Table of Contents

CircuitNet

CircuitNet: An Open-Source Dataset for Machine Learning Applications in Electronic Design Automation (EDA)

CircuitNet is an open-source dataset dedicated to machine learning (ML) applications in electronic design automation (EDA). We have collected more than 10K samples from versatile runs of commercial design tools based on open-source RISC-V designs with various features for multiple ML for EDA applications.

This documentation is organized as followed:

- Introduction: introduction and quick start.
- Feature Description: name conventions, calculation method, characteristics and visualization.

This project is under active development. We are expanding the dataset to include diverse and large-scale designs for versatile ML applications in EDA. If you have any feedback or questions, please feel free to contact us.

Citation

Paper Link

```
@article{chai2022circuitnet,
  title = {CircuitNet: An Open-Source Dataset for Machine Learning Applications in Electronic Design Automation (EDA)},
  author = {Chai, Zhuomin and Zhao, Yuxiang and Lin, Yibo and Liu, Wei and Wang, Runsheng and Huang, Ru},
  journal={arXiv preprint arXiv:2208.01040},
  year = {2022}
}
```

Intro

CircuitNet

CircuitNet is an open-source dataset dedicated to machine learning (ML) applications in electronic design automation (EDA). We have collected more than 10K samples from versatile runs of commercial design tools based on open-source RISC-V designs with various features for multiple ML for EDA applications. The features are saved seperately as below:

```
.
├── routability_features
| ├── cell_density
| └── congestion
| | ├── congestion_early_global_routing
| | | ├── overflow_based
| | | | ├── congestion_eGR_horizontal_overflow
| | | | └── congestion_eGR_vertical_overflow
| | | └── utilization_based
| | |   ├── congestion_eGR_horizontal_util
| | |   └── congestion_eGR_vertical_util
| | └── congestion_global_routing
| |   ├── overflow_based
| |   | ├── congestion_GR_horizontal_overflow
| |   | └── congestion_GR_vertical_overflow
| |   └── utilization_based
| |     ├── congestion_GR_horizontal_util
| |     └── congestion_GR_vertical_util
| ├── DRC
| | ├── DRC_all
| | └── DRC_seperated
| ├── macro_region
| └── RUDY
|   ├── RUDY
|   ├── RUDY_long
|   ├── RUDY_short
|   ├── RUDY_pin
|   └── RUDY_pin_long
├── IR_drop_features
| ├── power_i
| ├── power_s
| ├── power_sca
| ├── power_all
| ├── power_t
| └── IR_drop
├── graph_features
| ├── instance_placement
| └── netlist
├── doc
| └── user_guide.pdf
└── script
| ├── decompress_routability.py
| ├── decompress_IR_drop.py
  └── generate_training_set.py
```

We separate the features and store them in different directories to enable custom applications. Thus they need to be preprocessed and combined in certain arrangement for training. Our scripts can preprocess and combine different features for training and testing. But we also encourage to implement different preprocessing methods

and use different combinations of features.

Quick Start

(1)Based on your target tasks, download Routability Features(for congestion and DRC) or IR Drop Features(for IR drop).

Google Drive

Baidu Netdisk

Decompress with scripts in the script dir

```
python decompress_routability.py
```

or

```
python decompress_IR_drop.py
```

This may take sometime, please be patient.

(2)Run preprocessing script to generate training set for corresponding tasks. Specify your task with option: congestion/DRC/IR_drop.

```
python generate_training_set.py --task [congestion/DRC/IR_drop] --data_path [path_to_decompressed_dataset] --save_path [path_to_save_output]
```

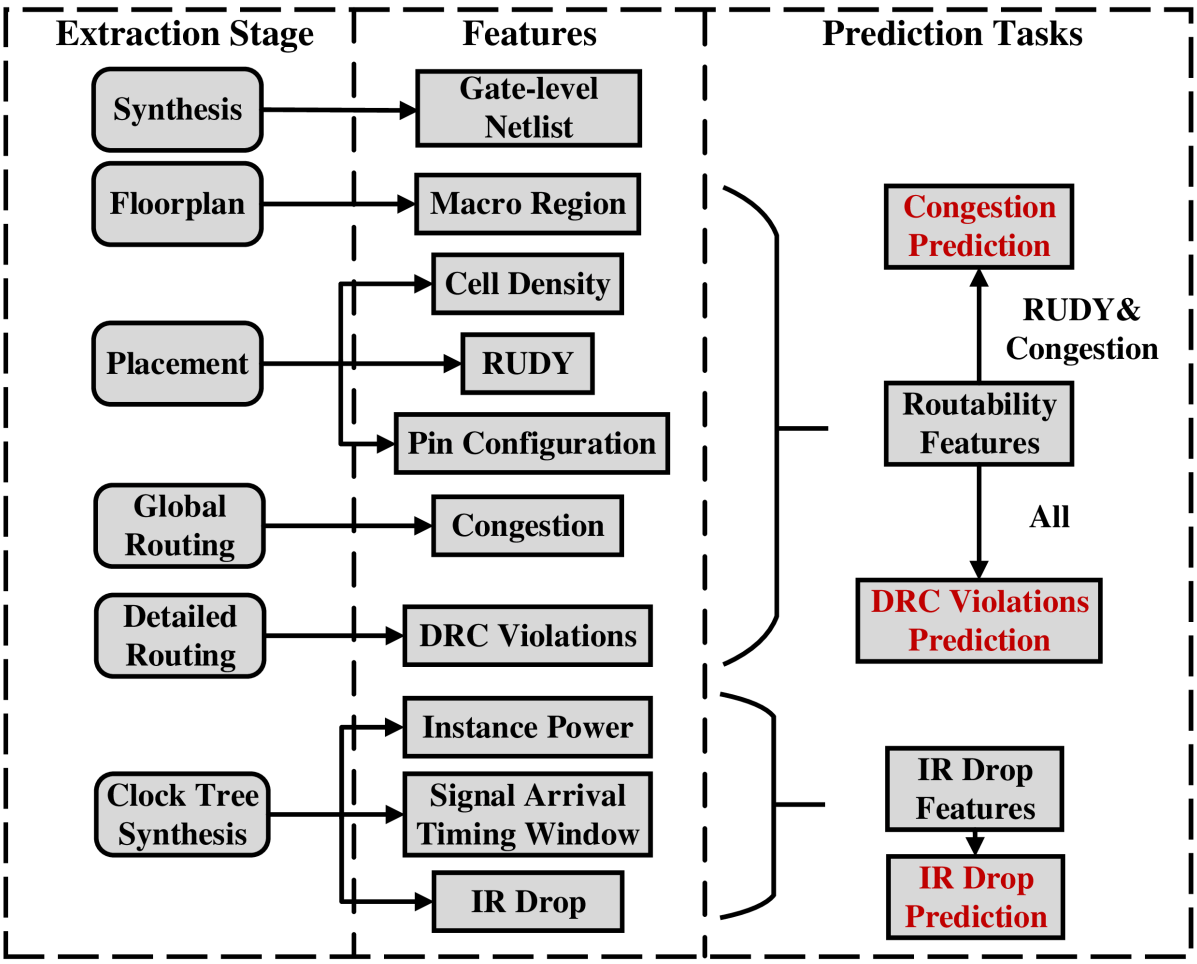
Dataset Overview

The dataset now mainly provide support for three cross-stage prediction tasks in back-end design: congestion prediction, DRC violations prediction and IR drop prediction. The common practice in these tasks is to leverage computer vision methods(e.g. CNN or FCN), thus the main part of CircuitNet is 2D image-like data.

