Developing Synthesis Flows Without Human Knowledge

Cunxi Yu
Integrated Systems Laboratory, EPFL
Lausanne, Switzerland
cunxi.yu@epfl.ch

Houping Xiao
SUNY Buffalo
Buffalo, NY, USA
houpingx@buffalo.edu

Giovanni De Micheli
Integrated Systems Laboratory, EPFL
Lausanne, Switzerland
giovanni.demicheli@epfl.ch

ABSTRACT

Design flows are the explicit combinations of design transformations, primarily involved in synthesis, placement and routing processes, to accomplish the design of Integrated Circuits (ICs) and System-on-Chip (SoC). Mostly, the flows are developed based on the knowledge of the experts. However, due to the large search space of design flows and the increasing design complexity, developing *Intellectual Property (IP)-specific* synthesis flows providing high Quality of Result (QoR) is extremely challenging. This work presents a fully autonomous framework that artificially produces design-specific synthesis flows without human guidance and baseline flows, using Convolutional Neural Network (CNN). The demonstrations are made by successfully designing logic synthesis flows of three large scaled designs.

1 INTRODUCTION

Electronic Design Automation (EDA) involves a diverse set of software algorithms and applications that are required for the design of complex electronic systems. Given the deep design challenges that the designers are facing, developing high-quality and efficient design flows has been crucial. A well-developed design flow could reduce time-to-market by enabling manufacturability, addressing timing closure and power consumption, etc. In general, the EDA vendors provide reference design flows along with the EDA tools. However, such design flows may not perform well for many designs.

There are two major reasons. First, the performance of the design flow varies on the Intellectual Property (IP) of the design. To achieve the design objectives, design flows need to be customized for the given IP. Such flows are called *IP-specific* or *design-specific* flows. This becomes more important while new types of designs are coming out, e.g., design methods for Neuromorphic chip [1]. Second, the design flows are mostly developed by the EDA developers and users based on their knowledge and user experience, with many testing iterations and intensive supervision. However, due to a large number of available flows, finding the best design flows among the entire search space by human-testing is impossible. It is particularly difficult to find the best flows for the recently developed transformations [2]. For example, given 50 synthesis transformation that each of them can be processed independently. The total number of available design flows is $50! \approx 3 \cdot 10^{64}$. The search space of general flows is formally defined in Section 2.1. Although the significant efforts spent in providing high-quality design flows, the technique that systematically generates *IP-specific* synthesis flows has been lagging. Similarly, these problems exist in designing System-on-Chip (SoC). In Section 2 (Figure 1), two motivating examples are provided to show the needs of developing such technique.

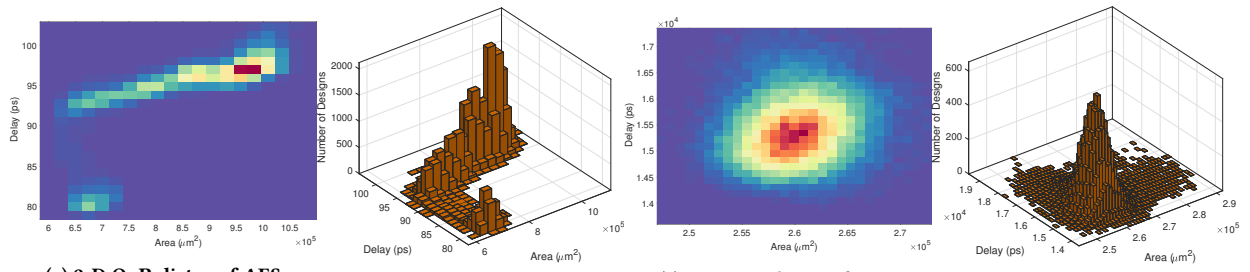
Design flows are considered as iterative flows since the transformations are applied to the designs iteratively. *Machine learning* technique has been leveraged in flow optimization, such as iterative flow optimization for compilers using *Markov Chain* [3]. Regarding synthesis flow optimization, Liu et al. recently introduced an area optimization approach for Look-up-table (LUT) mapping, in which the logic transformations are guided using *Markov Chain Monte Carlo* (MCMC) method [4]. However, Markov Chain model is not sufficient in autonomously designing synthesis flows. The main reason is that the synthesis transformation(s) may not affect the next transformation but affect the transformation several iterations later, which does not satisfy the *Markov Property* [5]. In this work, we formulate the problems of artificially developing synthesis flows as a *Multiclass classification* problem, and solved using *Deep learning* [6]. *Deep learning* has shown considerable success in tasks like image recognition [7] and natural language processing [8]. Several advances mitigate the deficiencies of traditional multilayer perceptrons (MLPs), e.g., CNNs have made it possible to robustly and automatically extract learned features; *over-fitting* is mitigated in fully connected layers using the random regularization called dropout [9].

