OpenFPGA Documentation

Release 1.0

Xifan Tang

Dec 01, 2020

MOTIVATION

CHAPTER

ONE

MOTIVATION

The growing complexity of semiconductor chips has revealed two critical limitations of contemporary EDA techniques. First, while contemporary EDA methodologies are extremely well developed and highly optimized, they usually only target specific types of logic networks. For instance, And-Inverter Graph (AIG)-based logic optimizations perform well for control-intensive logic networks, while the recently developed Majority-Inverter Graph (MIG)-based logic optimizations are efficient for the arithmetic-intensive logic network. As contemporary semiconductor chips become more diverse in functionality than before, using a unified logic optimization methodology may no longer guarantee best Performance-Power-Area (PPA). To tightly follow the trend in circuit complexity, it is worthwhile to study how to partition a design efficiently into portions of specific attributes (arithmetic, control, etc.) and apply ad-hoc logic optimization techniques on them. In contemporary EDA flow, such optimizations are only achieved by intensive manual efforts and strong expertise is required. Another contemporary issue of logic synthesis is the difficulty to parallelize the optimization steps. As the complexity of chips exploded in recent years and considering that such trend may remain in future, using a traditional logic optimization methodology causes the runtime of EDA flows (even only the runtime of logic synthesis) to be more than 24 hours. Large semiconductor designs can be partitioned into equal blocks that are distributed over different processors for EDA optimizations to achieve parallelization. However, such simple parallelism would reduce runtime at the cost of PPA degradation. Indeed, as partitioning would be achieved regardless of logic attributes, each partition would have to be treated with generic optimization techniques, leading the final QoR to be far from the optimum. Therefore, we target a fast, automatic tool capable of identifying efficient logic optimization techniques for the different portion of circuit designs. By partitioning SoC designs in terms of logic attributes and applying ad-hoc logic optimization technique, it brings an opportunity for logic synthesis to achieve significant runtime reduction and performance improvement simultaneously.

CHAPTER

TWO

EPFL LOGIC SYNTHESIS LIBRARIES

2.1 Alice

Alice is a library developed in C++14, by EPFL, that provides the LSOracle shell interface. Alice allows the user to create commands to manipulate user-defined data types. Furthermore, Alice supports Python bindings, easing the task of creating scripts. To access the detailed documentation, please refer to Alice Documentation.

2.2 Kitty

Kitty is a C++14 library that provides basic structure for truth table generation and manipulation. In the context of this project, kitty is extended to extract the logic attributes of different partitions. The attributes are represented to black and white figures, which are taken as input by the Neural Network to delegate the appropriate optimizer for a given partition. For a detailed explanation about the library, please access the Kitty Documentation.

2.3 Lorina

Lorina is a C++ library to parse logic synthesis recurrent file formats. LSOracle uses the And-Inverter Graph (AIG) reader from lorina. Also, the Verilog writer was extended to output sequential circuits, so that sequential Majority-Inverter Graphs (MIGs) can be functionally verified, and used in further steps of the ASIC flow.

The Lorina full documentation is available here.

2.4 Mockturtle

Mockturtle is a C++17 library that implements different kinds of Boolean networks, as well as algorithms for logic manipulation. So far, LSOracle uses AIGs and MIGs to optimize different partitions of a circuit, taking advantage of the best features of each optimizer.

The MIGs capabilites were extended on this library, so that Mockturtle can also deal with sequential MIGs. In order to unlock sequential MIG logic optimization, the depth and area-oriented MIGs optimization algorithms were adapted.

Mockturtle complete documentation is available on this link.

