AiEDA: An Open-Source AI-Aided Design Library for Design-to-Vector

Yihang Qiu, Zengrong Huang, Simin Tao, Hongda Zhang, Weiguo Li, Xinhua Lai, Rui Wang, Weiqiang Wang, Xingquan Li

***Abstract*—Recent research has demonstrated that artificial intelligence (AI) can assist electronic design automation (EDA) in improving both the quality and efficiency of chip design. But current AI for EDA (AI-EDA) infrastructures remain fragmented, lacking comprehensive solutions for the entire data pipeline from design execution to AI integration. Key challenges include fragmented flow engines that generate raw data, heterogeneous file formats for data exchange, non-standardized data extraction methods, and poorly organized data storage. This work introduces a unified open-source library for EDA (AiEDA) that addresses these issues. AiEDA integrates multiple design-to-vector data representation techniques that transform diverse chip design data into universal multi-level vector representations, establishing an AI-aided design (AAD) paradigm optimized for AI-EDA workflows. AiEDA provides complete physical design flows with programmatic data extraction and standardized Python interfaces bridging EDA datasets and AI frameworks. Leveraging the AiEDA library, we generate iDATA, a 600GB dataset of structured data derived from 50 real chip designs (28nm), and validate its effectiveness through five representative AAD tasks spanning prediction, generation, and optimization. The code is publicly available at https://github.com/OSCC-Project/AiEDA, while the full iDATA dataset is being prepared for public release, providing a foundation for future AI-EDA research.**

***Index Terms*—Electronic design automation (EDA), AI-aided design (AAD), AI for EDA library, vectorization dataset.**

I. Introduction

PHYSICAL implementation is an important part of electronic design automation (EDA), transforming gate-level netlists into manufacturable graphic design system (GDS-II) files. Recently, machine learning (ML) techniques have gained significant traction in EDA, demonstrating substantial potential for enhancing both efficiency and quality of physical design. These ML-based methods primarily target prediction [1]–[5], generation [6], [7], and optimization [8]–[10] tasks across various design stages, including floorplanning [10], placement [7], clock tree synthesis [4], and routing [8]. Despite considerable progress in AI-aided design (AAD), the field still lacks specialized research infrastructure [11]. This gap has motivated the development of several AAD infrastructures, each addressing specific aspects of the research ecosystem.

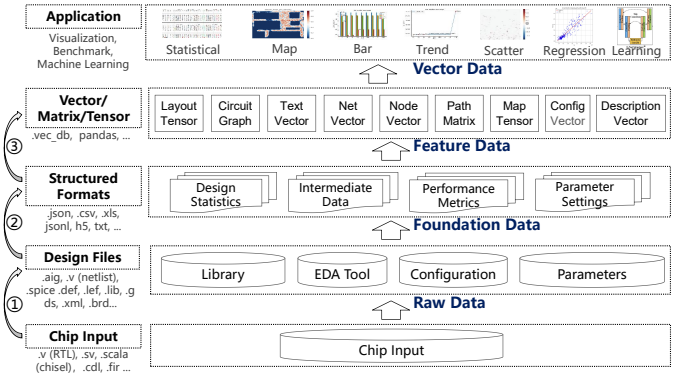


Fig. 1. Data transformation pipeline for AI-aided design.

A. Available AI-EDA Infrastructures

To effectively advance AAD, establishing foundational infrastructures is essential. From a data perspective, the transformation pipeline involves three key stages. First, initial chip design data (`.verilog`) is converted into detailed design process data (`.netlist/.def/.gds`); we define these primary EDA tool outputs as **Raw Data**. Second, this Raw Data is parsed and structured into an AI-friendly, near-lossless representation that we term **Foundation Data** (e.g., organized into `.json/.csv` files). Inspired by the paradigm of Foundation Models in AI, this Foundation Data is engineered to be a clean, comprehensive, and reusable foundation for a diverse array of downstream AI tasks. Third, task-specific features (**Feature Data**) are extracted from the Foundation Data to generate the final vector, matrix, or tensor representations used directly by machine learning models. Collectively, we refer to these three stages as the **design-to-vector** pipeline. Fig. 1 illustrates the complete data transformation pipeline for AAD. Several critical components are essential for this process: complete physical design flows with programmatic data generation capabilities, standardized methodologies for data representation and organization, and unified AI interfaces bridging EDA datasets and ML frameworks. These foundational elements collectively constitute AAD infrastructures, serving as the backbone for systematic and reproducible AI-driven EDA research.

Table I compares existing open-source infrastructures for physical design across three dimensions: toolchain integra-

This article was recommended by Associate Editor Bei Yu. *(Corresponding authors: Weiqiang Wang, Email: wqwang@ucas.ac.cn; Xingquan Li, Email: lixq01@pcl.ac.cn)*

This work was supported in part by the Major Key Project of PCL (No. PCL2025AS04, PCL2025AS05), the NSF of China (No. 62090024), and the NSF of Fujian Province under Grants (No. 2024J09045).

Yihang Qiu, Hongda Zhang, Xinhua Lai, Weiqiang Wang are with the School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, 100049, China.

Zengrong Huang, Simin Tao, Weiguo Li, Xingquan Li are with Pengcheng Laboratory, Shenzhen, 518055, China.

Rui Wang is with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, 518060, China.

Digital Object Identifier xx.xxxx/TCAD.xxxx.xxxxxxx.

TABLE I
COMPARISON OF INFRASTRUCTURES FOR PHYSICAL DESIGN.

Work \ Function	Run Flow (Flow APIs)	Extract Data (Data APIs)	AI Integration
OpenLANE [12]	Multi	×	×
SiliconComliper [13]	Multi	×	×
OpenROAD [14]	Single	✓	×
iEDA [15], [16]	Single	✓	×
METRICS2.1 [17]	Single	Design Process Metrics	×
CircuitOps+ [18], [19]	Single	Graph-based Netlist Features	✓
AiEDA(This work)	Multi	Foundation Data	✓

tion, data extraction support, and AI interface availability. OpenLANE [12] and SiliconCompiler [13] integrate multiple EDA tools to automatically generate layouts from any register transfer level (RTL) design. Single-toolchain infrastructures such as OpenROAD [14] and iEDA [15], [16] provide Python interfaces that encapsulate their underlying C++ APIs, enabling rapid data extraction. METRICS2.1 [17] supports flow execution using OpenROAD and standardizes design process metrics collection in JSON format. CircuitOps [18] introduces an intermediate representation for netlists, constructing labeled property graphs and storing feature data in multiple CSV files to facilitate data preprocessing using Python libraries. Its integration into OpenROAD (CircuitOps+ [19]) enhances ML interaction through Python API feedback loops. Other related efforts structured design data for different goals, such as accelerating EDA tool [20], rather than for AI integration. Table I reveals that while pioneering infrastructures provide critical capabilities, they often target specific aspects. For instance, CircuitOps is expertly optimized for a graph-centric representation of netlists, while iEDA, is a self-contained physical design implementation, offers limited capabilities for AI integration. In contrast, AiEDA (this work) provides comprehensive coverage across all dimensions: support for multiple design flows (e.g., iEDA, OpenROAD, Innovus), enabling Foundation Data extraction, and unified AI interfaces bridging EDA datasets and ML frameworks.

B. Our Motivation and Contribution

The fragmentation of existing AAD infrastructures, highlighted in Table I, underscores the need for a more systematic approach. Our key insight is that overcoming these limitations requires a robust **methodology** and **platform** for transforming design data into AI-ready formats. Building directly upon the design-to-vector paradigm, we developed AiEDA, a comprehensive library designed to serve as this unified foundation. The contributions can be summarized as follows:

- We propose the **design-to-vector paradigm**, a systematic methodology for transforming heterogeneous chip design data into a variety of structured, AI-friendly formats (e.g., graphs, tensors, sequences). Based on this, we define the AAD paradigm, which shifts the focus from traditional process-centric approaches to data-oriented workflows optimized for AI integration.
- We develop **AiEDA**, a comprehensive open-source AAD library featuring complete physical design workflows, programmatic data extraction, structured data management, and unified interfaces for AI applications.
- Leveraging our AiEDA library, we present **iDATA**, a ready-to-use 600GB dataset featuring multi-level structured data derived from 50 real 28nm chip designs.
- We implement five representative tasks to demonstrate the effectiveness of our library and structured dataset, covering prediction, generation, and optimization, each targeting a distinct level of its representation hierarchy.

This paper is structured as follows: Section II details data challenges and examples of the design-to-vector concept. Section III presents the AiEDA library architecture. Section IV describes the iDATA dataset. Section V presents five downstream tasks with experimental results. Section VI draws conclusions.

