# Predicting Power and Area for Chip Design

Other (Electronic Design Automation)
https://github.com/KPrabs106/rtl-synthesis-predictor

Kartik Prabhu (kprabhu7)
Kalhan Koul (kkoul)

November 8, 2019

## 1 Abstract

Chip design is an iterative and drawnout process. Each time an engineer wants an estimate of power or area for a given design, they must run logic synthesis, a procedure that takes relatively long. Our project focuses on speeding up iterations of the chip design process. The first step in our project was converting the Verilog code from behavioral to structural code and then converting this to a graph. Once we have a graph we can utilize Graph Convolutional Neural Networks to predict the area and power of a given design. We achieve our best results utilizing a Graph Convolutional Network consisting of ARMA Conv Layers. Since synthesis runtime depends on the size of a design and inference of a model has a constant runtime, we see massive benefits.

## 2 Introduction

Hardware design is a time consuming procedure. Once the design is written in a hardware description language, such as Verilog, a series of complex tools are used to realize the hardware into an actual design that can be fabricated. This can take anywhere from several hours to days for larger designs. First, in a step called logic synthesis, the code is mapped to logic gates. Then, place and route is performed, where gates are placed, and wires between them are routed. For engineers trying to get a quick estimate of how the changes in their code affect power and area numbers, this amounts to a lot of time wasted. The goal of this project is to use deep learning to reduce the latency in the steps between hardware design and area and power estimation.

## 3 Related Works

Machine learning has been applied to hardware design and CAD tools in many different ways. However, we were unable to find any literature on applying deep learning to accelerate the synthesis portion of design automation. First, we looked for research on performing deep learning on code. We were able to find several related works that helped describes converting code into an Abstract Syntax Tree [3]. We used this idea to realize we could turn our Verilog code into a tree and then turn the tree into a graph using [6]. Once we had a graph, we first turned to Node2Vec to create inputs for our model [2]. However, we quickly realized that this model would be hard to train for reasons discussed in a later section. After doing further research we realized the graph could be used directly as an input to a neural network. We used Spektral, a library that uses Keras to implement Graph Convolutional Networks. This library contained many different layer types that we tried in our models. For this report and our poster we focused on the GraphConv layer[4] and the ARMAConv layer[1]. These layers are described in more detail below. Overall, we were able to piece together many contemporary topics in machine learning for our project.

## 4 Describing the Dataset

Synopsys, a hardware design CAD company, has a package of small Verilog modules called DesignWare. This package consists of the basic building blocks of digital circuits, such as adders, multipliers, and arbiters. In total, there are roughly 200 small and configurable designs. Since these designs are configurable and take in parameters, we used data augmentation to increase the size of the dataset. More specifically, we can change bit-widths for the designs to generate more designs. We were able to create slightly over 4000 different combinational and sequential circuits.

To label the data, the designs go through synthesis with Synopsys Design Compiler using the open-source FreePDK 45nm technology library. Running these designs in parallel took several days to generate the data. Plenty of effort went into automating this process and the synthesis scripts can be found on our Github.

## 5 Method

Verilog is usually written in behavioral code. This describes the hardware at an abstract level using operators (+,-,*) and "if-else" statements. Synthesis takes in this behavioral RTL and first maps this to generic gates. Then, it selects the appropriate sizing for the gates, such that timing is met. Almost all the time is spent is sizing the gates, so this is the part we are targeting to accelerate with a neural network.
First, we take the behavioral Verilog code and generate generic Verilog structural code, using Synopsys Design Compiler. Verilog structural code consists of a gate level implementation of the design. This will look like a series of AND/OR/etc. gates. Once the design is described is at the gate level, the design can be then rewritten as a abstract syntax tree (AST) [5]. Using Yosys [6], a Verilog synthesis suite, an AST can be derived from the structural code. This AST can then be translated to a weighted graph using a Python script. In this weighted graph, vertices represent gates and edges represent nets. Now that we have a graph, we can turn to the model. We attempted two methods once the graph was generated: encoding the graph as a vector and using traditional CNNs and using (Graph Convolutional Neural Networks) GCNNs directly on the graph.