CHAPTER

THREE

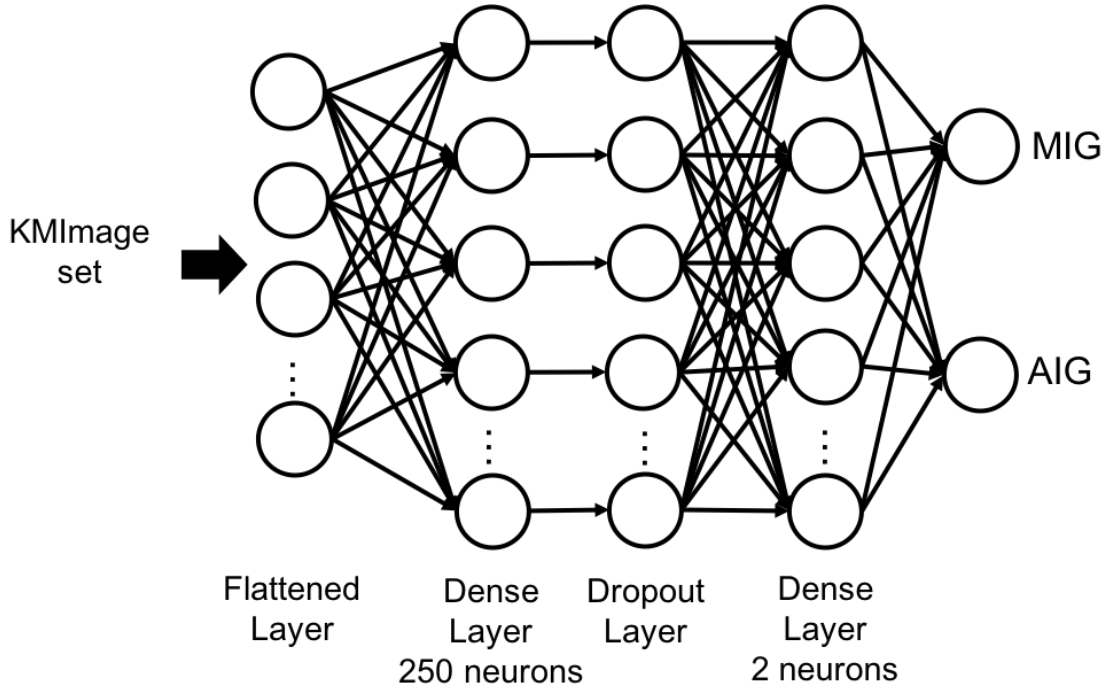
LOGIC SYNTHESIS ORACLE

3.1 Deep Learning Network Classifier

A Deep Neural Network (DNN) is used to determine the best-fit algorithm to optimize a partition. Image representations of a karnaugh-map are used (KM-Image) to model the logic network's functionality in the 2D space. An example of a KM-Image of a 6-input logic cone is shown below.

ABC \ DEF	000	001	011	010	110	111	101	100
000	1	1	1	1	1	1	1	1
001	1	1	0	0	0	0	1	1
011	1	1	1	1	0	0	0	0
010	1	0	0	1	1	0	0	1
110	1	0	0	1	1	0	0	1
111	1	1	1	1	0	0	0	0
101	1	1	0	0	0	0	1	1
100	0	0	0	0	0	0	0	0

The DNN model was based primarily on a solution to the MNIST handwriting problem because the input for both applications are simple grayscale images and there are very few classes (10 in MNIST and 2 in this application). The topology is shown below.



Combinational networks from the EPFL and ISCAS85 benchmark suites were partitioned into smaller sub-circuits, and the logic cones from these sub-circuits were optimized using both AIG and MIG methods in order to create the training dataset. The best optimization method was determined which reduced the area of the cone the most. We used this metric because it resulted in a relatively balanced dataset. After training with this dataset, the overall accuracy achieved with this model was 79% with similar accuracy for each of the classes meaning that the model is not biased to one method over the other.

3.2 Partition Manager

The partition manager is the class responsible for creating and managing partitions over a Boolean network. The manager is in charge of partitions classification and assignment to different optimizers. Finally, the manager synchronizes optimized partitions over the original Boolean network.

For partition generation, the manager receives as input the circuit Boolean network within the number of desired partitions by the user, and interfaces with the KaHyPar tool for k-way partitioning. KaHyPar returns a vector *A*, where the index *i* corresponds to the node index, and the value *A[i]* refers to which partition the *i th* node belongs. From this vector, the partition manager keeps track of partitions inputs, outputs, and nodes, as well as connections in between partitions. These kinds of information are used by the *partition_view* method for optimization purposes.