II. DATA CHALLENGES AND DESIGN-TO-VECTOR

A. Data Challenges for AI-aided Design

To enable AI-aided design (AAD), high-quality training data from real chip designs is crucial. However, current AAD infrastructure remains fragmented, with no complete solution existing for the entire data pipeline. Key challenges include: (1) fragmented flow engines that generate raw data, (2) inconsistent file formats for data exchange, (3) non-standardized data extraction methods, and (4) poorly organized data storage.

1) Challenge 1: Fragmented Flow Engines: AI training data generation from RTL to GDS involves multiple stages and heterogeneous EDA tools, both commercial and open-source [3], [5], [7]. Current tools often have inconsistent objectives and output diverse formats and metrics, leading to fragmented and low-quality data. This fragmentation lowers data quality, limits interoperability, and hinders model deployment in real design flows. A unified workflow platform is needed to standardize configurations, ensure consistency, and enable seamless AI integration for inference and optimization.

2) Challenge 2: Heterogeneous File Formats: Table II summarizes common input formats for physical design tools, illustrating file diversity. Each format possesses distinct syntax, semantics, and structures. These formats are incompatible with direct AI training, requiring extraction and transformation into vectorized representations. Without a unified conversion library, preprocessing becomes a bottleneck, delaying dataset creation and reducing reliability. Developing a standardized file-to-vector tool is essential to preserve design intent and relationships while enabling rapid, consistent data preparation for AI models, supporting scalable and effective automated design workflows.

3) Challenge 3: Non-standardized Data Extraction: AI-EDA research suffers from inconsistent data extraction methods, making results irreproducible. Different studies use diverse datasets, features, and parsing approaches, requiring custom tool interfaces and manual feature engineering. This inconsistency increases engineering effort, reduces feature reliability, and prevents fair benchmarking. The lack of standardization raises entry barriers and introduces variability in metrics. A unified data extraction framework with standardized interfaces and verification mechanisms is essential to convert raw EDA data into consistent, AI-ready formats.

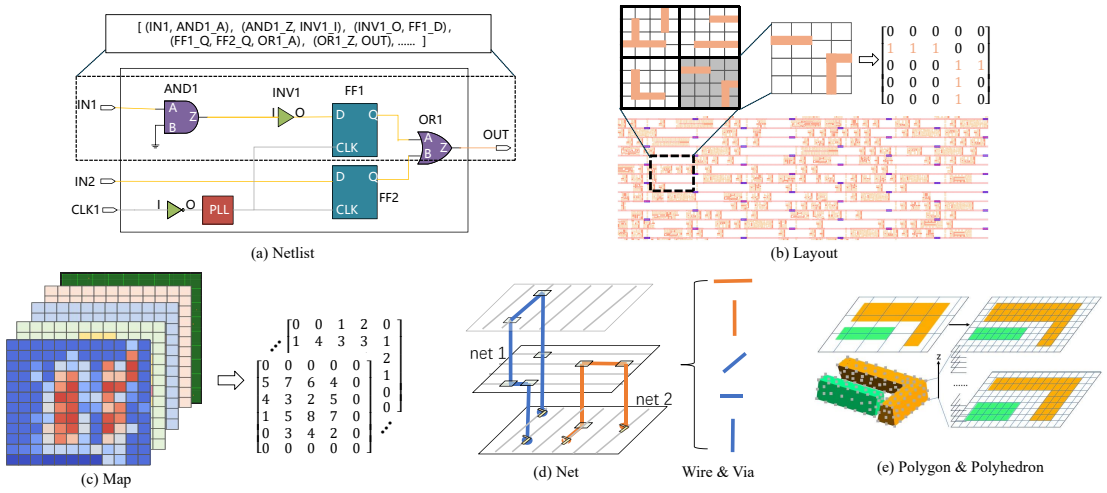


Fig. 2. Examples of design-to-vector conversion for different data types.

TABLE II
FILE FORMATS USED BY PHYSICAL DESIGN TOOLS.

Function	Input
Physical Implementation	`.v` (Verilog), `.lef` (Library Exchange Format), `.def` (Design Exchange Format), `.lib` (timing library), `.sdc` (Synopsys Design Constraints)
Layout Design	`.cdl` (Circuit Description Language), `.spice` (simulation netlist), `.oa` (OpenAccess database), `.gds` (GDSII stream format)
Power Analysis	`.lef`, `.def`, `.lib`, `.upf/.cpf` (power intent), `.vcd` (switching activity), `.saif` (switching activity), `.captable` (parasitic tables)
Thermal Analysis	`.xml` (thermal parameters), `.csv` (thermal boundary conditions), `.flp` (floorplan), `.ptrace` (power trace)
Timing Analysis	`.sdc`, `.lib`, `.spef` (parasitic extraction), `.mmmc` (multi-mode multi-corner)
Design Rule Check (DRC)	`.gds`, `.lef`, `.rule/.drc` (DRC rules), `.layermap` (layer mapping)
Signal Integrity	`.spef`, `.lib`, `.sdc`, `.def`

4) Challenge 4: Chaotic Data Organization: Feature data from EDA tools is often stored in fragmented formats like CSV or NumPy, losing critical information and limiting reuse. Researchers repeatedly rebuild datasets, reducing efficiency and reproducibility. A unified framework is needed to preserve data integrity, support multi-tool interoperability, enable Python integration, and standardize access. This ensures consistent, AI-ready datasets, facilitates reproducible research, and accelerates AI-EDA development, overcoming inefficiencies caused by disorganized data storage and fragmented workflows.

B. Solutions of Design-to-Vector

To address the data challenges, we introduce the **design-to-vector paradigm**. While converting design data for machine learning is not new, our paradigm formalizes and unifies prior task-specific efforts [21] into a general framework. Crucially, "vector" is used broadly to mean any structured, AI-ready data representation, explicitly including graphs for GNNs and images/tensors for CNNs. Our framework standardizes the conversion of raw, heterogeneous EDA data into these diverse, analysis-ready formats. This systematically lowers the engineering barrier for researchers, enabling them to readily generate graphs from netlists, tensors from layouts, and sequences from timing paths. The following examples illustrate how this paradigm is applied to different types of design data.

1) Netlist-to-Vector: A gate-level netlist consists of functional gates (e.g., AND, OR), sequential elements (e.g., flip-flops, latches), and their interconnecting nets. This structure can be abstracted as a hyper-graph $G = (V, E)$, where V represents vertices (gates and primary I/Os) and E denotes hyper-edges (nets). For efficient storage, sparse representations such as the Sparse Adjacency Matrix and Incidence List are employed. Fig. 2(a) illustrates converting netlists (graph or path) to vectors, enabling AI tasks like gate classification, timing/power prediction, and performance optimization.

2) Layout-to-Vector: A physical layout, represented by patterned layers (e.g., metal, via), can be treated as multi-channel image data. Discretization is achieved via Binary Pixelization or Multi-channel Encoding. Fig. 2(b) shows how geometric patterns are converted into vector representations. Layout and patch-level vectors support AI-EDA tasks such as congestion evaluation, DRC violation prediction and hotspot detection.

3) Map-to-Vector: Physical design produces evaluation metrics such as timing, power, DRC, congestion, and IR drop maps, reflecting the spatial layout distribution. These metrics form grid-based structures that can be discretized into matrices. Fig. 2(c) shows this transformation into structured tensors. Map-level discretization enables multi-channel convolution for joint spatial analysis, supporting predictive modeling for hotspot mitigation, design closure acceleration, and Pareto-optimal optimization across electrical, thermal, and mechanical domains.

4) Net-to-Vector: During routing, interconnects from the netlist are instantiated across metal and via layers following design rules. Each net can be decomposed into geometric primitives—wires and inter-layer vias using Steiner or critical points as anchors. Metal wires are represented as vectors $\vec{w} = (x_s, y_s, x_e, y_e, l)$ and inter-layer vias are encoded as $via = (x_c, y_c, l_b, l_t)$. Fig. 2(d) shows nets converted into discrete wires and vias. This representation supports AI tasks such as wirelength estimation, RC prediction, and rule-compliant routing generation.