Specifically, this paper includes the following contributions: **a)** The search space of artificially developing synthesis flows is formally defined in Section 2. **b)** We introduce a flow-classification model (Section 3.1) combining with the one-hot modeling of flows (Section 3.2), such that the problem can be modeled as *Multiclass Classfication* problem. **c)** We develop a fully autonomous framework for developing synthesis flows based on Convolutional Neural Network (CNN). This framework takes HDL as input and output two sets of synthesis flows, namely *angel*-flows and *devil*-flows that provide the best and worst QoRs respectively[1]. **d)** Our framework is demonstrated by successively developing delay-driven and area-driven *angel/devil*-flows for 64-bit Montgomery Multiplier, 128-bit AES core and 64-bit ALU. Evaluations of the CNN architecture and training process for classifying synthesis flows are also provided. **e)** The datasets and demos are released publicly[2].

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196026

[1] *devil*-flows could provide information for improving the synthesis transformations.
[2] https://github.com/ycunxi/FLowGen-CNNs-DAC18.git



(a) 2-D QoR distro of AES (b) 3-D QoR distro of AES (c) 2-D QoR distro of ALU (d) 3-D QoR distro of ALU

Figure 1: Delay and area results of the 50,000 random ABC synthesis flows of 128-bit AES core and 64-bit ALU.

2 BACKGROUND

2.1 Notations and Search Space

Definition 1 ***none*-repetition Synthesis Flow**: *Given a set of unique synthesis transformations* $\mathbb{S}$=$\{p_0, p_1, ..., p_n\}$, *a synthesis flow* $\mathbb{F}$ *is a permutation of* $p_i \in \mathbb{S}$ *performed iteratively.*

Example 1: Let $\mathbb{S}$=$\{p_0, p_1, p_2\}$. p_i are the transformations in the synthesis tools and can be processed independently. Then, there are totally six flows available:

$$F_0 : p_0 \rightarrow p_1 \rightarrow p_2 \; F_1 : p_0 \rightarrow p_2 \rightarrow p_1$$
$$F_2 : p_1 \rightarrow p_0 \rightarrow p_2 \; F_3 : p_1 \rightarrow p_2 \rightarrow p_0$$
$$F_4 : p_2 \rightarrow p_0 \rightarrow p_1 \; F_5 : p_2 \rightarrow p_1 \rightarrow p_0$$

Remark 1: *Let* $\mathbb{N}$ *be the number of all available flows, where* $\mathbb{S}$ *includes n elements, such that* $\mathbb{N} \leq n!$.

The upper bound of $\mathbb{N}$ happens *iff* all elements in $\mathbb{S}$ can be processed independently. In practice, there could be some constraints have to be satisfied for processing these transformations. In this case, $\mathbb{N}$ will be smaller than $n!$. For example, given a constraint that p_1 has to be processed before p_2, the available flows include only F_0, F_2, and F_3.

Definition 2 ***m*-repetition Synthesis Flow (m≥2)**: *Given a set of unique synthesis transformations* $\mathbb{S}$=$\{p_0, p_1, ..., p_n\}$, *a synthesis flow with m-repetition* $\mathbb{F}_m$ *is a permutation of* $p_i \in \mathbb{S}_m$, *where* $\mathbb{S}_m$ *contains m* $\mathbb{S}$ *sets.*

Example 2: Let $\mathbb{S}$=$\{p_0, p_1\}$. Each p_i can be processed independently. For developing 2-repetition synthesis flows, $\mathbb{S}_2$=$\{p_0, p_1, p_0, p_1\}$. The available flows include:

$$F_0 : p_0 \rightarrow p_0 \rightarrow p_1 \rightarrow p_1 \; F_1 : p_1 \rightarrow p_1 \rightarrow p_0 \rightarrow p_0$$
$$F_2 : p_0 \rightarrow p_1 \rightarrow p_0 \rightarrow p_1 \; F_3 : p_1 \rightarrow p_0 \rightarrow p_1 \rightarrow p_0$$
$$F_4 : p_0 \rightarrow p_1 \rightarrow p_1 \rightarrow p_0 \; F_5 : p_1 \rightarrow p_0 \rightarrow p_0 \rightarrow p_1$$

Remark 2: *Let* $\mathbb{L}$ *be the length of a synthesis flow. Given a m-repetition* $\mathbb{F}_m$ *with n transformations in* $\mathbb{S}$, $\mathbb{L} = n \times m$.