As for the classification, the partition manager is responsible for generating the KM-Images for every logic cone. The generated images are then classified and assigned to different optimizers by Frugally Deep. Since the same partitions might have different cones assigned to different optimizers, the manager applies a heuristic to weight each cone. The optimizer with the highest sum of weights is used for the partition optimization.

After optimization has taken place over a set of nodes (partition), the manager synchronizes the optimized partition with the original network. Therefore, proper connections are established with adjacent partitions, keeping the functional equivalence.

3.3 Partition View

Partition view works in a similar way to the window_view. It receives a Boolean network, and two sets of nodes: one bounding the partition inputs and the other defining the partitions outputs. For each output, the method recursively adds nodes in the partition scopes until it reaches the partitions inputs. Note that partitions are non-overlapping, and one node cannot belong to more than one partition. It is ensured by the *partition_manager* class. Conversely to the *window_view*, the *partition_view* is mutable and may be passed straight to any optimization method.

3.4 Optimization

3.4.1 Optimization with Classification

The main functionality of this command is to use the DNN model to determine the best-fit algorithm for each partition of a network. The general flow of this command is: read network into either AIG (must begin with AIG network), perform k-means hypergraph partitioning, determine the algorithm to use with each using classification, perform optimization on each partition, merge partitions back into one, and write out an optimized network. An example of using this command is shown below.

```
//Read AIGER file and store as AIG network
read_aig <AIGER file>
//Partition stored AIG
partitioning <number of partitions>
//Perform classification, optimization, merging, and write out to Verilog file
optimization -c deep_learn_model.json -o <Verilog file to write to>
```

There are also a couple other options for this command.

Partitions with only AIG Optimization

This option skips classification and optimizes all partitions using just AIG optimization.

```
//Read AIGER file and store as AIG network
read_aig <AIGER file>
//Partition stored AIG
partitioning <number of partitions>
//-a flag denotes using AIG optimization
optimization -a -o <Verilog file to write to>
```

Partitions with only MIG Optimization

This option skips classification and optimizes all partitions using just MIG optimization. Note that the AIGER file must still be stored as an AIG network. Conversion for MIG optimization will occur after partitioning.

```
//Read AIGER file and store as AIG network
read_aig <AIGER file>
//Partition stored AIG
partitioning <number of partitions>
//-m flag denotes using MIG optimization
optimization -m -o <Verilog file to write to>
```

High Effort Classification

Instead of using the DNN model, a high effort method is used to determine what algorithm to use for each partition. After partitioning, each partition is optimized using both AIG and MIG methods and the one that reduces the area and delay product is the method to be used.

Note: This metric is not the same used to train the DNN model, and the model was also trained on cones rather than full partitions.

```
//Read AIGER file and store as AIG network
read_aig <AIGER file>
//Partition stored AIG
partitioning <number of partitions>
//-b flag denotes using high effort classification
optimization -b -o <Verilog file to write to>
```

3.4.2 Two-Step Mixed Synthesis

This method also uses classification, but it is slightly different. This method also partitions an AIG network and performs classification, but it only optimizes partitions that have been classified to use AIG optimization. The resulting network is then partitioned and re-classified, and now MIG classified partitions are optimized. Similar to the above command, the flag -b can be used instead of -c deep_learn_model.json to use a brute force method for classification.

```
read_aig <AIGER file>
mixed_2step -p 5 -c deep_learn_model.json -o <Verilog file to write to>
```

3.5 Detailed Command Usage

- aigscript

  Performs homogeneous AIG optimization using interleaved rewriting, refactoring, and balancing, similar to ABC's resyn2. AIG network must be stored before use.

- balance

  Performs network balancing. Uses AIG network by default. Use -m flag to balance stored MIG network.

- cut_e

  Performs cut enumeration on a stored AIG network

- cut_rewriting

  Performs cut rewriting on a stored network (AIG default. MIG use -m option). Default cut size is 4; use -c INT to set an alternative cut size.

- depth

  Displays depth of current network. AIG default; use -m for MIG, -x for XAG.

- depthr

  Performs depth oriented MIG rewriting. MIG only.

- interleaving

  Performs interleaved MIG rewriting and refactoring.

- migscript

  Performs homogeneous MIG optimization using interleaved rewriting, refactoring, and balancing, similar to ABC's resyn2. MIG network must be stored before use.