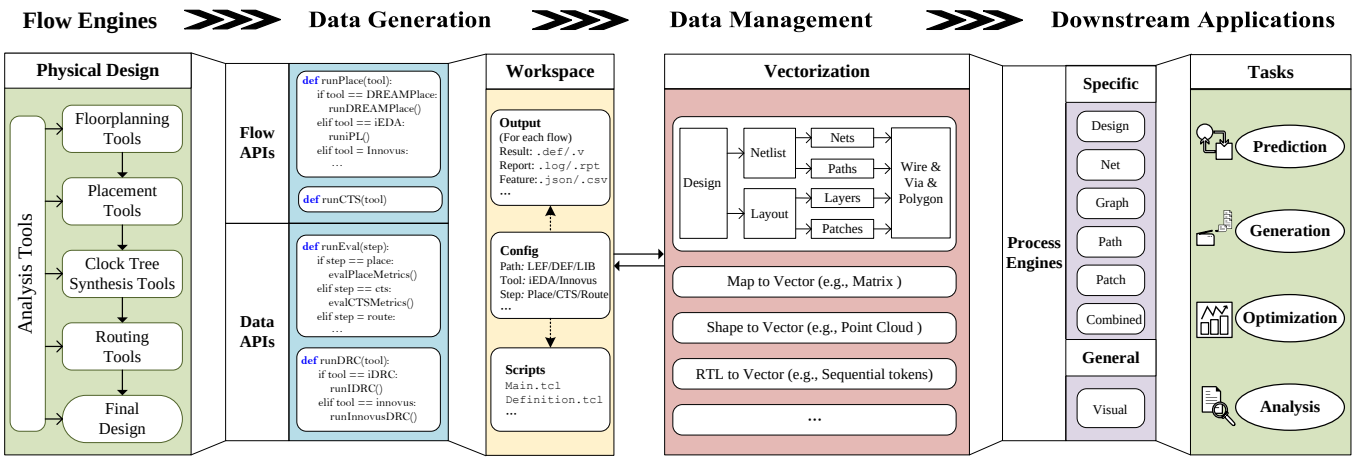


Fig. 3. The AI-Aided Design (AAD) library for design-to-vector.

5) Shape-to-Vector: The final chip implementation consists of 2D and 3D shapes representing device structures. These are discretized into machine-readable forms while preserving topological features. 2D shapes are rasterized into grids encoding material properties, with adjustable resolution balancing accuracy and storage. 3D structures use techniques such as 2.5D layered grids, finite-element meshing, and point cloud sampling. Fig. 2(e) illustrates planar and 2.5D discretization. This representation supports AI-aided tasks like parasitic extraction, thermal-structural co-simulation, electromigration analysis, and manufacturability-aware optimization.

In AiEDA, we implement some design-to-vector approaches to convert complex design data into structured formats, efficiently address the key data challenges. To demonstrate this in practice, a complete, vectorized instance of the `gcd` design is available for inspection in our open-source repository[1].

III. AiEDA: AI-Aided Design Library

Fig. 3 illustrates the AAD library architecture for design-to-vector. The library comprises four essential components: 1) Flow Engines: supporting complete physical design workflows with flexible engine switching capabilities; 2) Data Generation: enabling programmatic control for flow execution and rich data extraction, facilitating efficient data acquisition; 3) Data Management: organizing workspaces for complex tool interactions and enabling batch vectorization through structured data organization; 4) Downstream Applications: providing unified interfaces for feature engineering and AI-based analysis, streamlining model training, validation, and evaluation. The following subsections detail these components.

A. Flow Engines

Flow engines form the foundational component for implementing and evaluating physical design processes. We categorize them into **tool engines** (implementing specific steps like placement and routing) and **evaluation engines** (assessing design features and performance metrics through processes like static timing analysis). Available engines include open-source options (e.g., OpenROAD, iEDA) and commercial alternatives (e.g., Innovus, PrimeTime). Additionally, specialized tools like DREAMPlace [22] for placement and CUGR [23] for routing often achieve superior results for specific steps.

Our library integrates common open-source engines, commercial tools, and specialized point tools. This integration relies on standardized data exchange formats (e.g., `.def`) to enable flexible tool switching—for instance, using DREAMPlace for placement while reverting to Innovus for routing. This eliminates integration complexities and lowers barriers for AI-EDA research, significantly expanding possibilities for comparative analysis. As most engines are implemented in C++, Python interface wrappers are required to ensure seamless integration within AAD environments. AiEDA integrates these diverse EDA tools as third-party libraries, managing and invoking them through a standardized and unified interface.

B. Data Generation

We encapsulate flow engines with Python interfaces for systematic data generation. The library provides two complementary APIs: **flow APIs** for design implementation and **data APIs** for data extraction. These interfaces support both fine-grained operations (e.g., legalization, wirelength calculation) and coarse-grained workflows (e.g., full placement, comprehensive evaluation).

Listing 1 demonstrates the unified data generation approach. Flow APIs enable flexible tool selection through input parameters—for instance, placement can utilize either iEDA's iPL or DREAMPlace. Data APIs specify evaluation tools and target design stages to extract stage-appropriate data. For example, for wirelength metrics, the system computes half-perimeter wirelength (HPWL) and rectilinear Steiner minimum tree (RSMT) during placement, while routed wirelength (RWL) is calculated after routing. For the same metric, multiple tools may be employed; for instance, DRC evaluation can employ either iEDA's iDRC or Innovus's verification interface.

Implementation complexity varies significantly across tool types. Open-source engines require only C++/Python linking via `pybind11`, while commercial tools demand TCL script

[1] https://github.com/OSCC-Project/AiEDA/tree/master/example/sky130_gcd/output/iEDA/vectors

```python
from aieda.workspace import workspace_create
from aieda.flows import RunFlow
from aieda.data import RunFeature

def generate_data(ws_dir, tool, step):
    ws = workspace_create(ws_dir, tool)
    match step:
        case "floorplan":
            flow = RunFlow.runFP(ws, tool)
            feat = RunFeature.fp(ws, tool)
        case "place":
            flow = RunFlow.runPL(ws, tool)
            feat = RunFeature.pl(ws, tool)
        case "cts":
            flow = RunFlow.runCTS(ws, tool)
            feat = RunFeature.cts(ws, tool)
        case "routing":
            flow = RunFlow.runRT(ws, tool)
            feat = RunFeature.rt(ws, tool)
        case _:
            flow = RunFlow.run(ws, tool)
            feat = RunFeature.flow(ws, tool)
    return flow, feat
```

Listing 1. Unified Python API usage for data generation.

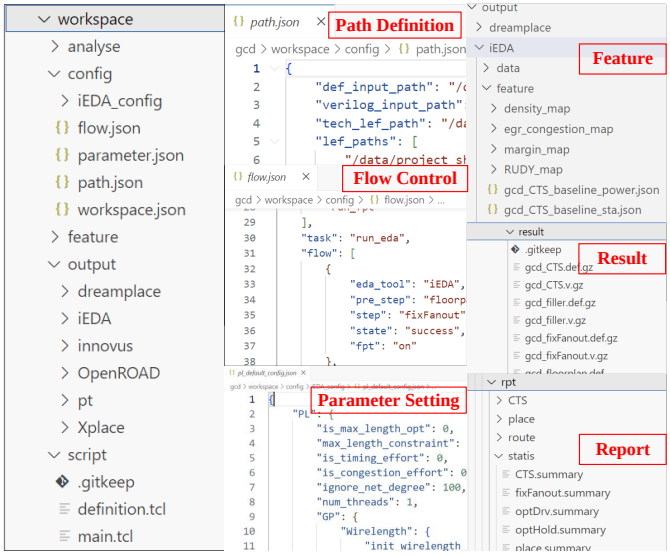


Fig. 4. Workspace architecture and file organization.

generation, execution, and report parsing—substantially increasing implementation overhead.

Regarding data output, flow APIs generate standardized design files (e.g., `.def/.v`). For comprehensive coverage, data APIs extract a wide range of information, spanning basic design statistics (e.g., cell counts, layout area), scalar metrics (e.g., wirelength, timing, power), and spatial data (e.g., congestion map, DRC hotspot).

C. Data Management

We propose two core concepts for comprehensive data management: **workspace** and **vectorization**. A workspace serves as a pre-configured environment that centralizes parameter settings, data access, and storage operations for each chip design. Vectorization transforms chip design data into structured representations compatible with standard neural network model input/output piplines. These enable standardized

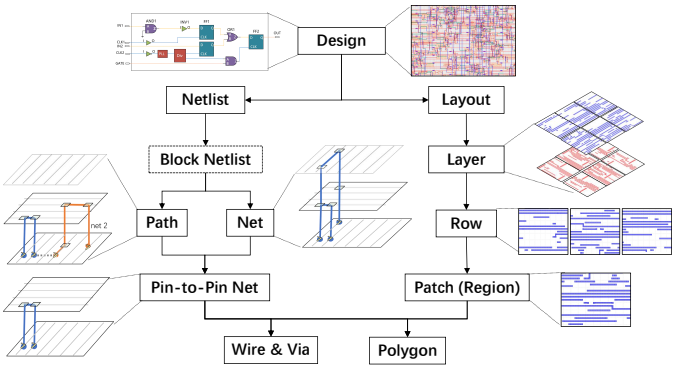


Fig. 5. Design-to-vector for chip data representation.

data management and high-throughput data generation through flexible Python interfaces.