Remark 3: *Let function* $f(n, \mathbb{L}, m)$ *be the number of available m-repetition flows with n elements in S.* $f(n, \mathbb{L}, m)$ *uniquely satisfies the following recursive formula :*

$$f(n, \mathbb{L}+1, m) = nf(n, \mathbb{L}, m) - n\binom{\mathbb{L}}{m} f(n-1, \mathbb{L}-m, m)$$

The number of available m-repetition flows with n synthesis transformations is the same as counting $\mathbb{L}$-*permutations* of n objects. The proof of the recursive formula is similar to [10] that will not be included in this paper. The upper and lower boundary conditions are $n! < f(n, \mathbb{L}, m) < n^{\mathbb{L}}$. We can see that $f(n, \mathbb{L}, m)$ becomes dramatically larger than $n!$ (*non*-repetition flows) as m increasing.

2.2 Motivating Example

We provide two motivating examples using the Open-source logic synthesis framework ABC [11] shown in Figure 1. The setups are as follows:

- $\mathbb{S}$={balance, restructure, rewrite, refactor, rewrite -z, refactor -z} (n=6); the elements in $\mathbb{S}$ are logic transformations in ABC[3] that can be processed independently.
- 50,000 unique 4-repetition flows are generated by random permutations of $\mathbb{S}_4$ (m=4, n=6, $\mathbb{L}$=24).
- Input designs: 128-bit Advanced Encryption Standard (AES) core, and 64-bit ALU taken from OpenCore [12].
- Delay and area of these flows are obtained after technology mapping using a 14nm standard-cell library.

The QoR distributions of AES and ALU designs using the 50,000 random flows are shown in Figure 1-(a, b) and (c, d). There are several important observations based on Figure 1, which show the main motivations of this work:

- Given the same set of synthesis transformations, the QoR is very different using different flows. For example, delay and area of AES design produced by the 50,000 flows have up to 40% and 90% difference, respectively.
- The search space of the synthesis flows is large. According to Remark 3, the total number of available 4-repetition flows with n = 6 independent synthesis transformations is more than 10^{16}. Discovering the high-quality synthesis flows with human-testing among the entire search space is unlikely to be achieved.
- The same set of flows perform differently on different designs. For example, in Figure 1, QoR distributions of AES and ALU are *statistically significant*. This means that the high-quality flows for AES design could perform poorly for ALU. Therefore, synthesis flows need to be customized for specific IP or design.

3 APPROACH

3.1 Overview

This section presents our framework that artificially develops synthesis flows for a given design. Our framework takes the HDL as input and outputs two sets of synthesis flows, namely *angel-flows* and *devil-flows*, which provide the best and worst QoR according to the design objectives. This problem is formulated as *Multiclass Classification* and solved using CNN classifier. The main idea of our approach is that training a CNN Classifier with a small set of *labeled*

[3]The names of these transformations are the same as the commands in ABC.

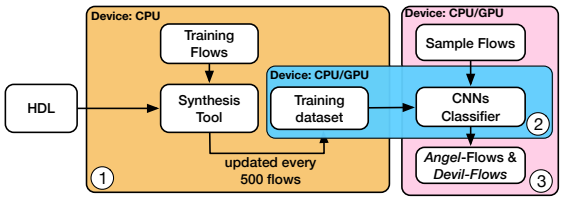


Figure 2: Overview of the proposed framework, performing in sequence ① → ② → ③.

random flows. The classes (or labels) of the synthesis flows are labeled based on one or multiple QoR metrics, such as delay, area, power, etc. The trained classifier is used to predict the classes of a large number of *unlabeled* random flows. Finally, *angel-flows* and *devil-flows* are generated by sorting the *prediction confidence*, i.e., the probability to be in a certain class (Section 3.3). This framework is a generic model for designing synthesis flows in many stages, such as High-level synthesis and logic synthesis. The demonstration is made by designing logic synthesis flows using ABC [11] shown in Section 5. The flow of our framework is shown in Figure 2, including three main components:

① **Generate training datasets.** In this work, the training dataset is a set of *labeled* synthesis flows, namely *training flows*. However, the training flows are originally *unlabeled*. This first step of our approach is labeling a set of random flows. This requires applying these synthesis flows to the input design and collecting the QoR result at the end of each flow. Note that applying a synthesis flow to a large design could be time-consuming. Hence, our framework is performed in an incremental fashion. The CNN training (component ②) starts after 1000 labeled flows collected, and it will be re-trained every 500 new labeled flows collected. In this case, our framework can produce the intermediate results during the training process.