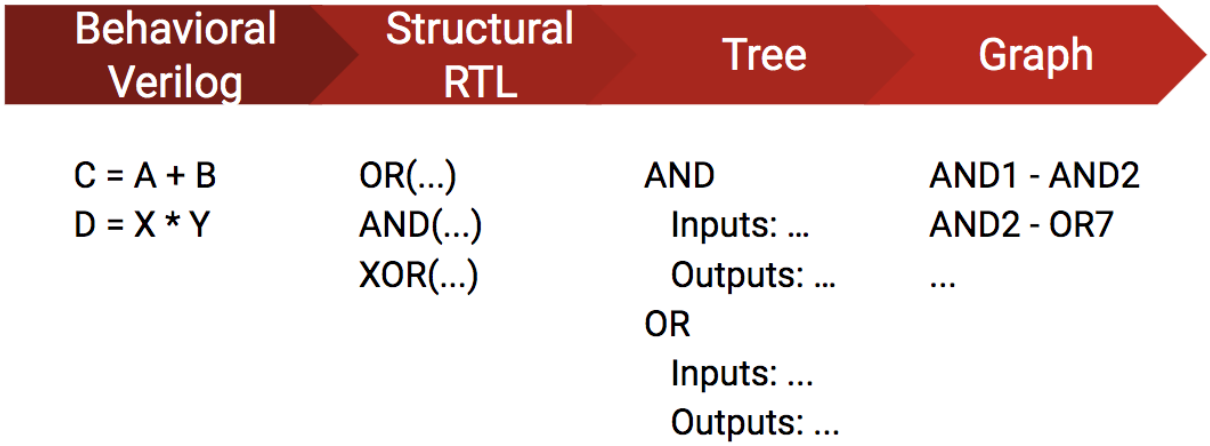


Figure 1: Steps in our Method

In the first method, after encoding the AST as a graph, we then use Node2Vec, a tool developed here at Stanford to create low dimension matrices for neural network interpretation [2]. Once the Verilog is represented as a vector, a convolutional neural network (CNN) can be used to extract features and generate a estimate for power and area. Our implementation of the CNN is in Keras at keras_predict.py. As a starting point we

used a traditional CNN with a convolutional layer, pooling layer, RELU layer, and fully connected layer with an Adam optimizer and L2 loss. This method has several drawbacks. First, it requires hand-engineering to feed in an input acceptable to a CNN. As discussed in class, this is not an optimal way to perform machine learning. Additionally, since our graphs are of varying sizes, it required large amounts of padding for the smaller designs (1000s of rows).

After trying this model, we did more research and decided to look at GCNNs. We re-encoded each graph as an adjacency matrix of (NxN) dimension a node feature matrix of (NxF), where N is the number of nodes and F is the number of features per node. To create the feature of each node, we used a string hash to represent each gate uniquely. A diagram of this is shown below Figure 1. This method provides two benefits over the previous one. First, it represents an end-to-end solution once we have generated a graph from the verilog. Second it allows us to avoid the padding problem from the first method, by utilizing disjoint batches. Disjoint batches put a batch of graphs together by linking up the adjacency matrices diagonally and stacking the feature matrices. This is shown below in Figure 2. Now we are ready to try different network models with graph convolutional layers.

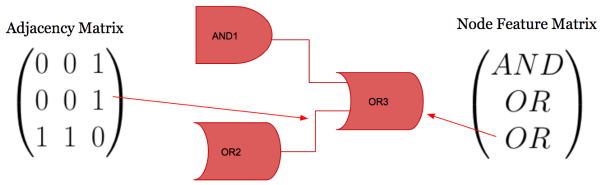


Figure 2: Adjacency Matrix and Feature Matrix

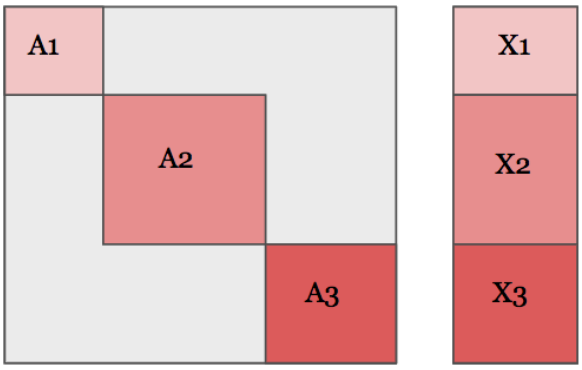


Figure 3: Disjoint Union

# 6 Models

To implement the model we used Keras as the deep learning tool and Spektral, a graph convolutional network library. We started with the most simple graph convolutional (GraphConv) layer, whose equation is below [4].

$$Z = \sigma(\tilde{A}XW + b)$$

Here $\tilde{A}$ is the normalized Laplacian of the adjacency matrix, and X is the node feature matrix. An example of a normalized Laplacian is given below. W and b, are the weights and bias respectively. We used a model with three GraphConv layers of 128 channels, 64 channels and 32 channels, a pooling layer and a dense layer. A diagram of the model is pictured below.