Image-like Feature Maps

The information on layout is converted into image-like feature maps based on tiles of size

1.5μm$\times$1.5μm, and they make up the main part of CircuitNet.



- **Macro Region**:

  the regions covered by macros, used for estimation routing resources available in each tile.

- **Routability Features**:

  (1) Cell density: the cell number counted in each tile.

  (2) RUDY: a routing demand estimation for each net over spatial dimension. It is widely used for its high efficiency and accuracy. A variation named pin RUDY is also included as the pin density estimation.

(3) Pin configuration: a high resolution representation of pin and routing blockage shapes that conveys pin accessibility in routing.

(4) Congestion: the overflow of routing demand in each tile.

(5) DRC violations: the number of DRC violations in each tile.

- **IR Drop Features**:

  (1) Instance power: the instance level internal, switching and leakage power along with the toggles rate from a vectorless power analysis.

  (2) Signal arrival timing window: the possible switching time domain of the instance in a clock period from a static timing analysis for each pin.

  (3) IR drop: the IR drop value on each node from a vectorless power rail analysis.

Supported Prediction Tasks

Congestion Prediction

Predict congestion at post-placement stages.

Input features:

- Macro region
- RUDY
- Pin RUDY

Label:

Congestion

DRC Violations Prediction

Predict DRC violations at post-global-routing stages.

Input features:

- Macro region
- RUDY
- Pin RUDY
- Cell density
- Congestion

Label:

DRC violations

IR Drop Prediction

Predict IR drop at post-CTS stages.

Input features:

Spatial and temporal power maps

Label:

IR drop

Tutorial

Herein, we select several representative methods to give a brief introduction of applying machine learning to VLSI physical design cycle that provides an intuitive awareness of the functionality and practicability of `CircuirNet` to users. Please refer to our github repository for the entire example.

Note that all three selected methods utilize image-like features to train a generative model, such as fully convolutional networks (FCNs) and U-Net, formulating the prediction task into an image-to-image translation task. We did our best to reproduce the experimental environment in the original paper, including model architecture, feature selection and loss. The name of the features are matched with the ones in CircuitNet to avoid confusion.

Congestion Prediction

Congestion is defined as the overflow of routing demand over available routing resource in the routing stage of the back-end design. It is frequently adopted as the metric to evaluate routability, i.e., the prospective quality of routing based on the current design solution. The congestion prediction is necessary to guide the optimization in placement stage and reduce total turn-around time.

The network of `Global Placement with Deep Learning-Enabled Explicit Routability Optimization` [1] uses an FCN based encoder-decoder architecture to translate the image-like features into a congestion map. The architecture is shown in Fig 1.

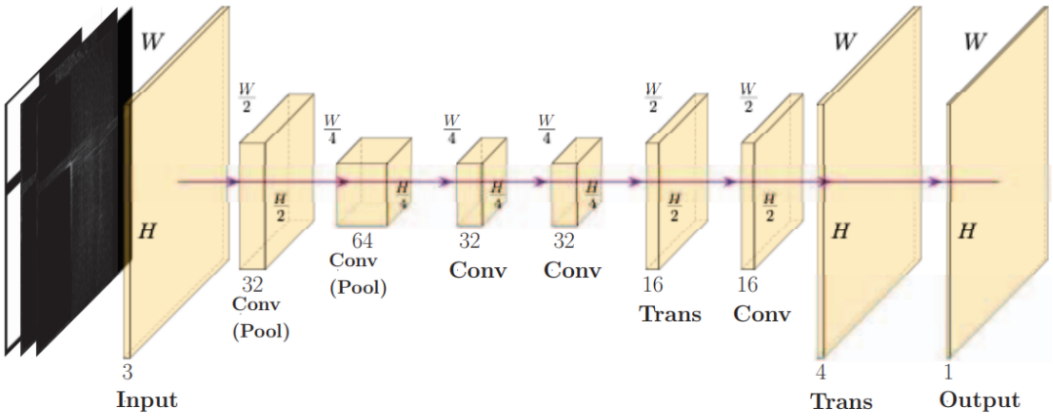


Fig 1 Model architecture.

The generation network consists of two fundamental modules, encoder and decoder, which are designed according to the architecture illustrated in Fig 1.

```
class conv(nn.Module):
    def __init__(self, dim_in, dim_out, kernel_size=3, stride=1, padding=1, bias=True):
        super(conv, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(dim_in, dim_out, kernel_size=kernel_size, stride=stride, padding=padding, bias=bias),
            nn.InstanceNorm2d(dim_out, affine=True),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(dim_out, dim_out, kernel_size=kernel_size, stride=stride, padding=padding, bias=bias),
            nn.InstanceNorm2d(dim_out, affine=True),
            nn.LeakyReLU(0.2, inplace=True),
        )

    def forward(self, input):
        return self.main(input)

class upconv(nn.Module):
    def __init__(self, dim_in, dim_out):
        super(upconv, self).__init__()
        self.main = nn.Sequential(
                nn.ConvTranspose2d(dim_in, dim_out, 4, 2, 1),
                nn.InstanceNorm2d(dim_out, affine=True),
                nn.LeakyReLU(0.2, inplace=True),
                )

    def forward(self, input):
        return self.main(input)

class Encoder(nn.Module):
    def __init__(self, in_dim=3, out_dim=32):
        super(Encoder, self).__init__()
        self.in_dim = in_dim
        self.c1 = conv(in_dim, 32)
        self.pool1 = nn.MaxPool2d(kernel_size=2,stride=2)
        self.c2 = conv(32, 64)
        self.pool2 = nn.MaxPool2d(kernel_size=2,stride=2)
        self.c3 = nn.Sequential(
                nn.Conv2d(64, out_dim, 3, 1, 1),
                nn.BatchNorm2d(out_dim),
                nn.Tanh()
                )

    def init_weights(self):
        generation_init_weights(self)


    def forward(self, input):
        h1 = self.c1(input)
        h2 = self.pool1(h1)
        h3 = self.c2(h2)
        h4 = self.pool2(h3)
        h5 = self.c3(h4)
        return h5, h2  # shortpath from 2->7


class Decoder(nn.Module):
    def __init__(self, out_dim=2, in_dim=32):
        super(Decoder, self).__init__()
        self.conv1 = conv(in_dim, 32)
        self.upc1 = upconv(32, 16)
        self.conv2 = conv(16, 16)
        self.upc2 = upconv(32+16, 4)
        self.conv3 =  nn.Sequential(
                nn.Conv2d(4, out_dim, 3, 1, 1),
                nn.Sigmoid()
                )
```

```
    def init_weights(self):
        generation_init_weights(self)

    def forward(self, input):
        feature, skip = input
        d1 = self.conv1(feature)
        d2 = self.upc1(d1)
        d3 = self.conv2(d2)
        d4 = self.upc2(torch.cat([d3, skip], dim=1))
        output = self.conv3(d4)  # shortpath from 2->7
        return output  return self.main(input)
```

In this work, three features are selected as input features to feed into the model. The included features are (1)macro_region, (2)RUDY, (3)RUDY_pin, and they are preprocessed and combined together as one numpy array by the provided script `generate_training_set.py` (check the quick start page for usage of the script). The visualization of the array is shown in Fig 2.