- optimization

  Performs Mixed Synthesis on network after partitioning. Network must be loaded and separately partitioned before use.

  - "-n" to specify neural network model
  - "-o" to specify verilog output
  - "-s INT" to specify classification strategy. 0 = Balanced, 1 = Area optimized, 2 = Delay optimized
  - "-a" to perform only AIG optimization
  - "-m" to perform only MIG optimization
  - "-c" to combine adjacent partitions of the same type
  - "–skip-feedthrough" to not include feedthrough nets when writing output

- oracle

  All in one command to partition stored AIG network and perform mixed synthesis, as with "optimization" command. Use

  - "–partition INT" to manually specify the partition count instead of using the automatic selection.

- rwscript

  XAG cut rewriting

- generate_truth_tables

  Generates truth tables for every logic cone in every partition. AIG default; use "-m" for MIG

- read_aig

  Reads an AIGER file (binary, not ASCII) and stores the resulting network. It is stored as an AIG by default.

  - "-m" store resulting network as MIG
  - "-x" store resulting network as XAG

- read_bench

  Reads a bench file and stores the resulting network as a KLUT network.

  - "-a" store resulting network as AIG
  - "-m" store resulting network as MIG
  - "-x" store resulting network as XAG

- read_blif

  Reads a bench file and stores the resulting network as a KLUT network.

  - "-a" store resulting network as AIG
  - "-m" store resulting network as MIG
  - "-x" store resulting network as XAG

- read_verilog

Reads a simple, structural verilog file and stores the resulting network as AIG. If you need to read real world verilog files,

  - "-m" store resulting network as MIG

- lut_map

  Converts the stored network to an LUT network. Reads from the stored AIG network by default.

  - "-m" read from MIG network instead
  - "-K INT" LUT size for mapping (default 6)
  - "-C INT" Max number of priority cuts (default 6)
  - "-o FILENAME" Write LUT mapping to bench file

- ps

  Print network statistics for:

  - "-a" the AIG network
  - "-m" the MIG network
  - "-x" the XAG network
  - "-k" the KLUT network
  - "–all" all stored networks

- disjoint_clouds

  Writes status of all disjoint combinational clouds in the stored AIG.

- get_all_partitions

  Exports every partition to a verilog file. Must provide the directory to write to as a positional argument.

  - "-m" export partitions in the MIG network

- print_karnaugh

  Print all partitioned truth tables as karnaugh maps

- show_ntk

  Display details of the stored network. AIG default.

  - "-m" show MIG network

- techmap

  Experimental ASIC mapper. Not ready for general use; please contact developers if you would like a detailed usage guide.

- write_bench

  Write KLUT network to bench format

- write_blif

  Write KLUT network to blif format

- write_dot

  Writes AIG network to dot format for visualization. Do not use except with very small networks. * "-m" to write MIG network

- write_hypergraph

  Output stored network in hypergraph format used by hMETIS, gMETIS, etc. For use with an external partitioning tool.

  - "-m" write stored MIG network (default AIG)

- write_verilog

  Writes stored AIG network into very simple structural verilog. For robust verilog support, use Yosys plugin.

  - "-m" write MIG network
  - "-x" write XAG network
  - "–skip-feedthrough" exclude feedthrough nets in resulting verilog file

- crit_path_stats

  Determines the number of AND and MAJ3 nodes along the critical path in an MIG network.

- get_cones

  Displays size and depth of all logic cones in an AIG network

- ntk_stats

  Writes number of AND2 and MAJ3 nodes in stored MIG network

- partitioning

  Partition the AIG network. Number of partitions is a positional argument.

  - "-m" partition MIG network
  - "-c" path to config file for KaHyPar
  - "-f" path to external partition file, if using an external partitioner.

- partition_detail

  Display all nodes in each partition.

  - "-m" use stored MIG
  - "-n" use internal net names (default is yes for AIG)

CHAPTER

FOUR

CONTACT

General questions:

Prof. Pierre-Emmanuel Gaillardon

pierre-emmanuel.gaillardon@utah.edu

Technical Details about LSOracle:

Max Austin

max.austin@utah.edu

Walter Lau Neto

walter.launeto@utah.edu