1) Workspace: Fig.4 demonstrates the hierarchical file organization within a workspace. The three core components are detailed as follows:

a) Configuration: serves as the workspace core, managing path definitions for all input/output files (e.g., `.lef/.def/.lib`) and providing centralized control over flow engines, including engine selection and execution parameters. Tool-specific configurations (e.g., "*target_density*" for iEDA's iPL) are integrated through JSON files, ensuring interactive flexibility and workflow reproducibility.

b) Output Management: automatically routes generated data to corresponding output paths based on the selected engine. Data is systematically categorized into: (1) design files `.def/.v` stored in the result directory; (2) runtime files `.log/.rpt` archived in the report directory; and (3) extracted data `.json/.csv` organized in the feature directory. Users specify only the workspace root path while the system automatically handles file routing and organization.

c) Script Management: maintains execution scripts for commercial tool integration. When evaluating DRC with Innovus, the system generates TCL scripts comprising parameter definitions (`definition.tcl`) and tool execution commands (`main.tcl`). Parameter scripts configure input/output paths while execution scripts source these parameters and invoke tool-specific commands such as `verify_drc`.

2) Vectorization: Fig. 5 illustrates a hierarchical **design-to-vector** decomposition methodology for chip data representation. The methodology decomposes the overall design into two fundamental components: netlist and layout. The netlist captures logical connectivity between circuit cells, while the layout represents physical geometric information. Decomposition continues on both branches with increasing granularity. The netlist decomposes into path-level representations (encoding routing connectivity and signal propagation paths), and net-level representations (capturing individual point-to-point connection between specific pins). Recognizing that path-level representations can introduce redundancy due to overlapping segments, we also provide a holistic graph-level representation. This allows users to directly access global connectivity information in a complete and non-redundant format, serving as a macro-level counterpoint to the fine-grained path data. The layout decomposes into layer-level (manufacturing

layers such as metal and via layers) and patch-level (localized spatial regions) data. Both branches converge at the geometric primitive level, encompassing three fundamental elements: wires (linear conductors), vias (inter-layer connection), and polygons (geometric shapes).

This collection of structured, multi-level data constitutes what we term **Foundation Data**, pivotal for AI applications. It converts complex chip designs into a computationally efficient, structured representation that preserves high data integrity. Its inherent compatibility with AI processing frameworks enables seamless integration across all design abstraction levels. Ultimately, our vectorization methodology yields a unified, fine-grained data foundation that ensures consistent processing and supports diverse downstream applications.

In AiEDA, vectorization is streamlined through simple and intuitive interfaces, as demonstrated in Listing 2. The framework supports flexible granularity control, allowing users to generate vectors at different abstraction levels based on specific application requirements.

```python
from aieda.workspace import workspace_create
from aieda.data import RunVectors

def vectorize_data(ws_dir, tool, level):
    ws = workspace_create(ws_dir, tool)
    vectors = RunVectors(ws)
    vectors.read_def(ws.input_def)
    match level:
        case "net":
            vectors.generateNet(ws.vec_dir)
        case "graph":
            vectors.generateGraph(ws.vec_dir)
        case "path":
            vectors.generatePath(ws.vec_dir)
        case "patch":
            vectors.generatePatch(ws.vec_dir)
        case _:
            feat.generateVectors(ws.vec_dir)
```

Listing 2. Unified Python API usage for vectorization.

D. Downstream Application

We develop **process engines** that selectively extract and organize data from the Foundation Data, streamlining preparation for downstream tasks. This workflow is shown in Fig. 6. Note that the data structures depicted in Fig. 6 are illustrative examples and not an exhaustive list.

We categorize process engines into two types: **specific** and **general** engines. Specific engines comprise six types: design, net, graph, path, patch, and combined engines. These engines follow a consistent three-stage approach: (1) loading the Foundation Data to filter task-specific feature data using `load_data()`, (2) performing feature engineering and reorganizing the data into AI-ready formats (e.g., tabular, sequence, spatial) via `parse_data()`, and (3) providing vector data to AI models through `get_data()`. The combined engine, specifically, merges data from multi-level representations (e.g., net-level and patch-level) to form multi-modal representations. We also provide a general visual engine for analyzing dataset characteristics.

A key challenge in preparing this data is handling the variable dimensions inherent in different circuits and tasks

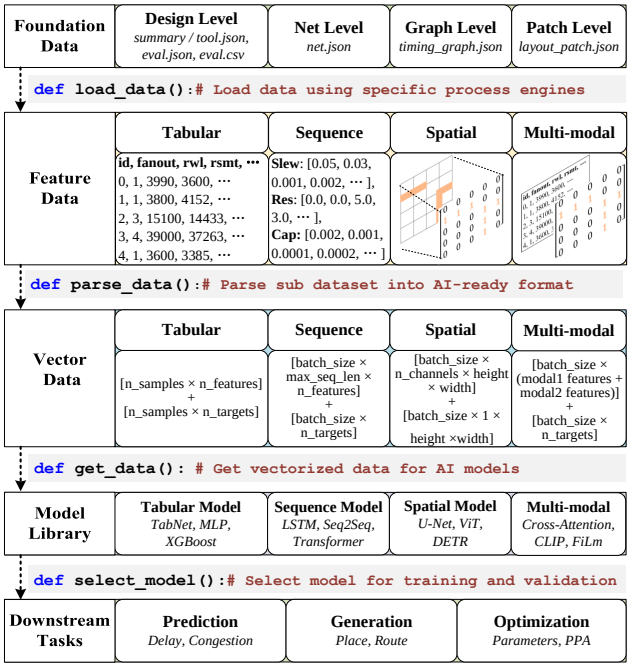


Fig. 6. Data processing pipeline for downstream tasks. The data structures depicted are illustrative examples and not an exhaustive list.

(e.g., varying net counts or path lengths). Our process engines resolve this by applying modality-specific techniques: sequential data is padded or truncated to a uniform length, spatial data is organized into fixed-size grids, and graph data is managed by native batching mechanisms (e.g., PyG). This ensures the data from `get_data()` is seamlessly compatible with mainstream AI frameworks, abstracting complex preprocessing from the end-user.

Furthermore, AiEDA supports loading diverse model libraries through `select_model()`. For each data modality, we provide multiple models for training and validation in a unified environment. The model library is extensible, accommodating both third-party libraries and user-defined custom models encapsulated as classes. To simplify development, AiEDA promotes a standardized "config → process → model → trainer" workflow. This pattern, inspired by leading AI frameworks, lowers the barrier for contributing new models by providing clear, established precedents.

By introducing process engines and model libraries, AiEDA standardizes the AAD pipeline from data loading to model validation, thereby advancing AAD methodology development.

IV. iDATA: Dataset for AI-Aided Design

A. Data Source and Statistics

This section introduces iDATA, a large-scale, open-source dataset built to facilitate AAD research. The complete dataset, derived from 50 real-world chip designs, totals approximately 600 GB of structured Foundation Data, excluding the original raw design files. Table III summarizes the statistical characteristics of our dataset comprising 50 designs. The dataset encompasses diverse chip designs including digital signal/image processors (DSP/ISP), peripheral/interface circuits, functional modules, memories, CPUs, GPUs, and SoCs,

TABLE III
STATISTICAL CHARACTERISTICS OF THE DATASET.

Circuits	Design			Net		Path		Patch	
	#Cells	#Nets	#Wires	#Files	Size	#Files	Size	#Files	Size
s713	135	125	1426	121	890K	56	676K	324	1.59M
apb4_rng	195	204	2230	169	1.39M	132	1.29M	441	2.45M
gcd	297	270	3733	270	2.32M	136	3.81M	625	3.85M
... 13 more designs ...									
ASIC	1228	796	10737	796	6.82M	76	660K	121	5.36M
s15850	2088	1926	27941	1925	17.33M	1724	21.00M	4225	27.66M
apb4_uart	5981	5606	83268	5555	53.45M	4652	99.96M	11449	83.34M
... 10 more designs ...									
jpeg	27671	29160	366397	29160	245.32M	18128	786.71M	66049	424.45M
eth_top	42279	38552	646875	38552	434.20M	20000	562.18M	169744	967.16M
yadan	63514	31280	483369	31280	331.30M	19832	816.37M	15376	313.25M
beihai	211236	133086	2161829	132424	1.53G	52628	4.87G	92256	1.60G
SHMS	268721	251772	3610024	251686	2.51G	70672	6.45G	19153	1.98G
nvdla	289344	226974	3708427	226904	2.69G	20000	1.21G	28224	2.40G
... 12 more designs ...									
nanhu-G	2793215	2646672	42524007	2643701	27.84G	20136	1.70G	75040	25.04G
openC910	3282828	2948743	52259408	2942510	36.25G	40588	2.85G	152100	33.82G
T1_sand	4816399	4728816	79050737	4728737	50.22G	40000	5.99G	77841	44.13G
Total	23.26M	21.47M	347.15M	21.45M	**235.91G**	1.63M	**149.87G**	1.61M	**207.18G**

primarily sourced from open repositories (OSCPU [24], OSCC [25], OpenLane [26], ISCAS89 [27], CHIPS Alliance [28]) and internal projects. All designs undergo RTL synthesis in 28nm technology, followed by complete physical design using **Innovus** and data extraction via **iEDA**. We chose Innovus for the physical design process because results from commercial tools are widely regarded as a high-quality "ground truth" in the AI research community, providing a stable and reliable benchmark. The iDATA dataset is the direct artifact generated from this workflow using our AiEDA library. Since routing results contain comprehensive physical design information, we construct our dataset based on routing outcomes, corresponding to design-level, net-level, graph-level, path-level, and patch-level vectorizations. Moreover, to support cross-stage predictive tasks, we have explicitly preserved key placement-stage features (e.g., cell density) within the Foundation Data.