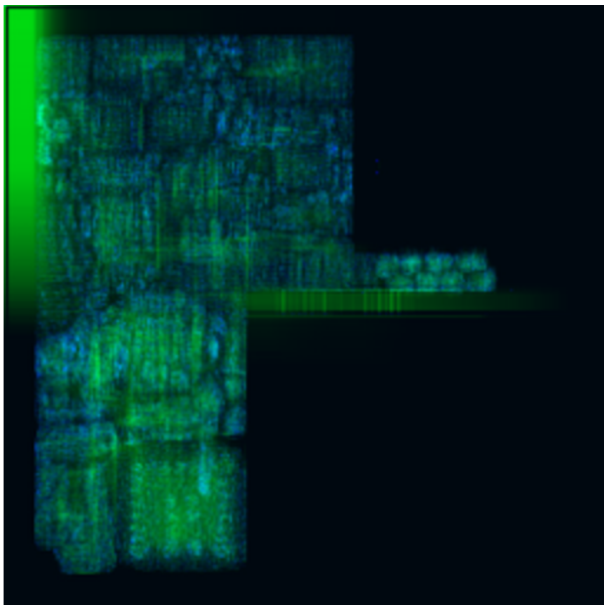


Fig 2 Visualization of the input numpy array.

We create a class called `CongestionDataset` to intake the numpy array of congestion feature and label, while reading and processing them through pytorch `DataLoader`.

```
class CongestionDataset(object):
    def __init__(self, ann_file, dataroot, pipeline=None, test_mode=False, **kwargs):
        super().__init__()
        self.ann_file = ann_file
        self.dataroot = dataroot
        self.test_mode = test_mode
        if pipeline:
            self.pipeline = Compose(pipeline)
        else:
            self.pipeline = None

        self.data_infos = self.load_annotations()

    def load_annotations(self):
        data_infos = []
        with open(self.ann_file, 'r') as fin:
            for line in fin:
                feature, label = line.strip().split(',')
                if self.dataroot is not None:
                    feature_path = osp.join(self.dataroot, feature)
                    label_path = osp.join(self.dataroot, label)
                data_infos.append(dict(feature_path=feature_path, label_path=label_path))
        return data_infos

    def prepare_data(self, idx):
        results = copy.deepcopy(self.data_infos[idx])
        results['feature'] = np.load(results['feature_path'])
        results['label'] = np.load(results['label_path'])

        results = self.pipeline(results) if self.pipeline else results

        feature =  results['feature'].transpose(2, 0, 1).astype(np.float32)
        label = np.expand_dims(results['label'], axis=0).astype(np.float32)

        return feature, label, results['label']

    def __len__(self):
        return len(self.data_infos)

    def __getitem__(self, idx):
        return self.prepare_data(idx)
```

We train this network in an end-to-end manner and compute the loss between the output and the golden congestion map, which are the features named congestion_GR_horizontal_overflow and congestion_GR_vertical_overflow from CircuitNet.

```
class GPDL(nn.Module):
    def __init__(self,
                 in_channels=3,
                 out_channels=2,
                 **kwargs):
        super().__init__()

        self.encoder = Encoder(in_dim=in_channels)
        self.decoder = Decoder(out_dim=out_channels)

    def forward(self, x):
        x = self.encoder(x)
        return self.decoder(x)

    def init_weights(self, pretrained=None, pretrained_transfer=None, strict=False, **kwargs):
        if isinstance(pretrained, str):
            new_dict = OrderedDict()
            weight = torch.load(pretrained, map_location='cpu')['state_dict']
            for k in weight.keys():
                new_dict[k] = weight[k]
            load_state_dict(self, new_dict, strict=strict, logger=None)
        elif pretrained is None:
            generation_init_weights(
                self, init_type=self.init_type, init_gain=self.init_gain)
        else:
            raise TypeError("'pretrained' must be a str or None. "
                            f'But received {type(pretrained)}.')
```

The model is trained for 200k iterations. The curve of training loss and evaluation metrics in training are presented in Fig 3 and Fig 4. Peak signal-to-noise ratio (PSNR) and structure similarity index measure (SSIM) are used to evaluate pixel level accuracy, and the final result of these metrics are 31.0 and 0.873 respectively.

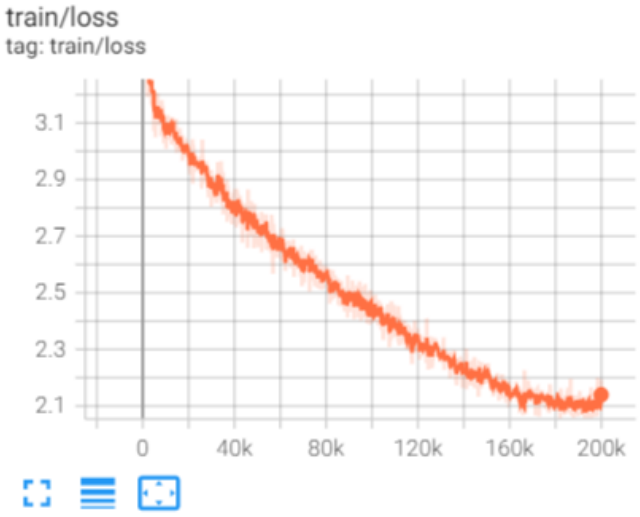


Fig 3 Training loss at different training iterations.

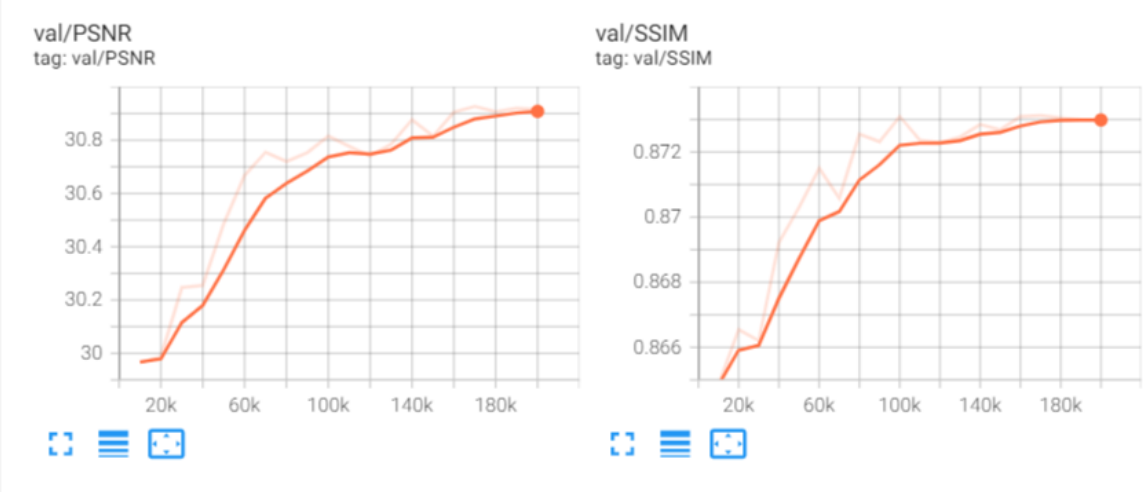


Fig 4 Evaluation metrics(PSNR, SSIM) at different training iterations.

After finishing the training procedure, we dump the visualization of the predicted congestion map, which is shown in Fig 5. The parts with high-contrast indicate the congestion hotspot.

