the equivalence check of nodes with TSM. If an existing unique index j matches the given index i, it updates the function and the care set of j to $(ITE(c_i, f_i, f_j), c_i \vee c_j)$, which is one case of (f_k, c_k) shown above.

With BDD minimization using don't cares, reordering is no longer simple. The result of variable swap is affected by the minimization performed previously. To see the actual effect of variable reordering, we have to restore the original BDDs before variable swap. While this looks complicated, it can be integrated quite easily with our truth-table-based implementation; we only need to reload and sort the original truth table and rerun the minimization. Additionally, we can save time by remembering the nodes above the swapped level and how they were merged. This approach, performing reordering based on the result of minimization, results in a significant reduction in the number of nodes as compared to just performing the minimization to reordered BDDs.

IV. Experimental results

A. Benchmarks

We used Boolean functions from the LogicNets project [4] as our target. These are functions of quantized sparse neural networks, where each neuron is represented as a truth table to comprise a truth table network. Let β denote the bit-width of each activation and γ denote the number of input activations per neuron, which are uniform across each network. The number of inputs is $\beta \times \gamma$ and the number of outputs is β for each truth table. The properties of the neural networks used to generate functions in our experiments are shown in Table I. The LogicNets networks were trained on three publicly available datasets: 1) jet substructure classification (JSC) [11]; 2) network intrusion detection (NID) [12]; and 3) handwritten digit classification (MNIST) [13];

For some of the networks listed in Table I, the number of inputs and outputs to each truth table is different for the first and last layers of the network as shown below. The input bit-width and the number of input activations in the first layer are given by β_i and γ_i, respectively. The number of input activations for the last layer is given by γ_o, and the output bit-width of the last layer is given by δ. The number of inputs and outputs for each truth table is $\beta_i \times \gamma_i$ and β in the first layer, and $\beta \times \gamma_o$ and δ in the last layer.

- JSC_L: $\beta_i = 4$, $\gamma_i = 3$, $\gamma_o = 5$, $\delta = 7$
- NID_S: $\beta_i = 1$
- NID_M: $\beta_i = 1$
- NID_L: $\beta_i = 1$, $\gamma_i = 7$
- MNIST_S: $\delta = 4$
- MNIST_M: $\delta = 4$

TABLE I
LogicNets benchmark

Name	Neurons per layer	β	γ	Accuracy
JSC_S	64, 32, 32, 32	2	3	69.41%
JSC_M	64, 32, 32, 32	3	4	71.90%
JSC_L	32, 64, 192, 192, 16	3	4	73.01%
NID_S	593, 100	2	7	89.36%
NID_M	593, 256, 128, 128	2	7	92.62%
NID_L	593, 100, 100, 100	3	5	93.12%
MNIST_S	2^{10}, 2^{10}, 2^{10}, 2^{10}, 2^{10}, 128	1	8	95.83%
MNIST_M	2^{10}, 2^{10}, 2^{10}, 2^{10}, 2^{10}, 128	2	5	97.97%

TABLE II
Performance comparison (time in sec and memory in MB)

	CUDD		Swap Table		Swap Node	
	Time	Mem	Time	Mem	Time	Mem
JSC_S	1.62	14.3	0.04	3.6	0.07	3.6
JSC_M	3.38	14.4	4.80	3.8	1.92	3.8
JSC_L	10.68	15.0	18.94	4.1	6.24	4.2
NID_S	8.36	14.6	4.35	3.8	1.94	3.8
NID_M	15.34	14.7	10.90	3.9	5.18	4.1
NID_L	17.54	15.2	40.32	4.2	9.78	4.2

B. Performance of truth-table-based algorithm

We first conducted a runtime comparison against an existing BDD package, CUDD [14]. We performed variable reordering by sifting, starting with a random initial variable order, 20 times for each neuron [15]. For all experiments, we used an AMD EPYC 7313 processor.