Design-level vectorization captures basic statistics, tool metrics, and evaluation metrics. Table III presents a subset of basic statistical data, specifically cell, net, and wire counts. Cell counts range from 135 to 4,816,399; net counts from 125 to 4,728,816; and wire counts from 1,426 to 79,050,737. This broad spectrum demonstrates the extensive design scale coverage in our dataset.

Table III also provides statistics on file counts and storage sizes across net-level, path-level, and patch-level representations. For **net-level** data, *#Files* may be fewer than *#Nets* since files are not generated for nets without wires. For **path-level** data, we extract only paths containing driver pins, potentially resulting in significantly fewer files than *#Nets*. Designs with exceptionally large path counts utilize configurable parameters to limit path generation. To ensure data completeness, we generate a comprehensive **graph-level** representation for each netlist, capturing its full topology. For **patch-level** data, patch sizes are adjustable to accommodate different design scales. For small circuits, the default patch size is set to 9 times the pitch width, while for large designs, it is 180 times the pitch width. This configurability enables flexible dataset generation tailored to specific design requirements.

To assess computational requirements and scalability, vectorization was performed using 32 threads on an Intel Xeon Platinum 8268 CPU with 1.5TB RAM. Vectorization time ranges from tens of minutes to hours, depending on design complexity and parameter settings (e.g., patch size). The 1.5TB memory configuration proved sufficient even for our largest design (T1_sand), demonstrating scalability for incremental dataset expansion.

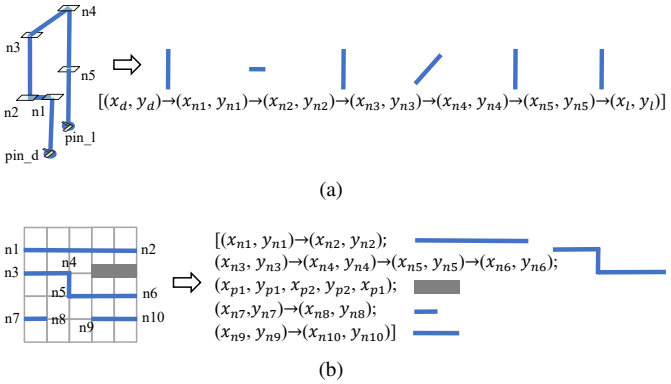


Fig. 7. Foundation Data generation. (a) Pin-to-pin net. (b) Patch.

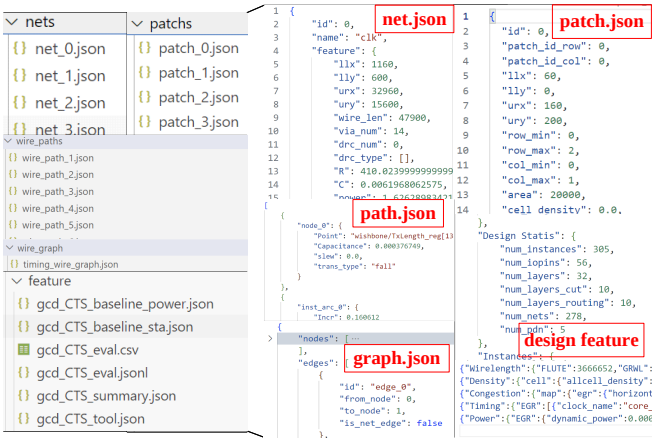


Fig. 8. File organization structure of the vectorized dataset.

B. Foundation Data Generation

Fig. 7 illustrates how Foundation Data is generated at the net, path, and patch levels. In Fig. 7(a), a pin-to-pin net connects driver pin *pin_d* to load pin *pin_l* through intermediate nodes *n1-n5*. This net is decomposed into six wire segments, which can be represented as a vector, as shown in Fig. 7(a). Following the decomposition approach described in Fig. 5, both paths and nets can be broken down into multiple pin-to-pin nets. Therefore, source paths and nets can be readily vectorized in the same manner as in Fig. 7(a). Additionally, Fig. 7(b) shows a patch consisting of four wire segments and an obstacle polygon, all stored in vector form. The vector includes the x and y coordinates of the connected nodes, along with the contour coordinates of the obstacle polygon.

We generate the Foundation Data using the method in Fig. 7 and store it in files as shown in Fig. 8. Table IV presents the key data at each hierarchical level. **Design** level provides basic statistics, tool metrics, and evaluation metrics with outputs ranging from scalar values to spatial maps. **Net** level stores

TABLE IV
DATA CONTENT VECTORIZED AT EACH HIERARCHICAL LEVELS.

Level	Key Data
Design	Layout dimensions, area utilization, net/cell counts and types, routing layers, pin distribution, tool runtime metrics (e.g., *"buffer_insertion"* during CTS), wirelength, density, timing, power, congestion maps
Net	Net name/features, pin information, bounding box, parasitic R/C, delay, slew, power, fanout, DRC counts/types, wire segments and geometric properties
Graph	Node definitions (unique ID, name), pin/port identifiers, and edge definitions specifying connectivity
Path	Node positions, capacitance, slew rates, transition types, cell/interconnect delays, edge decomposition, wire R/C parameters, input/output slew, wire delay parameters
Patch	Patch ID, position identifiers, boundaries, grid indices, density metrics, RUDY/EGR congestion, timing, power, IR drop, sub-net decomposition, layer-wise wire features

TABLE V
NET-LEVEL FIDELITY COMPARISON FOR THE `s713` DESIGN.

Metric	Original	Reconstructed	Fidelity Ratio
WNS (ns)	-0.972	-0.989	0.983
TNS (ns)	-3.887	-3.987	0.975
Violating Paths	7	7	1.000
Total Power (W)	0.0621	0.0622	0.998

individual net information including net features, parasitic parameters, and wire geometry details. **Graph** level represents the netlist as a topological graph, defining nodes and edges to capture the full circuit structure. **Path** level captures timing path data with detailed node properties, signal parameters, and decomposed interconnect information for each wire segment. **Patch** level employs uniform spatial division to generate patches containing position identifiers, quality indicators, and layer-specific features for localized analysis.

C. Fidelity and Accuracy Loss Validation

The design-to-vector paradigm presents a controllable trade-off between data fidelity and processing efficiency. This accuracy loss, primarily stemming from discretization techniques like layout gridding, is minimal and considered acceptable for AI-EDA tasks that prioritize spatial patterns over absolute coordinate precision. The process's high fidelity is illustrated with the `s713` designs.

At the **net-level** (logical topology), we reconstructed a DEF file from its Foundation Data (`net.json`) and compared it to the original. The results, summarized in Table V, revealed negligible deviation. Key metrics such as the number of timing-violating paths remained identical, while fidelity ratios for timing (WNS, TNS) and power approached 1.0, confirming the preservation of critical electrical characteristics. At the **patch-level** (geometric layout), a comparison between the original and the reconstructed cell density maps is shown in Fig. 9. Despite a reduction in resolution, the reconstructed map clearly preserved the original's key spatial features. This visual similarity was quantified by a high correlation coefficient of 0.993. These results confirm our process preserves critical electrical characteristics and spatial distributions, making the minimal accuracy loss an acceptable trade-off for the significant gains in research efficiency and accessibility.

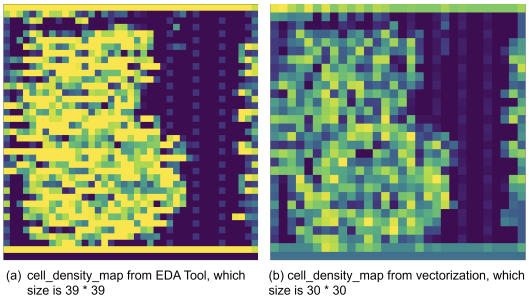


Fig. 9. Patch-level fidelity comparison for the `s713` design.