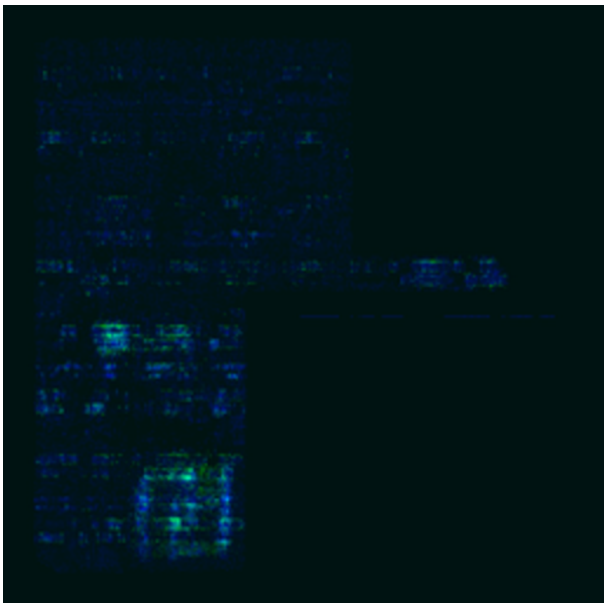


Fig 5 Visualization of the predicted congestion map.

DRC Violation Prediction

Design rule check (DRC) violation is another estimation for routability. The congestion is available after global routing, while DRC violation is reported after detailed routing. And there is a deviation between them at advanced tech node, such as 7 nm. Thus it is also necessary to predict DRC violations directly. `RouteNet: Routability Prediction for Mixed-Size Designs Using Convolutional Neural Network` [2] is a typical method for accurately predicting violation hotspots. The architecture is shown in Fig 6.

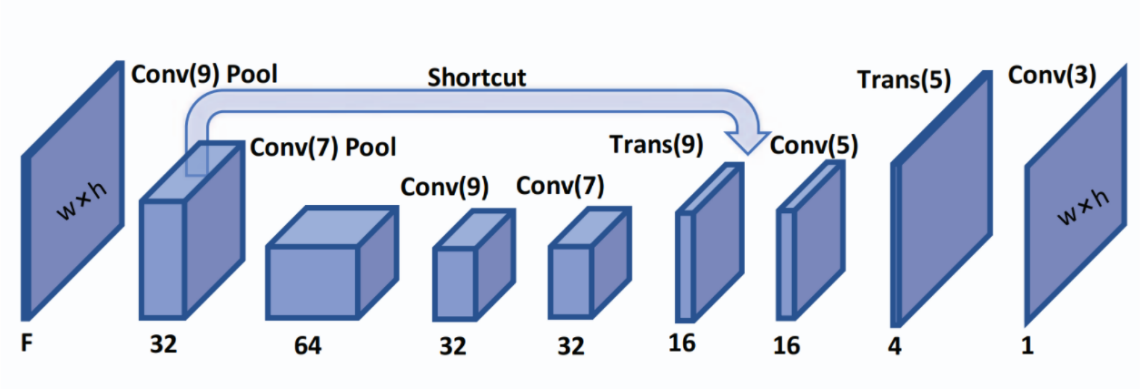


Fig 6 Model architecture.

The network is the same as the one in congestion prediction .

```
class conv(nn.Module):
    def __init__(self, dim_in, dim_out, kernel_size=3, stride=1, padding=1, bias=True):
        super(conv, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(dim_in, dim_out, kernel_size=kernel_size, stride=stride, padding=padding, bias=bias),
            nn.InstanceNorm2d(dim_out, affine=True),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(dim_out, dim_out, kernel_size=kernel_size, stride=stride, padding=padding, bias=bias),
            nn.InstanceNorm2d(dim_out, affine=True),
            nn.LeakyReLU(0.2, inplace=True),
        )

    def forward(self, input):
        return self.main(input)

class upconv(nn.Module):
    def __init__(self, dim_in, dim_out):
        super(upconv, self).__init__()
        self.main = nn.Sequential(
                nn.ConvTranspose2d(dim_in, dim_out, 4, 2, 1),
                nn.InstanceNorm2d(dim_out, affine=True),
                nn.LeakyReLU(0.2, inplace=True),
                )

    def forward(self, input):
        return self.main(input)

class Encoder(nn.Module):
    def __init__(self, in_dim=3, out_dim=32):
        super(Encoder, self).__init__()
        self.in_dim = in_dim
        self.c1 = conv(in_dim, 32)
        self.pool1 = nn.MaxPool2d(kernel_size=2,stride=2)
        self.c2 = conv(32, 64)
        self.pool2 = nn.MaxPool2d(kernel_size=2,stride=2)
        self.c3 = nn.Sequential(
                nn.Conv2d(64, out_dim, 3, 1, 1),
                nn.BatchNorm2d(out_dim),
                nn.Tanh()
                )

    def init_weights(self):
        generation_init_weights(self)


    def forward(self, input):
        h1 = self.c1(input)
        h2 = self.pool1(h1)
        h3 = self.c2(h2)
        h4 = self.pool2(h3)
        h5 = self.c3(h4)
        return h5, h2  # shortpath from 2->7


class Decoder(nn.Module):
    def __init__(self, out_dim=2, in_dim=32):
        super(Decoder, self).__init__()
        self.conv1 = conv(in_dim, 32)
        self.upc1 = upconv(32, 16)
        self.conv2 = conv(16, 16)
        self.upc2 = upconv(32+16, 4)
        self.conv3 =  nn.Sequential(
                nn.Conv2d(4, out_dim, 3, 1, 1),
                nn.Sigmoid()
                )
```

```
    def init_weights(self):
        generation_init_weights(self)

    def forward(self, input):
        feature, skip = input
        d1 = self.conv1(feature)
        d2 = self.upc1(d1)
        d3 = self.conv2(d2)
        d4 = self.upc2(torch.cat([d3, skip], dim=1))
        output = self.conv3(d4)  # shortpath from 2->7
        return output  return self.main(input)
```

In this work, nine features are selected as input features to feed into the model. The included features are (1)macro_region, (2)cell_density, (3)RUDY_long, (4)RUDY_short, (5)RUDY_pin_long, (6)congestion_eGR_horizontal_overflow, (7)congestion_eGR_vertical_overflow, (8)congestion_GR_horizontal_overflow, (9)congestion_GR_vertical_overflow. Again, these features are preprocessed and combined together as one numpy array. The visualization of the array is shown in Fig 7.

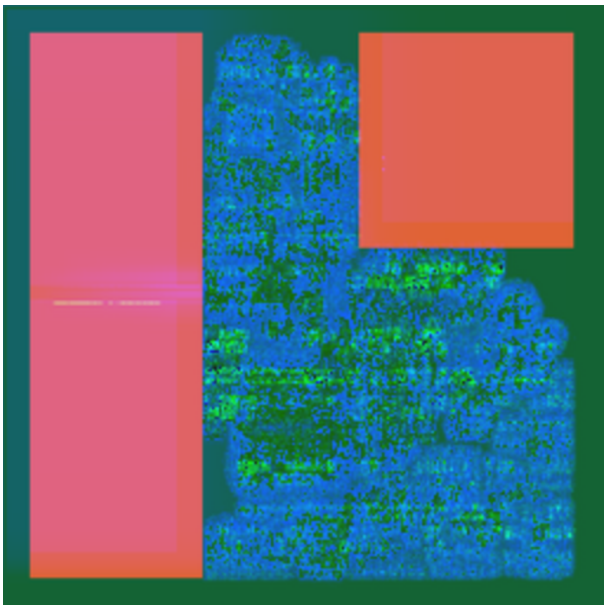


Fig 7 Visualization of the input numpy array.