The runtime and memory footprint are shown in Table II. Our truth-table-based algorithm used less memory than CUDD, and with the node-based variable swap, it worked more than 1.5x faster than CUDD. CUDD takes a large amount of memory to maintain data that are useful to dynamically apply many operations, which is not the case here. Our truth table algorithm uses memory just to store the truth table and the unique indices if variable swap is performed by sorting the truth table. With node-based swap, it also needs a storage for cofactors, which explains the observed overhead. JSC_S is a special case where each truth table fits in one word (64 bits) and the table-based swap worked faster than the node-based swap.

C. LUT mapping comparison

We performed synthesis and mapping for 6-LUT networks. For each neuron, we construct and reorder BDDs and map them into 6-LUTs. We did not use don't cares in this experiment. Since each BDD node is a multiplexer, we constructed a LUT network by grouping the multiplexers in a depth-first order to have nodes with at most six inputs. Typically, two cascaded multiplexers are grouped into a node with five inputs. Then, as an inter-neuron optimization, we apply ABC's optimization [16] ("mfs2" and "&if -K 6 -a") iteratively as long as the area keeps on reducing. The command "mfs2" optimizes a given network without converting it into AIG, and "&if" reperforms mapping to restructure the network and create more optimization opportunities for "mfs2". Other AIG optimization commands in ABC did not work well to reduce the LUT count for these

benchmarks, sometimes resulting in a larger network than without optimization.

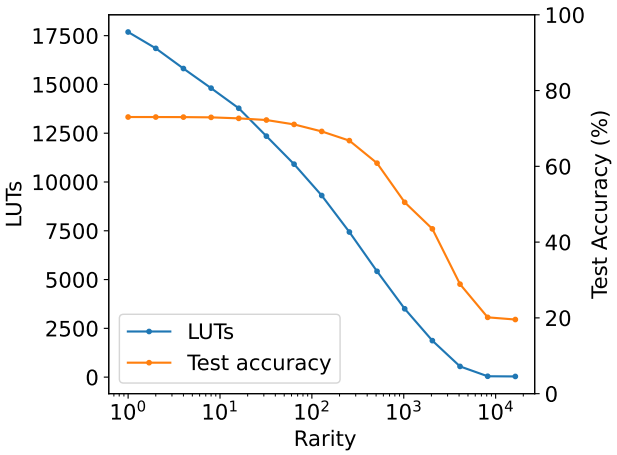
We compared our method with Xilinx Vivado 2021.2 with default synthesis settings, but with "resource sharing" set to "on". Additionally, we tried FBDD [17], which implements one of the state-of-the-art BDD-based synthesis algorithms. We applied FBDD to each neuron with a 1 minute time limit, and mapped the synthesized circuit using ABC. Note that FBDD was not stable in our environment and sometimes crashed. In case of timeout or crash, we passed the original truth table of the neuron to ABC. We also checked the results when only ABC's optimization was applied.

The results are shown in Table III. Besides LogicNets benchmark, we synthesized 12-input 32-output discrete cosign transform (DCT) taken from [18], as an example of mathematical function. In general, Vivado took the longest time, 10x slower than the proposed method on average, while ABC and FBDD did not work well, ending up with much larger overall area. Our method achieved about 1.5x area reduction compared to Vivado except JSC_S and MNIST_S, where the number of inputs per neuron is too small (6 and 8) for our method to work.

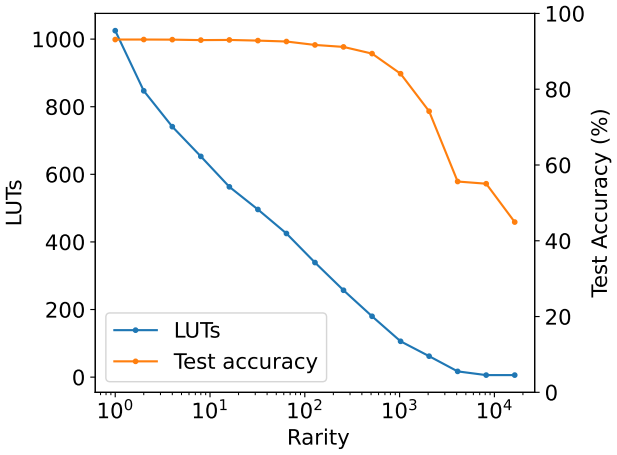
D. Optimization using don't cares

Next, we performed don't-care-based optimization when some input patterns are treated as don't cares. To begin with, we assigned don't care for the patterns that do not appear at the inputs of a neuron on any examples from the training set. Furthermore, we used a threshold, called *rarity*, where the patterns that occur at least *rarity* times in the training set are cares, while the other patterns are don't cares. The same pattern can appear multiple times in the training set because different examples may result in the same quantized values at the inputs on a neuron. We considered values of rarity equal to powers of two.