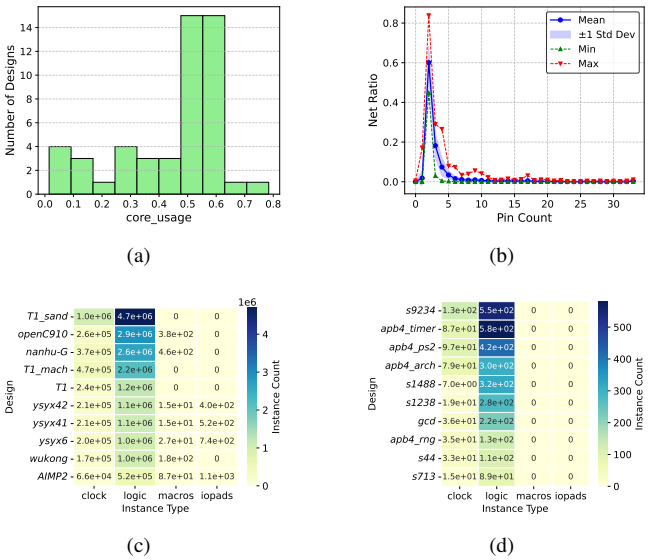
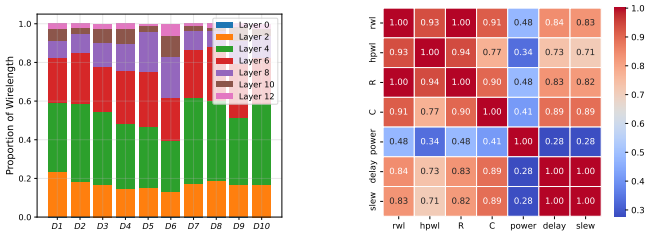


Fig. 10. Design-level characteristics. (a) Core usage. (b) Pin count. (c)-(d) Instance type distribution.

D. Data Insight Analysis

Leveraging our process engines (Fig. 6), we efficiently analyze dataset characteristics across multiple granularities. This analysis has a twofold objective: to provide a quantitative overview of the iDATA dataset and to establish a foundation for uncovering novel EDA insights. By creating a statistical baseline and validating fundamental domain principles, we enhance the dataset's accessibility and reliability for both AI researchers and EDA experts. Accordingly, our statistical analysis focuses on the feature-rich Design, Net, Path, and Patch levels. The Graph-level data is treated separately; as a pure topological representation (nodes and edges) intended for direct use in graph-centric AI models like GNNs, its value is structural rather than statistical. Therefore, it is excluded from the following characteristic summary.

Design-Level Characteristics. We examine three key metrics: core usage, pin count distribution, and instance type composition. Core usage represents the ratio of total instance area to core area. Fig. 10(a) shows that over half the designs exhibit core usage within 0.5-0.6. Pin count distribution analysis in Fig. 10(b) reveals that nets with 2-3 pins predominate, accounting for approximately 80% of all nets. The shaded regions represent ± 1 standard deviation across different pin counts. Instance types are categorized into four classes: clock,



(a) (b)

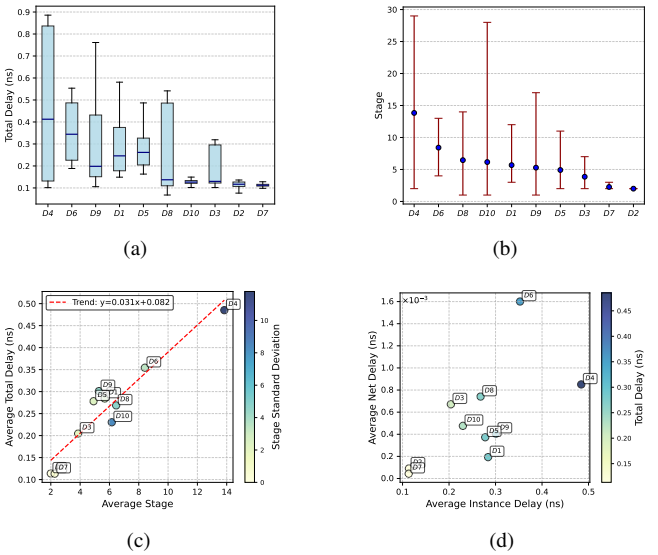
Fig. 11. Net-level characteristics. (a) Wirelength distribution across layers. (b) Correlation matrix of physical metrics (RWL, HPWL) and electrical parameters (R, C, power, delay, slew).

logic, macro, and IO pad. Fig. 10(c) and (d) present instance type distributions for the top 10 and bottom 10 designs by total instance count, respectively. The analysis shows clock instances constitute 14.93% on average, logic instances 85.04%, macro instances 0.01%, and IO pads 0.02%.

Net-Level Characteristics. We analyze both physical metrics (RWL, HPWL) and electrical parameters (R, C, dynamic power, average delay, and delta slew). Physical metrics characterize routing geometry, while electrical parameters reflect performance and timing characteristics. Fig. 11(a) illustrates wirelength distribution across layers for various designs. Odd-numbered layers represent vias, while even-numbered layers correspond to metal routing layers, with Layer 0 reserved for cell placement. The distribution shows that Layer 4 accommodates approximately 40% of total wirelength, while Layers 2 and 6 each account for approximately 20%. Wire utilization decreases with increasing layer numbers. Fig. 11(b) presents the correlation matrix, where the high correlation coefficient (0.93) between RWL and HPWL validates HPWL as an effective predictor for actual routing length, confirming the efficacy of HPWL optimization during placement. Both RWL and HPWL exhibit strong correlations with R and C, indicating that wirelength reduction effectively mitigates parasitic effects. The correlation analysis reveals that physical design optimization (reducing wirelength) leads to improved electrical performance through reduced parasitic C, which simultaneously enhances delay and signal transition quality.

Path-Level Characteristics. We analyze four critical timing metrics: instance delay, net delay, total delay, and stage count. Timing paths consist of instances interconnected by nets, with delays categorized as instance and net delays. Total delay accumulates instance and net delays along the path, while stage count indicates the number of instance-net pairs traversed. Fig. 12(a) and (b) present box plots of total delay and average stage count, revealing concentrated distributions within narrow ranges. Fig. 12(c) demonstrates an approximately linear relationship between total delay and average stage count, reflecting that increased stage count results in longer paths with higher cumulative delay. Fig. 12(d) shows the scatter plot of average instance delay versus average net delay, indicating both metrics distribute within specific ranges, with instance delay significantly exceeding net delay.

Patch-Level Characteristics. We examine characteristics from both macro and micro perspectives. At the macro level,



(a) (b) (c) (d)

Fig. 12. Path-level characteristics. (a) Total delay distribution. (b) Stage count distribution. (c) Total delay vs. stage count correlation. (d) Instance vs. net delay comparison.

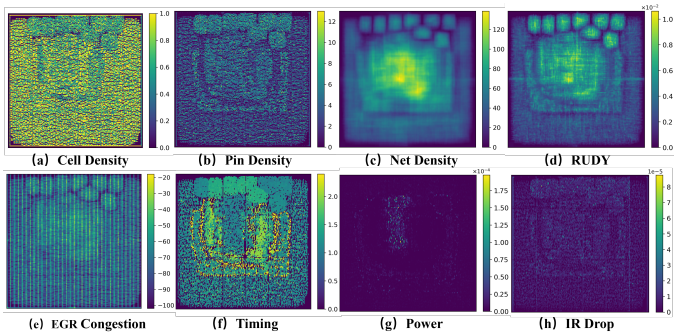
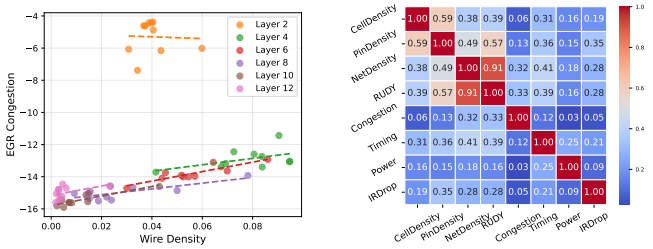


Fig. 13. Spatial distribution of key features in `aes` chip layout.

reassembled patches reconstruct complete design layouts. Fig. 13(a)-(h) visualize eight feature maps of the `aes` chip, enable layout-level tasks such as routability and IR drop prediction as demonstrated in CircuitNet [29]. At the micro level, each patch represents an individual sample. Fig. 14(a) demonstrates an approximately linear relationship between wire density and congestion across different layers, with congestion predominantly concentrated in Layer 2. Fig. 14(b) presents correlation analysis of patch features. Notably, congestion shows moderate correlation with net density (0.32) but weak correlation with cell density (0.06), indicating that optimizing cell density alone is insufficient for congestion mitigation in routability-driven placement.

Table VI compares existing open-source datasets for physical design across three dimensions: diversity, scale, and characteristics. While all these datasets provide raw data, they differ significantly in their focus and capabilities. CircuitNet [29], [30] extracts layout and graph features for prediction tasks but sacrifices substantial data information. EDA-Schema [31] specializes in metric extraction for prediction applications. ChipBench [32] provides comprehensive raw data but lacks extensive feature data, being primarily designed for metric analysis rather than AI tasks. In contrast, iDATA originates



(a) Wire Density vs. Congestion (b) Feature Correlation Analysis

Fig. 14. Patch-level characteristics. (a) Regression analysis between wire density and congestion across metal layers. (b) Correlation matrix of patch-level features.