We create a class called `DRCDataset` to intake the numpy array of congestion feature and label, while reading and processing them through pytorch `DataLoader`.

```
class DRCDataset(object):
    def __init__(self, ann_file, dataroot, test_mode=None, **kwargs):
        super().__init__()
        self.ann_file = ann_file
        self.dataroot = dataroot
        self.test_mode = test_mode
        self.data_infos = self.load_annotations()

    def load_annotations(self):
        data_infos = []
        with open(self.ann_file, 'r') as fin:
            for line in fin:
                feature, label = line.strip().split(',')
                if self.dataroot is not None:
                    feature_path = osp.join(self.dataroot, feature)
                    label_path = osp.join(self.dataroot, label)
                data_infos.append(dict(feature_path=feature_path, label_path=label_path))
        return data_infos

    def prepare_data(self, idx):
        results = copy.deepcopy(self.data_infos[idx])

        feature = np.load(results['feature_path']).transpose(2, 0, 1).astype(np.float32)
        label = np.load(results['label_path']).transpose(2, 0, 1).astype(np.float32)

        return feature, label, results['label_path']

    def __len__(self):
        return len(self.data_infos)

    def __getitem__(self, idx):
        return self.prepare_data(idx)
```

We train this network in an end-to-end manner and compute the loss between the output and the golden DRC violations map, which is the feature named DRC_all from CircuitNet.

```
class RouteNet(nn.Module):
    def __init__(self,
                 in_channels=9,
                 out_channels=2,
                 **kwargs):
        super().__init__()

        self.encoder = Encoder(in_dim=in_channels)
        self.decoder = Decoder(out_dim=out_channels)

    def forward(self, x):
        x = self.encoder(x)
        return self.decoder(x)

    def init_weights(self, pretrained=None, pretrained_transfer=None, strict=False, **kwargs):
        if isinstance(pretrained, str):
            new_dict = OrderedDict()
            weight = torch.load(pretrained, map_location='cpu')['state_dict']
            for k in weight.keys():
                new_dict[k] = weight[k]

            new_dict_clone = new_dict.copy()
            for key, value in new_dict_clone.items():
                if key.endswith(('running_mean', 'running_var')):
                    del new_dict[key]

            load_state_dict(self, new_dict, strict=strict, logger=None)
        elif pretrained is None:
            generation_init_weights(
                self, init_type=self.init_type, init_gain=self.init_gain)
        else:
            raise TypeError("'pretrained' must be a str or None. "
                            f'But received {type(pretrained)}.')
```

The model is trained for 200k iterations. The curve of training loss is presented in Fig 8.

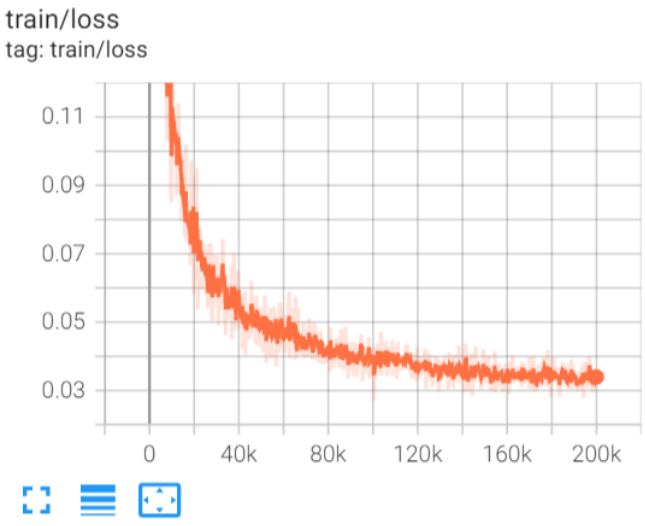


Fig 8 Training loss at different training iterations.

The DRC violations map provides the number of DRC violations in each tile, i.e., in each Gcell in the layout. The visualization of the DRC violations map is shown in Fig 9.

Fig 9 Visualization of the DRC violations map.

In this work, the tiles have number of violations exceeding the threshold are regarded as hotspots. The hotspot is much less than non-hotspot, which is imbalanced, thus the evaluation metrics receiver operating characteristic (ROC) curve and precision-recall(PR) curve, are adopted to evaluate the performance of the method. The result is shown in Fig 10.

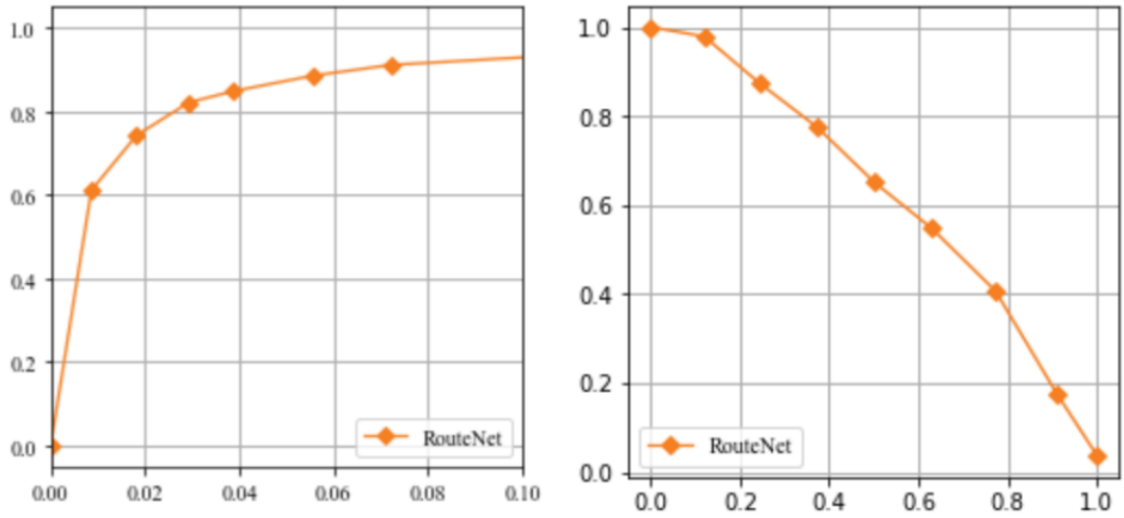


Fig 10 ROC curve and PR curve.

IR Drop Prediction

IR drop is defined as deviation of voltage from reference (VDD, VSS) and it has to be restricted to avoid degradation in timing and functionality. `MAVIREC: ML-Aided Vectored IR-Drop Estimation and Classification` [3] utilizes a U-Net based network to predict IR drop. Due to the demand for joint perception along the temporal and spatial axis, MAVIREC introduces a 3D encoder to aggregate the spatio-temporal features and output the prediction result as a 2D IR drop map.

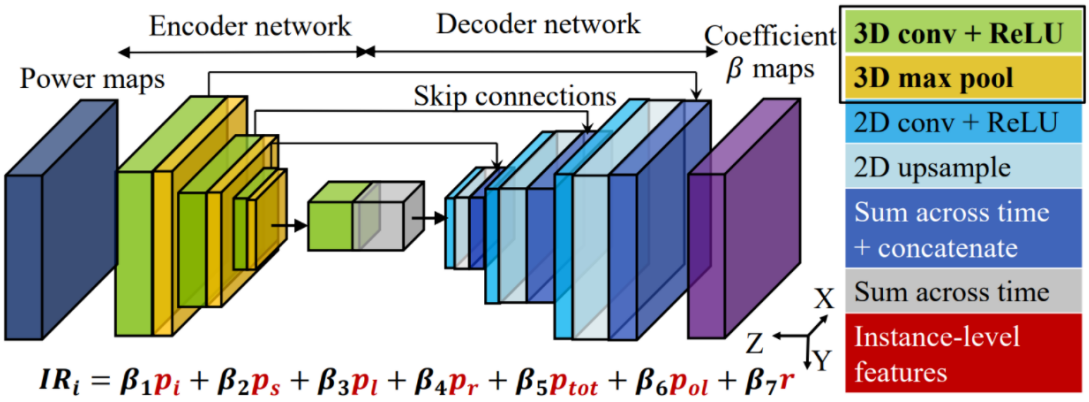


Fig 11 Model architecture.

