OpeNPDN: Neural Networks for Automated Power Delivery Network Synthesis

Vidya A. Chhabria[1], Andrew B. Kahng[2], Minsoo Kim[2], Uday Mallappa[2],
Sachin S. Sapatnekar[1], and Bangqi Xu[2]
[1]University of Minnesota, [2]University of California, San Diego

Abstract—**Designing an optimal power delivery network (PDN) is a time-intensive task that involves many iterations. This paper describes, OpeNPDN, a publicly available software that relies on a library of predesigned, stitchable templates, and uses machine learning (ML) to rapidly build a PDN with region-wise uniform pitches based on these templates. The software is designed to be used at the placement stage of a standard design flow methodology, where an optimized PDN is synthesized using a convolution neural network, based on current and congestion distributions available. The neural network builds a safe-by-construction PDN that meets static IR drop and electromigration (EM) specifications. The optimization of the PDN improves congestion in the design by saving thousands of routing tracks in congestion-critical regions, when compared to a globally uniform PDN, while staying within the IR drop and EM limits.**

Fig. 1. (a) A set of 3 templates different PDN utilizations and (b) a template-based PDN with piece-wise uniform pitches.

I. Introduction

OpeNPDN [1], is an open source, ML-based framework for optimized PDN design. This software overcomes the expensive, iterative nature of optimized PDN design by employing a convolution neural network (CNN) to synthesize a safe-PDN. This helps enable a no-human-in-loop, correct-by-construction design methodology, with a 24 our turn-around-time.

In OpeNPDN, synthesis is performed based on a predefined, technology-specific set of templates. These templates are defined across multiple layers and vary in their metal utilization in the intermediate layers. Fig. 1(a) shows three templates that are defined across multiple user-specified layers and differ in their metal utilization. These are specified based on a combination of metal densities of the PDN each layer. Templates with higher metal utilization (dense) are good for power integrity but bad for congestion. Therefore, OpeNPDN attempts to solve the optimization problem of using minimal possible routing resources while still meeting stringent IR drop and EM constraints, by using machine learning to rapidly synthesize a correct PDN. OpeNPDN translates the optimization problem into one of finding a template in every region on the tiling of a chip. The problem is solved as a classification problem using a trained convolutions neural network (CNN). For example, Fig. 1(b) depicts a chip with tessellated into 4 regions and a trained CNN answers the question — "which template goes into which region?" based on the current and congestion estimates in the region and its neighborhood. Using the CNN, for a specific PDK and region size, a correct-by-construction PDN can be rapidly synthesized for any design.

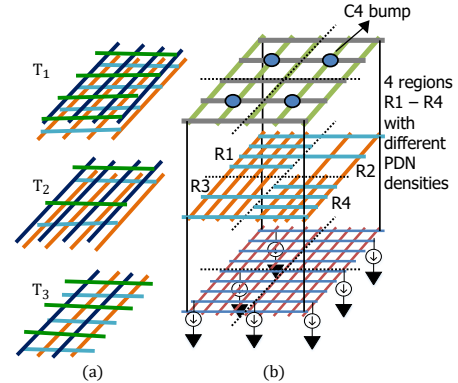In the OpenROAD flow [3], OpeNPDN is used as a tool that optimizes a synthesized PDN by depopulating stripes based on the placed design's current and congestion estimates. Fig. 2 (a) demonstrates how the trained model can be deployed into any design flow at the placement stage. The trained model, takes the following inputs:

- a placed design in standard DEF format [4]
- a per-instance-based power report from OpenSTA [5]
- a congestion report of the design

and generates the following outputs:

- a representation of the CNN-synthesized safe-PDN
- an IR drop map of the design, using the synthesized PDN
- a static IR drop report which states if the CNN-synthesized PDN is "safe" for the design
- a congestion improvement report that provides the number of tracks saved by using the piece-wise regular, template-based PDN when compared to a uniform PDN

In the following sections, we explain the steps to use OpeNPDN with details on the input and output specification for both the training and inference flows. A detailed explanation on the algorithms employed in OpeNPDN can be found in [2].