TABLE VI
COMPARISON OF OPEN-SOURCE DATASETS FOR PHYSICAL DESIGN. SCALE MEANS THE RANGE OF CELL COUNTS. ALL DATASETS PROVIDE RAW DATA.

Datasets	Chips	Scale	Characteristic
CircuitNet [29], [30]	8	46K-1.48M	Layout/Graph Features
EDA-Schema [31]	20	0.47K-0.12M	Design Process Metrics
ChipBench [32]	20	0.82K-0.86M	Raw Data Only
iDATA (This work)	50	0.14K-4.82M	Foundation Data

from more diverse and larger-scale designs and uniquely provides Foundation Data. This representation retains rich, original design information, setting it apart from the task-specific and often information-lossy Feature Data common to other approaches (as illustrated in Fig. 15). The richness of this Foundation Data makes iDATA highly versatile and directly applicable to a wide range of tasks, including analysis, prediction, optimization, and generation. Consequently, our approach lowers the development effort for new AI applications by providing a more powerful and flexible starting point.

V. AI-AIDED DESIGN TASKS AND RESULTS

A key advantage of the AiEDA library and the iDATA dataset is their native support for multi-modal and multi-source AI-EDA tasks. By preserving the unique identifiers and hierarchical relationships between different design objects (e.g., nets, patches, patches, and graph) during vectorization, AiEDA facilitates the fusion of data from logical, physical, and timing domains. This aligned data structure empowers researchers to easily implement sophisticated feature engineering and explore advanced model architectures. The applications in this section put these principles into practice, demonstrating tasks that leverage data from a single source as well as those that benefit from the fusion of multiple data modalities.

All experiments presented in this section are built upon this principle. They leverage iDATA's **Foundation Data** and AiEDA's **process engines** to load and prepare data for the models. This approach significantly reduces the development effort compared to other frameworks, such as CircuitNet or CircuitOps, where developing a new task would necessitate building a data processing pipeline from the complex raw data from scratch. The primary goal of these applications is to demonstrate the versatility of our framework through accessible **proof-of-concept** examples. They are intended to showcase a breadth of capabilities, rather than to establish

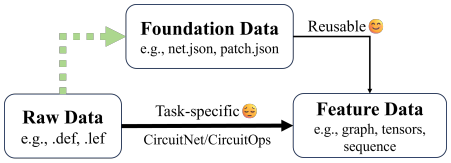


Fig. 15. The relationship between Raw Data, Foundation Data, and Feature Data, and the associated development effort for new AI tasks. The thicker the line, the greater the develop effort.

new state-of-the-art results, which we view as important future work this framework now helps facilitate.

To ensure representativeness, our downstream tasks cover three AI categories (prediction, generation, optimization) across five vectorization levels (design, net, graph, path, patch). This section details five representative downstream tasks to demonstrate the effectiveness of our AAD library and dataset, including methodologies and experimental results. All experiments were conducted on a system equipped with Intel Xeon Platinum 8380 CPU@2.30GHz (160 cores), 512 GB RAM, and NVIDIA A100 GPU (40 GB VRAM), running Ubuntu 18.04.5 with PyTorch 2.5.1 and CUDA 12.0.

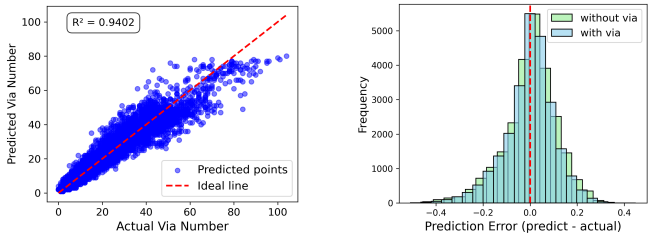
A. Net Level Wirelength Prediction

1) Methodology: This task estimates the wirelength ratio (RWL/RSMT) during placement. We construct the dataset from the top 30 circuit designs in Table III, merging their nets and randomly splitting into 80% training (>100,000 nets) and 20% testing (>25,000 nets). Each net sample is organized as tabular data with corresponding features and labels.

We employ TabNet [33] as the base model for its superior tabular data handling and feature selection capabilities. Our approach implements a two-stage prediction framework: (1) predicting via count using placement-stage features (aspect ratio, fanout, HPWL, RSMT, L-ness [34]), and (2) predicting wirelength ratio using both placement-stage features and the predicted via count as input. Both models are trained with mean squared error (MSE) loss.

2) Experimental Results: As shown in Fig. 16(a), our via count prediction achieves robust performance with $R^2 = 0.94$. Fig. 16(b) compares error distributions for wirelength ratio models with and without via count features. The leftward shift indicates improved accuracy. Specifically, incorporating via prediction reduces mean relative error (MRE) by 6%. This validates our framework's effectiveness in leveraging intermediate physical design knowledge to enhance prediction performance.

Notably, the trained wirelength prediction model can be exported to the standard ONNX format and integrated into a C++-based detailed placement engine (e.g., iEDA) via ONNX Runtime. During optimization, the engine can then query the ONNX model in real-time, replacing low-fidelity heuristics like HPWL with a far more accurate wirelength evaluation for potential cell swaps. This workflow exemplifies AiEDA's function as a critical **bridge**, connecting AI models trained on its vectorized data directly back into the physical design loop for **online optimization**. Enabling such closed-loop, AI-driven optimization is a core objective of our future work.



(a) (b)

Fig. 16. Net level wirelength prediction results. (a) Via count prediction. (b) Error distribution with/without via features.

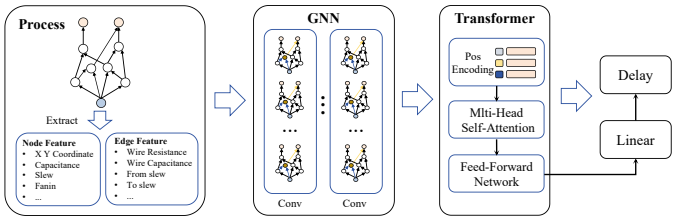
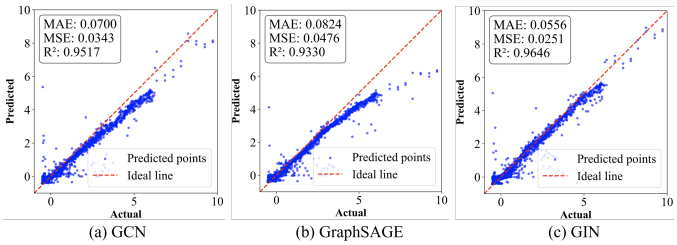


Fig. 17. Graph level delay prediction framework.



(a) GCN (b) GraphSAGE (c) GIN

Fig. 18. Scatter plot comparison of the prediction results of three GNNs combined with Transformer on the test data.

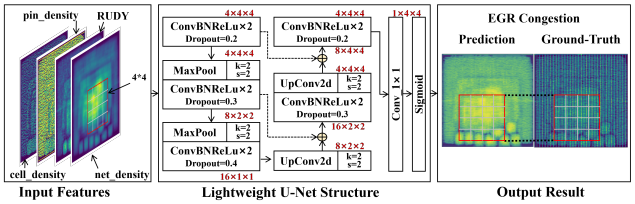


Fig. 19. U-Net architecture with multi-feature input and sliding window processing for patch level congestion prediction.

B. Graph Level Delay Prediction

1) Methodology: This task predicts node-wise incremental delays on locally-extracted critical-path subgraphs. We extract these subgraphs from the top-30 designs listed in Table III by aggregating all timing paths that shared a common clock source. Vertices and edges are annotated with their corresponding physical and electrical attributes (e.g., coordinates, capacitance, resistance, slew). For training, node delays are computed, log-transformed, and then normalized on a per-design basis. The final dataset is split into training (70%), validation (15%), and test (15%) sets, ensuring that all paths from a single design reside in the same split.

As illustrated in Fig. 17, our approach employs a GNN-Transformer architecture. The model processes annotated netlists where nodes and edges are represented by feature vectors. First, a configurable GNN encoder (GCN, GraphSAGE, or GIN) captures local topological and electrical context, producing initial node embeddings. Second, these embeddings are augmented with sinusoidal graph-Laplacian positional encodings and fed into a Transformer encoder. This allows the model to capture long-range dependencies along timing paths via its multi-head self-attention mechanism. Finally, a lightweight MLP maps the final node embeddings to the predicted delay values.

2) Experimental Results: The performance and the prediction scatter plots of the three GNN encoders on the test set are shown in Fig. 18. All models achieve strong results ($R^2 > 0.93$), demonstrating the effectiveness of GNN-based architectures for timing prediction. Among them, GIN delivers state-of-the-art performance, achieving an MSE of 2.51% and an R^2 of 0.9646. This represents a substantial improvement over the next-best model, GCN, with 26.7% lower MSE and 20.6% lower MAE. This advantage stems from GIN's powerful injective aggregation function, which allows it to distinguish subtle local sub-structures that influence delay. Conversely, the simple mean pooling of GraphSAGE proves less effective, resulting in the highest error and underscoring its inadequacy for tasks requiring sign-off-level precision.