These flows will be labeled according to the classification model shown in Table 1. This model can be changed according to the design objectives, using either a single-metric or multi-metric model. For example, if the design objective is area optimization, a single-metric model will be selected where r is the area metric. If the design objectives are minimizing delay with a given area budget, a multi-metric model will be selected. Note that the *number of classes* $(n+1)$ is a fixed input of the proposed framework, and the definition (QoR range) of each class is decided using a general model. For example, to define seven classes (n=6) in a single-metric model, it requires six determinators, $\{x_0, x_1, ..., x_n\}$. We define the six determinators using the {5%,15%,40%,65%,90%,95%} QoR results of collected labeled synthesis flows. For example, assuming 1000 labeled flows collected, x_0 is the 50^{th} least value of the select metric and x_6 is the 50^{th} largest value. Since the training dataset is updating incrementally, the definitions of classes may change dynamically. *Angel-flows* and *devil-flows* are the subset of the flows corresponding to classes **0** and **n**.

② **Design and train CNN Classifier**. The second component is training a CNN classifier that predicts the classes of unlabeled flows. To train a CNN classifier, the training data, i.e., labeled synthesis flows, need to be represented in the matrix. We present a

Table 1: Labeling the training flows based on synthesis QoR. r and r_i are the QoR metrics such as delay, area, power, etc.

Single-metric	Multi-metric	Class/Label
$r \le x_0$	$r_0 \le x_0, r_1 \le y_0$	*0*
$x_0 < r \le x_1$	$x_0 < r_0 \le x_1, y_0 < r_1 \le y_1$	*1*
$x_1 < r \le x_2$	$x_1 < r_0 \le x_2, y_1 < r_2 \le y_2$	*2*
...	...	...
$r > x_n$	$r_0 > x_n, r_1 > y_n$	*n*

one-hot modeling that represents synthesis flow in *binary matrix*. This model and the CNN architecture are introduced in Section 3.2.

③ **Output Angel-flows and Devil-flows**. The trained classifier will be used to predict the classes of a large number of *un-tested* sample flows. Although we are only interested in the flows in classes **0** and **n**, the classifier may label many flows in these two classes. However, for the synthesis perspective, selecting a small set of flows is sufficient. In this work, the *angel-flows* and *devil-flows* are selected from the flows labeled with **0** and **n** with highest *prediction confidence*. The details are included in Section 3.3.

3.2 CNN Classifier

3.2.1 One-hot Representation of Synthesis Flow. In this section, the one-hot representation model of synthesis flow is introduced for m-repetition flow. The *non*-repetition flow can be represented using the same model.

Let $\mathbb{M}$ be the binary matrix of a m-repetition flow F with $\mathbb{S}=\{p_0, p_1,..., p_n\}$ (see Definition 2). The number of transformations in F equals to the length of the flow $\mathbb{L}=n \times m$ (see Remark 2). Let the j^{th} synthesis transformation in F be p_i, $j \le \mathbb{L}$, $i \le n$. Its n-by-1 binary vector representation is $\mathbb{V}_j$, where i^{th} element is 1 and the other elements are 0. $\mathbb{M}$ is an $\mathbb{L}$-by-n matrix such that its j^{th} row is $\mathbb{V}_j$.

Example 3: We illustrate the one-hot representation model using flow F_0 shown in Example 2, such that $\mathbb{S}=\{p_0, p_1\}$ and $F=p_0 \rightarrow p_0 \rightarrow p_1 \rightarrow p_1$, $\mathbb{M}$ is an 4-by-2 matrix.

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

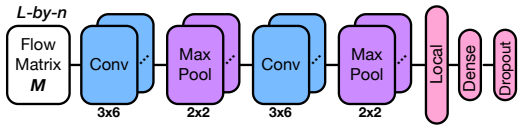


Figure 3: CNN architecture used for synthesis flows classification.