II. OpeNPDN User Guide

As a part of The-OpenROAD-Project organization, OpeNPDN is publically available on GitHub [1]. An overview of the complete OpeNPDN framework, implemented in Python3.6 using Numpy, Scipy, and TensorFlow packages, is shown in Fig. 2. Similar to standard supervised ML algorithms, this framework has two flows, (i) an inference flow (Fig. 2 (a)) which uses a trained CNN to rapidly synthesize PDN, for any design, provided a set of templates, and (ii) a one-time training
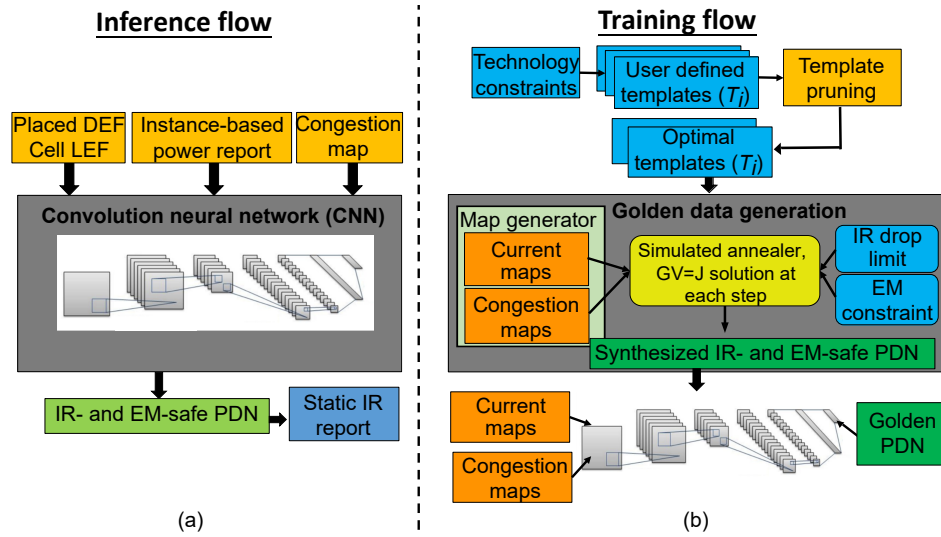
Fig. 2. The OpeNPDN framework: (a) inference flow which uses a CNN to rapidly synthesizes an optimized IR- and EM-safe PDN, and (b) a *one-time* technology and template specific training flow.

flow (Fig. 2 (b)) which includes the golden data generation and CNN model training itself. OpeNPDN has the following prerequisites:

- python 3.6
- pip 18.1
- python3-venv

OpeNPDN best runs on Red Hat Enterprise, Ubuntu or CentOS Linux distributions. Additionally, please refer to requirements.txt file in the repository for other dependencies. The packages in requirements.txt will be automatically installed in a virtual environment during build process.

A. Download and install

OpeNPDN can be downloaded and installed from the GitHub repository using the instructions in Listing 1.

```
1 git clone https://github.com/The-OpenROAD-Project/
      OpeNPDN.git
2 cd OpeNPDN
3 make clean
4 source install.sh
5 make build
6 make testing
```

Listing 1. Download and install OpeNPDN.

The install script (install.sh), shown on line 4 of Listing 1, creates a virtual python environment and installs all the necessary packages, mentioned in the requirements.txt file, using pip. At the build stage (line 5 of Listing 1), OpeNPDN exposes any existing CNN checkpoints and the golden training data necessary to train the CNN (line 6). The testing phase (line 6) runs a set of unit tests, using pytest, to check if the installation and build have been successful.

B. Tool Usage

As a ML-based framework, OpeNPDN has two flows (i) the training flow, a *one-time* step that is performed at the beginning for a specific technology, and (ii) the inference flow, a trained CNN is deployed into the design flow (Fig. 2).

```
1  pdn specify_grid stdcell {
2     name low
3     rails metal1
4     size {100.0 100.0}
5     layers {
6        metal1 {width 0.1 pitch  2.5 offset 0}
7        metal4 {width 0.5 pitch  50.0 offset 2}
8        metal7 {width 1.5 pitch 50.0 offset 2}
9     }
10    connect {{metal1 metal4} {metal4 metal7}}
11 }
12
13 pdn specify_grid stdcell {
14    name medium
15    rails metal1
16    size {100.8 100.0}
17    layers {
18       metal1 {width 0.1 pitch  2.5 offset 0}
19       metal4 {width 0.5 pitch 25.0 offset 2}
20       metal7 {width 1.5 pitch 50.0 offset 2}
21    }
22    connect {{metal1 metal4} {metal4 metal7}}
23 }
24
25 pdn specify_grid stdcell {
26    name high
27    rails metal1
28    size {100.0 100.0}
29    layers {
30       metal1 {width 0.1 pitch  2.5 offset 0}
31       metal4 {width 0.5 pitch 12.5 offset 2}
32       metal7 {width 1.5 pitch 50.0 offset 2}
33    }
34    connect {{metal1 metal4} {metal4 metal7}}
35 }
36
37 pdn specify_grid stdcell {
38    name upperGrid
39    layers {
40       metal8 {width 1.5 pitch 10.0 offset 0}
41       metal9 {width 1.5 pitch 10.0 offset 2}
42    }
43    connect {{metal7 metal8} {metal8 metal9}}
44 }
```

Listing 2. PDN template definition file with three templates.