The generation network consists of two fundamental modules, encoder and decoder, which are designed according to the architecture illustrated in Fig 11.

```python
class DoubleConv3d(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv3d(in_channels, mid_channels, kernel_size=3, padding=(0, 1, 1), bias=False),
            nn.BatchNorm3d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv3d(mid_channels, out_channels, kernel_size=3, padding=(0, 1, 1), bias=False),
            nn.BatchNorm3d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)


class DoubleConv2d(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)



class Down(nn.Module):
    """Downscaling with maxpool then double conv"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool3d((1, 2, 2), stride=(1, 2, 2)),
            DoubleConv3d(in_channels, out_channels)
        )

    def forward(self, x):
        return self.maxpool_conv(x)


class Up(nn.Module):
    """Upscaling then double conv"""

    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        # if bilinear, use the normal convolutions to reduce the number of channels
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
            self.conv = DoubleConv2d(in_channels, out_channels, in_channels // 2)
```

```
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2, kernel_size=2, stride=2)
            self.conv = DoubleConv2d(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)


class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)

    def forward(self, x):
        return self.conv(x)
```

In this work, five features are selected as input features to feed into the model. The included features are (1)power_i, (2)power_s, (3)power_sca, (4)power_all, (5)power_t. Again, these features are preprocessed and combined together as one numpy array. The visualization of the array is shown in Fig 12.

Fig 12 Visualization of input numpy array.

We create a class called `IRDropDataset` to intake the numpy array of congestion feature and label, while reading and processing them through pytorch `DataLoader` .

```
class IRDropDataset(object):
    def __init__(self, ann_file, dataroot, test_mode=False, **kwargs):
        super().__init__()
        self.ann_file = ann_file
        self.dataroot = dataroot
        self.test_mode = test_mode
        self.data_infos = self.load_annotations()

        self.temporal_key = 'Power_t'

    def load_annotations(self):
        data_infos = []
        with open(self.ann_file, 'r') as fin:
            for line in fin:
                infos = line.strip().split(',')
                label = infos[-1]
                features = infos[:-1]
                info_dict = dict()
                if self.dataroot is not None:
                    for feature in features:
                        info_dict[feature.split('/')[0]] = osp.join(self.dataroot, feature)
                    feature_path = info_dict
                    label_path = osp.join(self.dataroot, label)
                data_infos.append(dict(feature_path=feature_path, label_path=label_path))
        return data_infos

    def prepare_data(self, idx):
        results = copy.deepcopy(self.data_infos[idx])

        spatial_feature = []
        for k, v in results['feature_path'].items():
            if k != self.temporal_key:
                spatial_feature.append(np.load(v))
            else:
                temporal_feature = np.load(v)
        feature = [np.array(spatial_feature), temporal_feature]
        feature = np.ascontiguousarray(np.concatenate(feature, axis=0).astype(np.float32))
        feature = np.expand_dims(feature, axis=0)
        label = np.load(results['label_path']).astype(np.float32)
        return feature, label, results['label_path']


    def __len__(self):
        return len(self.data_infos)


    def __getitem__(self, idx):
        return self.prepare_data(idx)
```

We train this network in an end-to-end manner and compute the loss between the output and the golden IR drop map, which is the feature named ir_drop from CircuitNet.

```
class MAVI(nn.Module):
    def __init__(self,
                 in_channels,
                 out_channels,
                 bilinear=False,
                 init_cfg=dict(type='normal', gain=0.02),
                 **kwargs):
        super(MAVI, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.bilinear = bilinear

        self.inc = DoubleConv3d(in_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1

        self.up1 = Up(512, 256 // factor, bilinear)
        self.up2 = Up(256, 128 // factor, bilinear)
        self.up3 = Up(128, 64, bilinear)
        self.outc = OutConv(64, out_channels)

        self.init_type = 'normal' if init_cfg is None else init_cfg.get(
            'type', 'normal')
        self.init_gain = 0.02 if init_cfg is None else init_cfg.get(
            'gain', 0.02)

    def forward(self, x):
        x_in = x[:, :, :self.out_channels, :, :] # [b c 4 h w]
        x1 = self.inc(x)
        x2 = self.down1(x1)  # [1, 64, 20, 256, 256]
        x3 = self.down2(x2)  # [1, 128, 16, 128, 128]
        x4 = self.down3(x3)  # [1, 512, 12, 64, 64]

        x = self.up1(x4.mean(dim=2), x3.mean(dim=2))
        x = self.up2(x, x2.mean(dim=2))
        x = self.up3(x, x1.mean(dim=2))
        logits = self.outc(x)

        logits = x_in.squeeze(1)*logits
        return torch.sum(logits, dim=1)

    def init_weights(self, pretrained=None):
        if isinstance(pretrained, str):
            load_checkpoint(self, pretrained, strict=False, logger=None)
        elif pretrained is None:
            for m in self.modules():
                if isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
                    constant_init(m.weight, 1)
                    constant_init(m.bias, 0)

                if isinstance(m, nn.Conv3d):
                    kaiming_init(m)
                elif isinstance(m, _BatchNorm):
                    constant_init(m, 1)
        else:
            raise TypeError(f'"pretrained" must be a str or None. '
                            f'But received {type(pretrained)}.')
```

The IR drop map provides the maximum IR drop value in each tile, i.e., in each Gcell in the layout. The visualization of the IR drop map is shown in Fig 13.

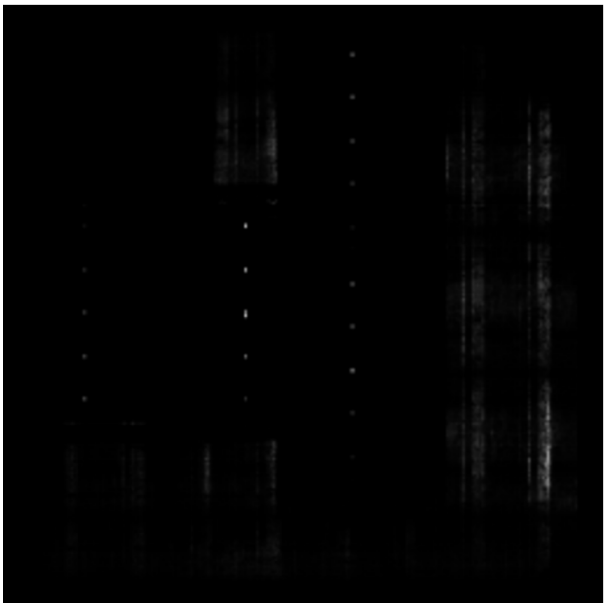


Fig 13 Visualization of the IR drop map.

The model is trained for 200k iterations. The curve of training loss is presented in Fig 14.

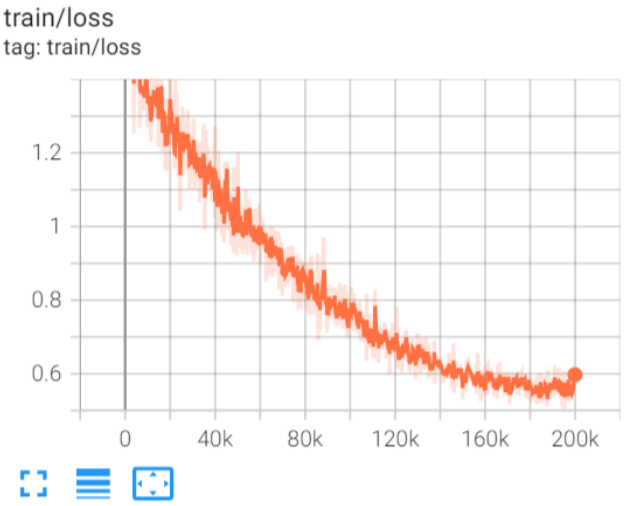


Fig 14 Training loss at different training iterations.