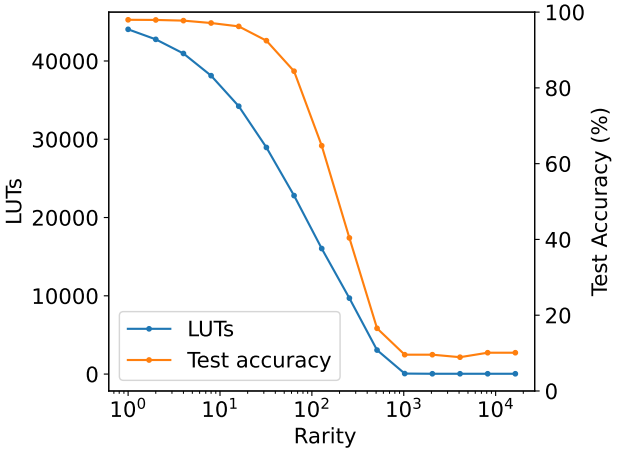
We optimized the largest model for each benchmark, JSC_L, NID_L, and MNIST_M. The results are shown in Fig. 2. The LUT count decreased as the rarity increased, while the accuracy decreased quite slowly compared to the LUT count until a certain rarity. The curve for training accuracy was omitted because it mostly overlaps with that for test accuracy. Table IV shows the values for the first three points as well as the values from the previous experiment without don't cares as a reference. With rarity 1, where all patterns appeared in the training set were preserved, we got about 20% area reduction for JSC_L and MNIST_M with less than 0.01% test accuracy drop, and 75% reduction for NID_L with 0.02% drop. By increasing the rarity, where some patterns in the training set were also assigned to don't care, we observed a little drop in both training and test accuracy, while we got about 10% more reduction at rarity 4. The runtime was at most three minutes.



(a) JSC_L



(b) NID_L



(c) MNIST_M

Fig. 2. Effect of don't-care-based optimization when changing rarity

V. Conclusion

The paper motivates the development of specialized logic synthesis methods for well-defined classes of Boolean functions. To this end, we isolate a practical class of Boolean

TABLE III

SYNTHESIS AND LUT MAPPING (TIME IN SEC)

	Vivado		ABC only		FBDD		Proposed method	
	LUTs	Time	LUTs	Time	LUTs	Time	LUTs	Time
JSC_S	227	32	242	0.5	242	7	233	1
JSC_M	14865	683	31647	13	26422	73	9665	28
JSC_L	35419	1207	87065	37	69547	263	22997	89
NID_S	85	1563	30	7	31	84	29	24
NID_M	2690	6783	4080	31	3756	183	1969	52
NID_L	6672	14242	13888	125	14662	1278	4057	173
MNIST_S	13509	496	16941	6	15729	229	15078	19
MNIST_M	73991	1344	121894	76	157377	341	57781	95
DCT	808	75	6627	7	6597	268	507	2

TABLE IV

RESULTS OF DON'T-CARE-BASED OPTIMIZATION (*RARITY 0 SHOWS THE RESULTS WITHOUT DON'T CARES AS A REFERENCE)

	Rarity	LUTs	Train Accuracy	Test Accuracy
JSC_L	*0	22997	73.17%	73.01%
	1	17687	73.17%	73.01%
	2	16849	73.17%	73.00%
	4	15815	73.15%	72.99%
NID_L	*0	4057	93.35%	93.12%
	1	1025	93.35%	93.10%
	2	847	93.35%	93.11%
	4	741	93.34%	93.08%
MNIST_M	*0	57781	97.85%	97.97%
	1	44045	97.85%	97.97%
	2	42763	97.83%	97.93%
	4	40959	97.67%	97.75%

functions characterized as random-looking dense functions. We observe that these functions appear in quantized neural networks, datapath applications, cryptographic applications, and possibly other areas.