3.2.2 CNN Architecture and Training. The input of the CNN are $\mathbb{L}$-by-n binary matrices representing the synthesis flows. The CNN includes convolution, pooling, locally connected, dense and dropout layers. The kernel size of the convolutional and pooling layers are shown in Figure 3. The dropout rate in the dropout layer is 0.4 to prevent the overfitting problem [9]. Since our inputs are in one-hot representation, the loss function is computed using *sparse softmax cross entropy* function. The output of the network comes from *softmax* function. The number of kernels (filters) of

convolutional layers are 200. The stride size of the convolutional and pooling layers are 1×1.

Regarding the CNN architecture, two parameters have significant impacts on the prediction performance (accuracy): **a)** kernel size of convolutional layers and **b)** activation functions of convolutional and dense layers. Unlike most of the CNN classification applications, the n-by-n kernel size does not perform well in classifying synthesis flows. We use $n \times 2n$ kernel size in this work. The reason is that there is only one non-zero element in each row of $\mathbb{M}$. Using $n \times 2n$ kernel could avoid computations over zero-matrix. The results of comparing the accuracy of the CNN classifier using 3×6, 6×6, and 6×12 kernels are shown in Section 5 Figure 6.

The activation function of the nodes in the neural network defines the output of the nodes with a given set of inputs. In artificial neural networks, this function is also called the transfer function. The activation operations should provide different types of nonlinearities in the neural networks to solve Multiclass Classification problems. In general, there are two types of activation functions, including smooth nonlinear functions, such as *Sigmoid, Tanh, Exponential Linear Units* (ELU) [13], *Scaled Exponential Linear Units (SELU)*[14], etc., and smooth continuous functions, such as *Rectified linear unit (ReLU)* [15], *Concatenated Rectified Linear Units (CReLu)*[16], etc. We find that for classifying synthesis flows, the activation functions with nonlinearities perform better, such SELU and Tanh. The activation functions including ReLU, ReLU6, ELU, SELU, Softplus, Softsign, Sigmoid and Tanh, have been compared in Section 5 Figure 7.

Regarding the training process, the CNN classifier is trained specifically for each design as described in Section 3.1. Since the training data are collected incrementally, the CNN will be re-trained after every 500 new data points collected. The Mini-Batch [17] training strategy is applied in this work with batch size 5, i.e., simultaneously evaluated five training examples in each iteration. In this work, we have evaluated five different gradient descent algorithms, including Stochastic gradient descent (SGD), Momentum [18], AdaGrad [19], RMSProp [20], and Follow the regularised leader (FTRL) [21]. The comparison result is included in Section 5 Figures 4 and 5.

3.3 Angel-Flows and Devil-Flows

In this work, the outputs of the proposed framework are 200 *angel*-flows and 200 *devil*-flows. There are two steps for generating these flows. First, it uses the trained CNN classifier to predict the class of a large number of random flows. According to the classification rule (Table 1), the *angel*-flows and *devil*-flows will be selected from the 0-class flows and n-class flows. The predicted class of a random flow is the class corresponding to the highest probability in the result of the CNN classifier coming from *softmax* function. For example, assuming the output of the classifier (# classes = 7) is $\{p_0 = 0.47, p_1 = 0.13, p_2 = 0.22, p_3 = 0.02, p_4 = 0.03, p_5 = 0.12, p_6 = 0.01\}$, where p_i is the probability of a flow being class-i, then the predicted class is class-0. To minimize the errors in selecting the *angel*(*devil*)-flows, our framework selects the flows with highest $p(0)(p(n))$ within the class-0(class-n) flows.

Example 4: Let the prediction results in Table 2 be the prediction outputs of the CNN classifier of four synthesis flows. If two *angel*-flows are required, F_0 and F_1 are selected and F_4 is eliminated.

Table 2: Example of finalizing the *angel*-flows.

Flow	p_0	p_1	p_2	p_3	p_4	p_5	p_6
F_0	**0.47**	0.13	0.22	0.02	0.03	0.12	0.01
F_1	**0.51**	0.12	0.01	0.09	0.17	0.08	0.02
F_2	0.02	**0.45**	0.14	0.12	0.11	0.10	0.06
F_3	0.12	0.03	0.17	**0.62**	0.01	0.02	0.03
F_4	**0.35**	0.23	0.09	0.02	0.13	0.17	0.01