$$Laplacian\ Matrix = Degree\ Matrix - Adjacency\ Matrix$$

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ -1 & -1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{1}$$

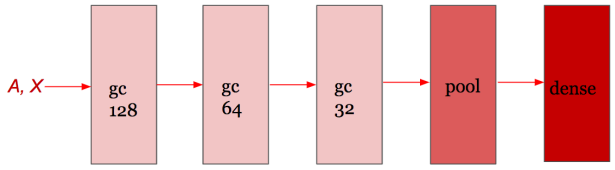


Figure 4: Baseline Model

After training this model and trying different hyperparameters we attempted to use several other graph convolutional layers provided in Spektral. We found the ARMAConv layer to work best for us [1]. The equations for the ARMAConv layer are shown below.

$$Z = 1/K \sum_{k=1}^{K} \bar{X}_k^{(T)}$$

$$\bar{X}_k^{(t+1)} = \sigma(\tilde{A}\bar{X}^{(t)}W^{(t)} + XV^{(t)})$$

Here Z is computed each layer and K is the order of the ARMA filter. $\tilde{L}$ is the normalized Laplacian of the adjacency matrix, and X is the node feature matrix. W, V, and b are trainable parameters. X is computed to be a graph convolutional skip layer. We used three layers, with the same channel sizes as above.

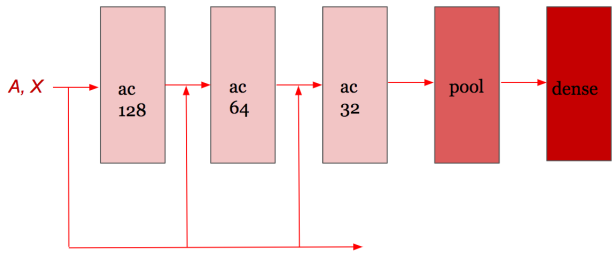


Figure 5: ARMA Conv Model

# 7 Results

Below are the results for both the baseline model as well as the ARMAConv model. When analyzing our results we noticed that for a given circuit inference performed either very well or very poorly. So to get an accurate measurement of our models performance we looked at the percentage of models within 75% accuracy.

| | Train - Area | Test - Area | Train - Power | Test - Power |
|---|---|---|---|---|
| Baseline Model | 65% | 61% | 68% | 58% |
| ARMA Model | 69% | 61% | 75% | 63% |

Table 1: Results for Baseline and ARMAConv Model

Now we can examine the speedup improvement of our model versus actually synthesizing the behavioral verilog. For this example we use an 64-bit carry lookahead adder. Our runtime for synthesis includes the entire process and our runtime for the model includes the pre-processing portion which converts the behavioral Verilog to structural Verilog (on the order of seconds) and the actual inference (on the order of millseconds). Overall, the larger a circuit design, the faster our model performs in comparison.

| | Runtime |
|---|---|
| Synthesis | 298 secs |
| Pre-processing + inference | 7.5 sec |

Table 2: Results for Baseline and ARMAConv Model

# 8 Analysis of Results

The speedup provided by using deep learning for power and area estimation would be extremely useful for hardware designers. Improving on our current model's accuracy would make this tool a useful product for digital design engineers. We found that our model would either estimate the correct power and area extremely well (within 20%) or would be incorrect by a much larger margin. As a first step in error analysis we listed out all the designs that were incorrect by a large amount. We found that most of these were circuits containing sequential logic and believe this is due to the fact that we had less sequential circuit training data compared to combinational circuit training data. Adding more sequential data is a feasible task but requires lots of time as running synthesis is a time consuming process.

# 9 Future Work

In the future, to improve our model we would like to explore the hyperparameter space even further and research more about GCNNs. There are many different layer types that have been recently published that we would like to try. Once we are able to achieve better performance on sequential circuits, we would like to expand our analysis to incorporate clock speed information into our analysis.

# 10 Contributions

The work was split up evenly in the project for research, data generation, model design, hyperparameter tuning, and report/presentation writing.

# References

[1] Filippo Maria Bianchi, Daniele Grattarola, Cesare Alippi, and Lorenzo Livi. Graph neural networks with convolutional arma filters, 2019.

[2] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[3] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216, Nov 2017.

[4] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[5] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. *Lecture Notes in Computer Science*, page 547–553, 2015.

[6] Clifford Wolf and Johann Glaser. Yosys-a free verilog synthesis suite.