1) Training flow: The training flow requires two inputs (i) template definition file (template_definition.cfg), an example is shown in Listing 2, and (ii) technology specific file config-

uration file (tech_spec.json), an example is shown in Listing 3, to generate a trained, technology-specific CNN. The entire training flow, including the template definition, golden-data generation, and CNN-training can be run using a simple "*make train*" in the shell.

```
1  {
2      "property": {
3          "NUM_layers": 8,
4          "TECH_layers": ["M1", "M2", "M3", "M4",
5                          "M5", "M6", "M7", "M8"],
6          "VDD": 1.1,
7          "IR_DROP_LIMIT": 0.010,
8      },
9      "layers": {
10         "M1": {
11             "min_width": 0.08e-6,
12             "via_res": 5.0,
13             "res": 0.38,
14             "direction": "H",
15             "t_spacing": 0.1e-06
16         },
17         "M2": {
18             "min_width": 0.08e-06,
19             "via_res": 5.0,
20             "res": 0.25,
21             "direction": "V",
22             "t_spacing": 0.1e-06
23         },
```

Listing 3.   Technology specification file.

PDN template set definition: PDN templates are defined by the user in a JavaScript Object Notation (JSON) format as a combination of different pitches in each metal layer of the PDN, while following DRC rules for the specific technology. A critical requirement of these templates is their *stitchability*, i.e., if two templates are placed adjacent to each other, then they should align at the edges. Therefore, it is important to avoid choosing template pitches that are co-prime. An example of the template definition file is shown in Listing 2.

The create_template.py script in the repository builds the corresponding resistor networks for each of these templates and stores them as python objects in the templates directory. These objects can be reused during inference. A template elimination script (template_elimination.py) is used to prune any sub-optimal templates (templates which are worse for both power integrity and congestion).

"Golden" training data generation: The script run_batch.py runs a simulated annealing (SA) engine to generate near-optimal training data. The script spawns multiple subprocesses in parallel based on the available compute resources. The parameters including the number of data points, number of parallel processes can be configured in the tool_config.json file. The SA engine produces the golden data using the simulated_annealer.py script and the run_batch.py script prepares the data for training the CNN.

CNN training: The CNN is built using TensorFlow Framework. Based on the compute resources available the software supports training on either a CPU or a GPU. All the hyper parameters that are used to train the CNN are configurable by the user and are defined in a tool configuration file (tool_config.json). If left unchanged, the default settings are used. The CNN is trained using the cnn_train.py script.

2) Inference flow: The inference flow requires the following inputs to synthesize a template-based PDN:

- a placed design in standard DEF format [4]
- a per-instance-based power report from OpenSTA [5]
- a congestion report of the design in the format specified in the congestion_format.txt file
- the template definition file (same as that with which the CNN was trained)

The paths to the above inputs are specified in the Makefile. A "*make inference*" command in the shell, loads the trained CNN and the specified inputs to generate the following outputs:

- a template map which represents the name of the PDN and its location on the chip (Fig. 3(a))
- an IR drop report (IR_drop.rpt), checks if CNN-synthesized grid meets IR drop specification (Fig. 3(b))
- an IR drop map (IR_map.png), a graphic showing the IR drop distribution (Fig. 3(c))
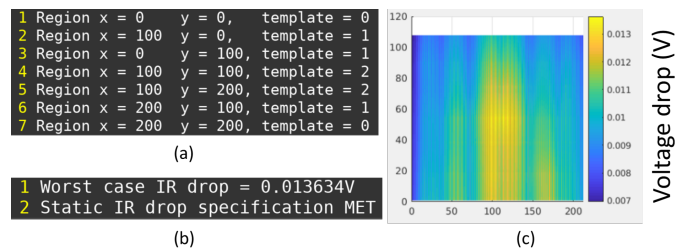

Fig. 3.   (a) Template map, (b) IR drop report, and (c) IR drop distribution.

Fig. 3(a) shows a sample of the template map, generated by OpeNPDN, which depicts the template id, template name and the location of the region on the chip to which it must be applied. Fig. 3(b) shows a sample IR drop distribution, generated by OpeNPDN, using the CNN-synthesized PDN and the current distribution.

III. FUTURE DIRECTIONS

In its current state, OpeNPDN is an ML-based framework which synthesizes an optimized, correct-by-construction, static IR-drop-safe PDN at the placement stage of the design flow. Near-term efforts are focused on developing ML-based methodologies for dynamic power integrity problems. We aim to eventually integrate OpeNPDN within an end to-end hardware design process as part of the OpenROAD project [3]. While the tool is found to provide convincing results in its current state, continuous effort is being focused on addressing certain limitations, involving calibration of the analyzer with commercial tools and SRAM awareness.

IV. LICENSE

The repository [1] is under a permissive BSD-3 Clause license.

REFERENCES

[1] OpeNPDN, https://github.com/The-OpenROAD-Project/OpeNPDN
[2] V. A. Chhabria, *et al.*, "Template-based PDN Synthesis in Floorplan and Placement Using Classifier and CNN Techniques", in *Proc. ASPDAC* (2020), to appear.
[3] The OpenROAD Project, https://theopenroadproject.org
[4] LEF/DEF reference 5.8, http://www.si2.org/openeda.si2.org/projects/lefdefnew
[5] OpenSTA, https://github.com/The-OpenROAD-Project/