4 EXPERIMENTAL RESULTS

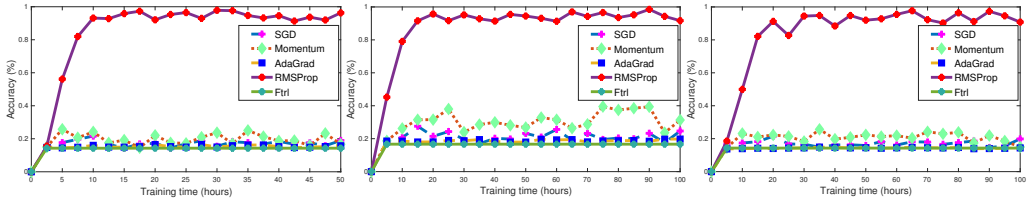
We demonstrate the proposed framework by designing logic synthesis flows Open-source synthesis framework ABC [11]. Our framework is implemented in C++. The CNN classifier is implemented using Tensorflow r1.3 [22] using its C++API. The demonstration is made with three designs, including 64-bit Montgomery multiplier, 128-bit AES core [12], and 64-bit ALU [12]. The goal is to generate 200 *angel*-flows and 200 *devil*-flows for area or delay optimization. We use the same setups shown in the motivating example (Section 2.2). Thus, the synthesis flows will be 4-repetition flows with six ABC synthesis transformations, $\mathbb{S}$={balance, restructure, rewrite, refactor, rewrite -z, refactor -z}. The inputs of CNN classifier are 24-by-6 matrices representing the synthesis flows using the one-hot modeling. These matrices are re-shaped to 12-by-12 matrices for using two convolutional layers.

For generating the area- or delay-driven flows, we use the single-metric classification model (Table 1) where r is the area/delay of the design. The number of classes is seven. The six determinators are defined using { 5%, 15%, 40%, 65%, 90%, 95% } of the area/delay results of the training flows. The area and delay results are obtained after technology mapping with 14nm standard-cell library. The number of training flows is 10,000 and the number of sample flows for generating the final flows is 100,000. The experimental results are obtained using a machine with Intel Xeon 2x12cores@2.5 GHz, 256GB RAM, 2x240GB SSD and 2 Nvidia Titan X GPUs.

The result section includes two parts. The first part contains the experimental results of training the CNN classifier. It consists of the evaluations of different gradient descent algorithms, various of convolutional kernel sizes and activation functions. Based on these results, we find the best settings for the CNN architecture and training strategy. Using these setting, we generate and evaluate the quality of generated *angel*-flows and the *devil*-flows. To evaluate the accuracy of the CNN classifier and the generated flows, we have explicitly collected the area and delay result by applying the 100,000 flows to the three designs. Hence, the true classes of the 100,000 sample flows are available for evaluation.

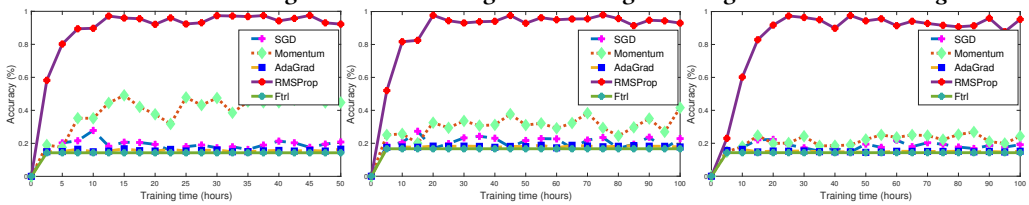
4.1 Results of Training CNN Classifier

4.1.1 Gradient Descent Algorithms. The results of training the CNN classifier using different gradient descent algorithms are shown in Figures 4 and 5. Figure 4 includes the results of training for generating area-driven flows using five different algorithms, including Stochastic gradient descent (SGD), Momentum [18], AdaGrad [19], RMSProp [20], and Follow the regularised leader (FTRL) [21]; Figure 5 includes results of generating delay-driven flows. The learning rate η=0.0001 and number of training steps is 100,000. The kernel size of convolutional layers is 6-by-12. In Figures 4 and 5, the y-axis represents the accuracy of prediction. Let N_{angel} be the number of generated *angel*-flows that their true class is class-0; let N_{devil} be the number of generated *angel*-flows that their true class



(a) Montgomery Multiplier (b) AES Core (c) ALU

Figure 4: Evaluation of different gradient descent algorithms for generating the area-driven *angel*/*devil* flows.