In this work, the tiles have IR drop value exceeding the threshold are regarded as hotspots. Thus, the same evaluation metrics as the DRC violation prediction task, which are ROC curve and PR curve, are adopted to evaluate the performance of the method. The result is shown in Fig 15.

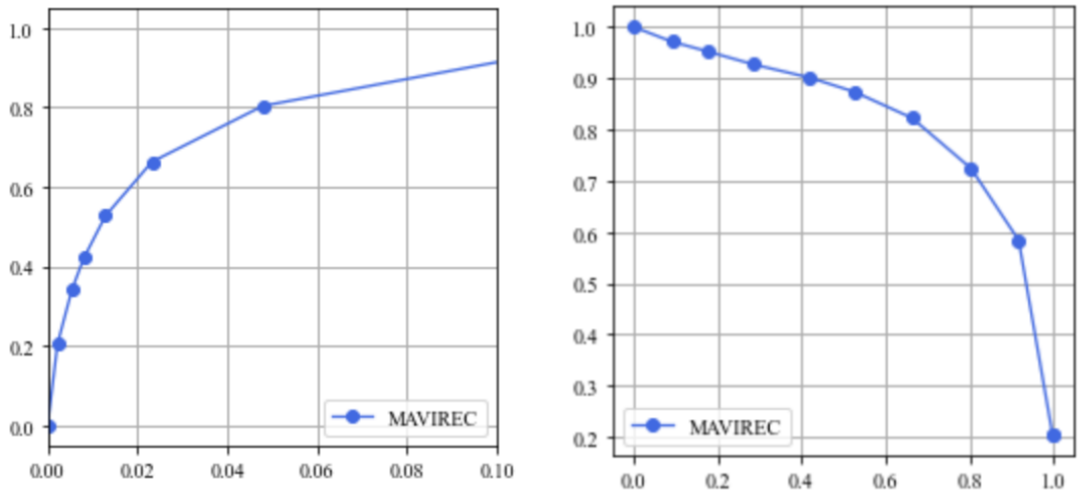


Fig 15 ROC curve and PR curve.

Citation

[1] S. Liu, et al. “Global Placement with Deep Learning- Enabled Explicit Routability Optimization,” in DATE 2021. 1821

[2] Z. Xie, et al. “RouteNet: Routability prediction for mixed-size designs using convolutional neural network,” in ICC

[3] V. A. Chhabria, et al. “MAVIREC: ML-Aided Vectored IR-Drop Estimation and Classification,” in DATE 2021. 1825-1828.

Basic Properties

All features are tile-based. Most information in layout is mapped into tiles with a size of 1.5μm$\times$1.5μm. Moreover, layouts are around 450μm$\times$450μm, resulting in feature maps of around 300$\times$300 tiles. **In summary, most of the feature maps are 2-dimension numpy array [w, h] unless otherwise indicated.** Their detailed calculations are described in the following sections.

Note that the features need to be preprocessed for training, including resizing and normalization. We provide script of our customized preprocessing method used in our experiment, but there is more than one way to complete preprocessing.

Naming Conventions

10242 samples are generated for feature extraction from 6 original RTL designs with variations in synthesis and physical design as shown in table below.

<table>
<tr><th rowspan="2">Design</th><th colspan="2">Synthesis Variations</th><th colspan="4">Physical Design Variations</th></tr>
<tr><th>#Macros</th><th>Frequency (MHz)</th><th>Utilizations (%)</th><th>#Macro Placement</th><th>#Power Mesh Setting</th><th>Filler Insertion</th></tr>
<tr><td>RISCY-a</td><td rowspan="3">3/4/5</td><td rowspan="6">50/200/500</td><td rowspan="6">70/75/80/85/90</td><td rowspan="6">3</td><td rowspan="6">8</td><td rowspan="6">After Placement /After Routing</td></tr>
<tr><td>RISCY-FPU-a</td></tr>
<tr><td>zero-riscy-a</td></tr>
<tr><td>RISCY-b</td><td rowspan="3">13/14/15</td></tr>
<tr><td>RISCY-FPU-b</td></tr>
<tr><td>zero-riscy-b</td></tr>
</table>

The naming convention for extracted feature maps is defined as: {Design name}-{#Macros}-c{Clock}-u{Utilizations}-m{Macro placement}-p{Power mesh setting}-f{filler insertion}

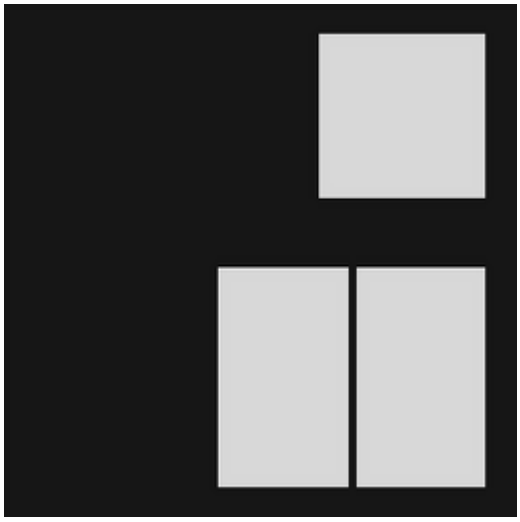
Here is an example: RISCY-a-1-c2-u0.7-m1-p1-f0

Comparison table		
Design name	6 RTL designs	
#Macros	3/4/5 or 13/14/15	1/2/3
Clock	Frequency 500/200/50 MHz	Clock period 2/5/20 ns
Utilizations	70/75/80/85/90%	0.7/0.75/0.8/0.85/0.9
Macro placement	3	1/2/3
Power mesh setting	8	1/2/3/4/5/6/7/8
filler insertion	After placement/After routing	1/0

Routability Features

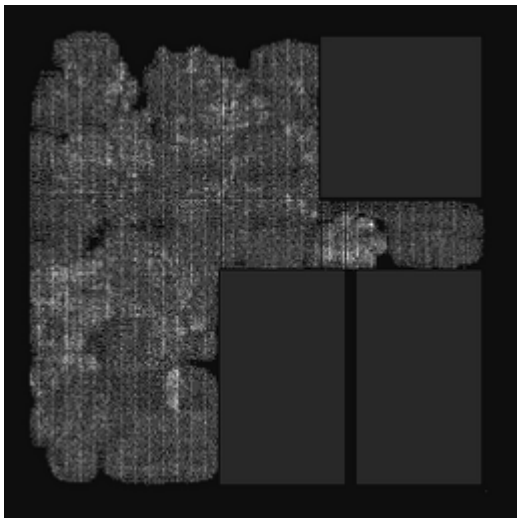
Macro Region ①

The region on the layout covered by macro which shows the relative routing resource distribution. Region covered and uncovered by macro denoted as different grey scale, 1 and 0, respectively.



Cell Density ②

Density distribution of cells, which is equivalent to the cell counts in each tile.