For the selected class of functions, a novel synthesis method is proposed. The method iteratively reorders the truth table representation of the function while trying to minimize the number of nodes which would be present in the BDDs of the function. Our truth-table-based implementation worked about 1.5x faster than an implementation using a state-of-the-art BDD package.

The experimental results show that the area improvements produced by the proposed methods are substantial, leading to 1.5x reduction, compared to the results produced on these benchmarks by the available tools, both academic and commercial. The proposed method is also often at least 10x faster than those tools.

Moreover, our method can exploit don't cares if they are present in the specification of the function. For the LogicNets benchmarks, by treating the patterns not present in the training set as don't care, we achieved 20% more area reduction with almost no accuracy drop. By increasing the number of don't cares, our method was able to efficiently trade accuracy for area furthermore until a certain threshold where the accuracy starts to degrade rapidly.

Regarding the scalability, the runtime of our algorithm increases exponentially over the number of inputs, as the size of truth table grows exponentially. We internally tested our method on truly-random completely-specified functions of 16 to 24 inputs, and the runtime increased exponentially by a factor of 3 (20 seconds for 20 inputs). The runtime of CUDD-based implementation gradually caught up as the number of inputs increased, but it suddenly blew up after 22 inputs, taking more than 30 minutes for 23 inputs, probably because of complicated memory management such as garbage collection.

ACKNOWLEDGEMENT

This research was supported in part by the SRC Contract 3173.001 "Standardizing Boolean transforms to improve quality and runtime of CAD tools" and the NSA grant "Novel methods for synthesis and verification in cryptanalytic applications".

REFERENCES

[1] R. Brayton *et al.*, "The decomposition and factorization of Boolean expressions," in *Proc. ISCAS*, 1982, pp. 29–54.

[2] R. Rudell *et al.*, "Multiple-valued minimization for PLA optimization," *IEEE TCAD*, vol. 6, no. 5, pp. 727–750, 1987.

[3] T. Sasao *et al.*, "LUTMIN: FPGA logic synthesis with MUX-based and cascade realizations," in *Proc. IWLS*, 2009, pp. 310–316.

[4] Y. Umuroglu *et al.*, "LogicNets: Co-designed neural networks and circuits for extreme-throughput applications," in *Proc. FPL*, 2020, pp. 291–297.

[5] Y. Qian *et al.*, "Approximate logic synthesis in the loop for designing low-power neural network accelerator," in *Proc. ISCAS*, 2021, pp. 1–5.

[6] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE TC*, vol. C-35, no. 8, pp. 677–691, 1986.

[7] T. R. Shiple *et al.*, "Heuristic minimization of BDDs using don't cares," in *Proc. DAC*, 1994, pp. 225–231.

[8] K. Brace *et al.*, "Efficient implementation of a BDD package," in *Proc. DAC*, 1990, pp. 40–45.

[9] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. ICCAD*, 1993, pp. 42–47.

[10] M. Fujita *et al.*, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis," in *Proc. DATE*, 1991, pp. 50–54.

[11] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *JINST*, vol. 13, no. 07, P07027, 2018.

[12] N. Moustafa *et al.*, "UNSW-NB15: A comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *Proc. MilCIS*, 2015, pp. 1–6.

[13] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[14] F. Somenzi, "Efficient manipulation of decision diagrams," *Int. J. on STTT*, vol. 3, no. 2, pp. 171–181, 2001.

[15] P. Fiser *et al.*, "How much randomness makes a tool randomized?" In *Proc. IWLS*, 2011.

[16] Berkeley Logic Synthesis and Verification Group, *ABC: A system for sequential synthesis and verification, release 20306.* `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[17] D. Wu *et al.*, "FBDD: A folded logic synthesis system," in *Proc. Int. Conf. ASIC*, vol. 2, 2005, pp. 746–751.

[18] *OpenCores: Video compression systems*, `https://opencores.org/projects/video_systems`.