(a) Montgomery Multiplier (b) AES Core (c) ALU

Figure 5: Evaluation of different gradient descent algorithms for generating the delay-driven *angel*/*devil* flows.

is class-6. The accuracy is defined as following:

$$accuracy = \{(N_{angel} + N_{devil})/2 \times 200\}/2$$

The x-axis represents the training time of our framework. Note that the training process of the 64-bit Montgomery multiplier is 2× faster than the other two designs. The reasons is that collecting the training dataset takes most of the runtime. The runtime of applying one synthesis flow to Montgomery multiplier is about 2× faster than the other two. The actually runtime for training the CNN classifier is about 3 - 5% of the entire training time. As shown in Figures 4 and 5, the RMSProp [20] outperforms other algorithms in classifying synthesis flows. The accuracy of the classifier in these six experiments reaches 95% after 24 hours.

4.1.2 Choice of Convolutional Kernel Size. As mentioned in Section 3.2, the size of the convolutional layer kernel has significant impacts on the CNN classifier. In Figure 6, three kernel sizes, 3×6, 6×12, have been tested using RMSProp algorithm [20], where the learning rate η=0.0001 and number of training steps is 100,000. The number of kernels at each convolutional layer is 200. The input design is the 128-bit AES core, and the objective is generating delay-driven flows. We can see that the kernel with size $n \times 2n$ (3×6, 6×12) perform much better than the $n \times n$ kernel (6×6).

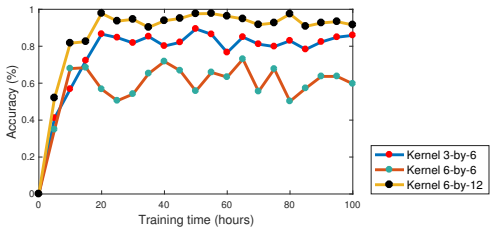


Figure 6: Evaluation of three convolutional kernels. Test case: generating delay-driven flows for the 128-bit AES core.

4.1.3 Evaluation of Activation Functions. For evaluating the performance of classifying synthesis flows using different activation functions, we set the learning rate η=0.0001, learning steps=100,000, convolutional kernel size is 6×12, and use RMSProp to minimize the loss function. Figure 7 includes the comparison of eight different activation functions, including ReLU, ReLU6, ELU[13], SELU[14], Softplus, Softsign, Sigmoid and Tanh. We can see that the ELU, SELU, Softsign and Tanh functions outperform the others, and SELU offers the best accuracy for generating delay-driven flows for the 128-bit AES core. Note that the accuracy of different activation functions varies on different datasets. In this work, SELU provides most reliable performance.

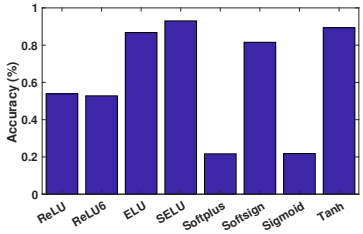
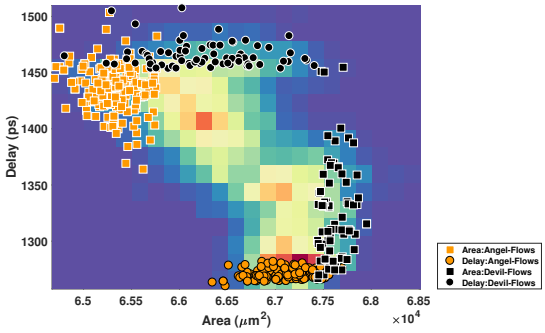


Figure 7: Evaluation of different activation functions. Test case: generating delay-driven flows for the 128-bit AES core.

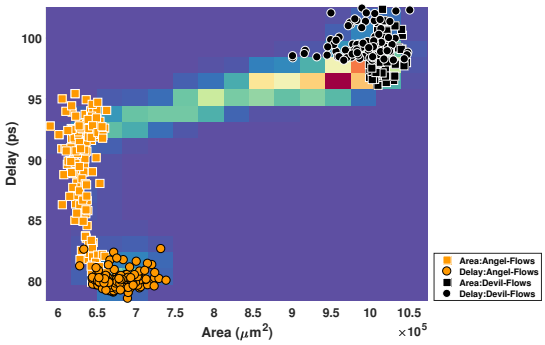
4.2 Quality of Generated Flows

Finally, we evaluate the quality of the generated *angel*-flows and *devil*-flows. The results shown in Figure 8 are obtained using the following settings: number of training flows is 10,000; number of sample flows is 100,000; η=0.0001; learning steps is 100,000; activation function is SELU; gradient descent algorithm is RMSProp; convolutional kernel size is 6×12. The four types of points shown in Figure 8 represent the area-delay result of *area-angel-flows*, *area-devil-flows*, *delay-angel-flows* and *delay-angel-flows*. The y-axis represents delay and x-axis represents area. The background of each