Congestion ③ ~ ⑩

name	computation apporach	stage	direction	used task
congestion_eGR_horizontal_overflow ③	overflow	early global routing	horizontal	Congestion/DRC
congestion_eGR_vertical_overflow ④			vertical	
congestion_GR_horizontal_overflow ⑤		global routing	horizontal	
congestion_GR_vertical_overflow ⑥			vertical	
congestion_eGR_horizontal_util ⑦	utilization	early global routing	horizontal	none
congestion_eGR_vertical_util ⑧			vertical	
congestion_GR_horizontal_util ⑨		global routing	horizontal	
congestion_GR_vertical_util ⑩			vertical	

- Computation method:

  Congestion is computed based on the routing resources reported by Innovus, and there are 2 computation method, overflow based and utilization based. The report basically contains 3 information: total tracks, remain tracks and overflow, based on each GCell, aka tile. Wires have to be routed on tracks, thus tracks are equivalent to routing resources.

  Overflow based congestion is computed as $\frac{overflow}{totaltracks}$. Overflow is the extra demand over total tracks and reflects where congestion occurs.

  Utilization based congestion is computed as $\frac{remaintracks}{totaltracks}$. Utilization reflects the distribution of routing resources.

- Stage: Congestion is reported by Innovus in 2 different stage, eGR and GR. eGR is early global routing, aka trial routing. It is done after placement as a quick and early estimation for congestion. GR is global routing, and the congestion is more accurate than eGR in this stage.

- Direction: The tech lef we use is of type HVH, which meaning that the wires on M1 is horizontal, the ones on M2 is vertical and so on. In this way, the congestion is divided into 2 directions, horizontal and vertical.

RUDY ⑪ ~ ⑮

RUDY refers to Rectangular Uniform wire DensitY which works as a early routing demand estimation after placement. There are several derivatives:

- RUDY ⑪
- RUDY long ⑫
- RUDY short ⑬
- RUDY pin ⑭
- RUDY pin long ⑮

(1) For the kth net with bounding box $(x_{k,min}, x_{k,max}, y_{k,min}, y_{k,max})$, its *RUDY* at tile (i,j) with bounding box $(x_{i,min}, x_{i,max}, y_{j,min}, _{j,max})$ is defined as

$$w_k = x_{k,max} - x_{k,min}$$

$$h_k = y_{k,max} - y_{k,min}$$

$$s_k = (min(x_{k,max}, x_{i,max}) - max(x_{k,min}, x_{i,min})) \times (min(y_{k,max}, y_{j,max}) - max(y_{k,min}, y_{j,min}))$$

$$s_{ij} = (x_{i,max} - x_{i,min}) \times (y_{j,max} - y_{j,min})$$

$$RUDY_k(i,j) = \frac{w_k+h_k}{w_k \times h_k} \frac{s_ij}{s_k}$$

where min()/max() return the smaller/larger value among 2 inputs, s_{ij} is the area of tile (i,j) and s_k denotes the area of tile (i,j) covered by net k.

(2) *RUDY long* and *RUDY short* are the decomposition of *RUDY*, concerning the length of net k. If net k covers more than 1 tile, it contributes to *RUDY long*. Otherwise, net k covers only 1 tile, then it contributes to *RUDY short*.

(3) *RUDY pin* is calculated on the basis of each pin and the net connected the pin, and it is in analog for pin density. For tile (i,j), *RUDY pin* of a pin belonging to net k is calculated as

$$RUDYpin(i,j) = \frac{w_k+h_k}{w_k \times h_k}$$

RUDY pin long is defined in symmetry with *RUDY long* as the decomposition of *RUDY pin* , i.e., if net k covers more than 1 tile, its pins contributes to *RUDY pin long*.

DRC ⑯

Design rule check violations counted in each tile. Different types of DRC are both saved together in one map and seperately saved.

IR Drop Features

Power Maps

Including 5 component：1. internal power: $power_i$, 2. switching power: $power_s$, 3. toggle rate scaled power: $power_{sca}$, 4. all: $power_{all}$, 5. time-decomposed power: $power_t$. They are generated with power report and timing window report from Innovus.

(1) Power report contains instance level power and toggles rate from a vectorless power analysis.

- Internal power (p_i)
- Switching power (p_s)
- Leakage power (p_l)
- Toggles rate (r_{tog})

Then these instance level power is merged into corresponding tile to form power maps.

$$power_i \propto p_i$$

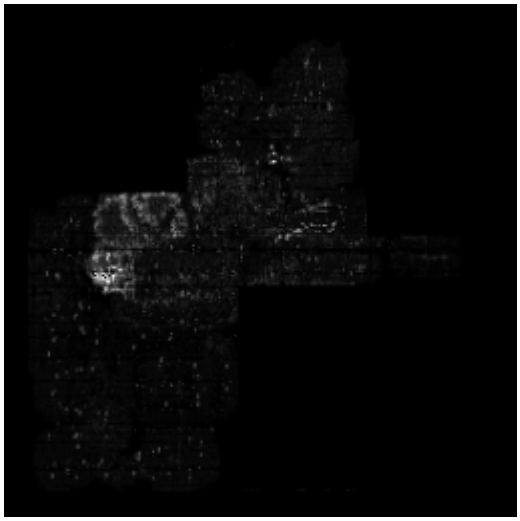
$$power_s \propto p_s$$

$$power_{sca} \propto (p_i + p_s) \times r_{tog} + p_l$$

$$power_{all} \propto p_i + p_s + p_l$$

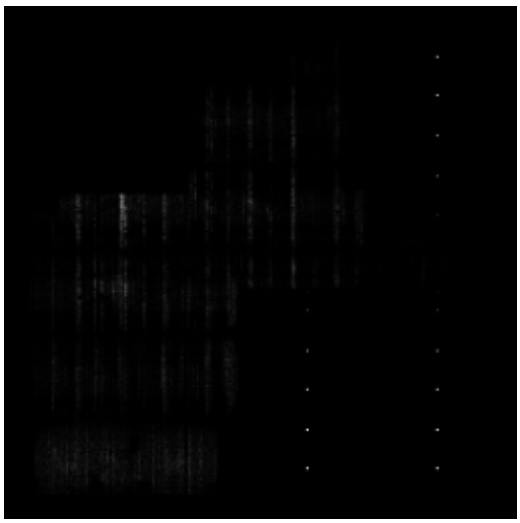
(2) Timing window report contains possible switching time domain of the instance in a clock period from a static timing analysis for each pin. The clock period is decomposed evenly into 20 parts, and the cell contributes to power map $power_t$ only in the parts that it is switching.

$$power_t[0, 19] \propto p_{sca}$$



IR Drop Map

IR drop value on each node from a vectorless power rail analysis is merged into corresponding tile to form IR drop maps.



Graph Features

Gate-level Netlist & Instance Placement

To enable application for graph based methods, we further provide 54 gate-level netlists and 10242 instance placement information.

(1) The gate-level netlists are the ones used in data generation. They are synthesised from 6 RISC-V designs with commercial 28 nm library and Synopsys Design Compiler with multiple variations. (see page Feature for detailed information about variations)

The name of standard cell and IP is encrypted because of copyright issue.

(2) The instance, i.e., standard cell and IP, is placed at certain location on layout after placement stage in back-end design.

The placement information for each layout, i.e., the location of instances, is saved as a dictionary, containing the name of instance (consistent with the ones in netlist) and the coordination for the bounding box of instance on layout.

e.g., InstanceN : [left, bottom, right, top]

The dictionary can be loaded with

```
numpy.load(FILE_NAME, allow_pickle=True)
```

(3) Graph can be obtained with the connectivity information from netlist as edges, and the instance placement information as vertices.

BSD 3-Clause License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.