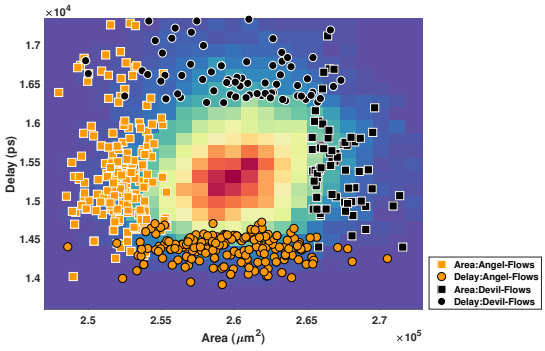
sub-figures in Figure 8 is the 2-D distribution of the 100,000 sample flows[4]. We can see that the generated area(delay) *angel*-flows provide the best results in terms of area(delay), and the *devil*-flows provide the worst results, among the 100,000 sample flows. For example, the data points of area-angel-flows of these three designs are clearly bounded with a certain area value. The total runtime for generating these flows takes 3-4 days. It is demonstrated that our framework can successively develop *angel*-flows and *devil*-flows.



(a) Flows generated for 64-bit Montgomery multiplier.



(b) Flows generated for 128-bit AES core.



(c) Flows generated for 64-bit ALU.

Figure 8: Quality of the generated ABC synthesis flows for 64-bit Montgomery multiplier, 128-bit AES and 64-bit ALU.

[4]The 2-D distribution represents the distribution similarly to Section 2.2 Figure 1, but with 100,000 data points.

5 CONCLUSIONS

This work presents a fully autonomous framework that artificially produces *design-specific* synthesis flows without human guidance and baseline flows. We introduce a general approach for flow optimization problems by modeling into *Multiclass Classification*. The one-hot modeling of iterative flows is proposed such that any flow can be represented using binary matrix. This approach is demonstrated by generating the best, and worst synthesis flows, using three large designs with 14nm technology. The future work will focus on artificially developing cross-layer synthesis flows to find the missing-correlations between logic and physical designs [23].

6 ACKNOWLEDGEMENT

This project is funded by ERC-2014-AdG 669354 grant.

REFERENCES

[1] F. Akopyan, J. Sawada *et al.*, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.
[2] C. Yu, M. J. Ciesielski, M. Choudhury, and A. Sullivan, "Dag-aware logic synthesis of datapaths," in *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, 2016, pp. 135:1–135:6.
[3] F. Agakov, E. Bonilla, *et al.*, "Using machine learning to focus iterative optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 2006, pp. 295–305.
[4] G. Liu and Z. Zhang, "A parallelized iterative improvement approach to area optimization for lut-based technology mapping," *FPGA'17*, 2017.
[5] R. Durrett, *Probability: theory and examples*. Cambridge university press, 2010.
[6] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
[8] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1915–1929, 2013.
[9] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.
[10] H. Mendelson, "On permutations with limited repetition," *Journal of Combinatorial Theory, Series A*, vol. 30, no. 3, pp. 351–353, 1981.
[11] A. Mishchenko *et al.*, "ABC: A System for Sequential Synthesis and Verification," *URL http://www. eecs. berkeley. edu/alanmi/abc*.
[12] "OpenCores," *URL https://opencores.org*.
[13] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.
[14] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," *arXiv preprint arXiv:1706.02515*, 2017.
[15] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.
[16] W. Shang, K. Sohn, D. Almeida, and H. Lee, "Understanding and improving convolutional neural networks via concatenated rectified linear units," in *International Conference on Machine Learning*, 2016, pp. 2217–2225.
[17] G. B. Orr and K.-R. Müller, *Neural networks: tricks of the trade*. Springer, 2003.
[18] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.
[19] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.
[20] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.
[21] H. B. McMahan, Holt *et al.*, "Ad click prediction: a view from the trenches," in *KDD'13*. ACM, 2013, pp. 1222–1230.
[22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.
[23] C. Yu, M. Choudhury, A. Sullivan, and M. J. Ciesielski, "Advanced datapath synthesis using graph isomorphism," in *ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*, 2017, pp. 424–429.