C. Patch Level Congestion Prediction

1) Methodology: To predict the early global routing (EGR) congestion map, we use a set of features that are all obtainable from the placement stage, including cell density, pin density, net density, and RUDY [35]. We construct our dataset from the top 30 designs listed in Table III. As the number of patches per design varies significantly, a simple random split is suboptimal. To ensure our training, validation, and test sets are representative of this diversity, we employ a stratified sampling strategy. We group the designs into three strata based on patch count (large, medium, and small) and then randomly sample from each stratum to create a 19-design training set, a 3-design validation set, and an 8-design test set.

Fig. 19 illustrates our data processing and baseline model architecture. We implement a sliding window approach with 4×4 patches and stride of 3 to extract input features, effectively capturing local spatial information and addressing inconsistent input dimensions across designs. After standard normalization, we design a lightweight U-Net model with end-to-end training using MSE loss for pixel-wise congestion regression.

2) Experimental Results: The model achieves an average normalized root mean squared error (NRMSE) of 0.18 on the test set. While the model tends to overestimate congestion, it accurately preserves relative spatial patterns, which is sufficient for congestion-aware optimization. We further conduct two comparative experiments to evaluate optimization strategies. First, using robust normalization (RobustScaler) resulted a suboptimal average NRMSE of 0.23. In contrast, enhancing the model capacity (expanding channels from 4→8→16 to 16→32→64) successfully reduced the average NRMSE to 0.17. We observe that the model performs notably better on medium and large-scale designs, achieving NRMSE values as low as 0.12. This improved performance is likely attributable to the richer and more diverse congestion patterns present in

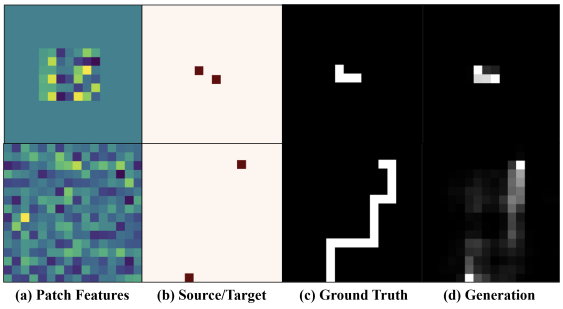


Fig. 20. Qualitative results of net map generation for short and long paths.

larger layouts, which provide more comprehensive learning opportunities for the model.

D. Net and Map Level Routing Mask Generation

1) Methodology: This task generates a two-pin net routing map using net-level data (source and target points) with 4 spatial patch features (cell density, pin density, net density, RUDY) in a multi-modal representation. These features are all obtainable from the placement stage. We employ 20 designs from the `apb4` series and `s` series as our dataset. Following the stratified sampling methodology detailed in Section V-C, the dataset is partitioned into a 12-design training set, a 3-design validation set, and a 5-design test set.

Net-level source and target points undergo one-hot encoding. Patch features are processed through intra-design normalization and spatial relative feature computation, yielding a 8-dimensional composite patch feature vector. All patch regions are resized to 16×16 grids for standardized processing.

We adopt a U-Net architecture as our base model. The encoder and decoder paths are represented as:

$$\{\mathbf{E}_1, \mathbf{E}_2, \mathbf{B}\} = \text{Encoder}(\mathbf{X}; 10 \to 16 \to 32 \to 64) \quad (1)$$

$$\mathbf{Y} = \text{Decoder}(\mathbf{B}, \{\mathbf{E}_1, \mathbf{E}_2\}; 64 \to 32 \to 16 \to 1) \quad (2)$$

The encoder progressively downsamples through double convolution blocks and max pooling to extract multi-scale features, while the decoder upsamples via transposed convolution and combines skip connections to recover spatial details. The input $\mathbf{X} \in \mathbb{R}^{10\times16\times16}$ contains 8-dimensional composite features and 2-dimensional source-target encodings, producing output $\mathbf{Y} \in \mathbb{R}^{1\times16\times16}$ representing the path probability map. The model is trained using binary cross-entropy loss:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\sigma(\hat{y}_i)) + (1-y_i)\log(1-\sigma(\hat{y}_i))] \quad (3)$$

where N is the total number of patches, y_i is the ground truth, $\hat{y}_i$ is the predicted logit value, and σ is the sigmoid function.

2) Experimental Results: The model generates a routing probability map for each patch. For evaluation, these probabilities are binarized using a threshold (default = 0.4), where 1 signifies a generated path. On the test set, the model achieves an 81% F1-score and a 67% Intersection over Union (IoU), indicating a reasonable spatial overlap with the ground truth. Fig. 20 demonstrates routing generation for nets with both short and long path lengths. As illustrated, for short-distance nets, the generated path closely matches the ground truth. However, for long-distance nets, while the model correctly captures the overall trajectory, the lower probabilities in the intermediate segments can result in discontinuous paths upon binarization. In a practical application scenario, the raw (i.e., non-binarized) probability maps from multiple nets can be aggregated to estimate overall routing density, which provides predictive insights at earlier design stages; for instance, by serving as a fast congestion predictor to guide routability-driven placement or acting as a look-ahead engine in early-stage global routing to anticipate resource contention.

TABLE VII
COMPARISON OF DEFAULT AND OPTIMIZED PARAMETERS ACROSS DIFFERENT DESIGNS.

Design	HPWL ↓		WNS ↑		TNS ↑	
	Default	Tuning	Default	Tuning	Default	Tuning
`s713`	1.24M	**1.17M**	-0.84	**0.33**	-4.78	**0.00**
`s1238`	**2.69M**	2.77M	-0.05	**0.29**	-0.05	**0.00**
`s1488`	**3.51M**	3.67M	-0.20	**0.16**	-0.87	**0.00**
`s9234`	6.74M	**5.30M**	-0.30	**-0.09**	-2.33	**-1.16**
`s13207`	5.94M	**5.81M**	0.44	**0.46**	0.00	**0.00**
`apb4_ps2`	4.76M	**4.00M**	0.09	**0.90**	0.00	**0.00**
`apb4_timer`	9.81M	**6.99M**	0.13	**0.59**	0.00	**0.00**
`apb4_i2c`	10.11M	**6.60M**	0.12	**0.65**	0.00	**0.00**
`apb4_pwm`	13.77M	**8.92M**	0.21	**0.63**	0.00	**0.00**
`apb4_wdg`	14.74M	**9.69M**	0.10	**0.51**	0.00	**0.00**
Impr. Rate	—	**8/10**	—	**10/10**	—	**10/10**

TABLE VIII
SINGLE-OBJECTIVE (HPWL) OPTIMIZATION ON LARGER DESIGNS.

Design	**# Cells**	**Default HPWL**	**Tuned HPWL**	**Impr.**
`jpeg`	27.7k	1118.1M	**312.6M**	72%
`eth_top`	42.3k	1410.3M	**691.4M**	51%
`yadan`	63.5k	1469.2M	**1155.7M**	21%
`SHMS`	268.7k	5669.5M	**4661.5M**	18%
`nvdla`	289.3k	26677.6M	**14253.1M**	47%

E. Design Level Parameter Optimization

1) Methodology: This task optimizes tool parameters to achieve superior performance across design stages. We select the iEDA placement engine as the optimization target. We employ the multi-objective tree-structured Parzen estimator (MOTPE) algorithm to optimize key placement metrics, including HPWL, WNS, and TNS. These metrics are extracted from design-level evaluation files (`eval.json`). In each iteration, MOTPE predicts parameter values that maximize expected improvement (EI) based on historical parameter-metric pairs. The predicted parameters are then applied to the placement tool to generate evaluation metrics. Subsequently, the new parameters and corresponding metrics are added to the historical dataset. We set the iteration number to 100. Upon completion, we collect metrics on the Pareto frontier as experimental results.

2) Experimental Results: Table VII compares default settings with optimized parameters across multiple designs. Our approach achieves improvements in HPWL (8/10 designs), WNS (10/10 designs), and TNS (10/10 designs), validating the effectiveness of our parameter optimization methodology. To

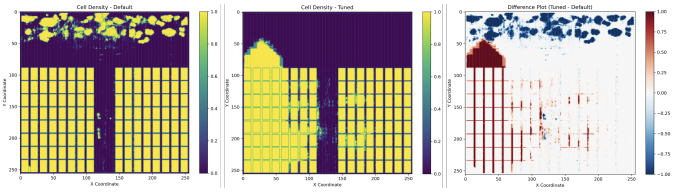


Fig. 21. Cell density heatmaps for the "nvdla" design before (left) and after (middle) single-objective HPWL optimization. The difference plot (right) visually confirms this dramatic shift in cell placement.

demonstrate the scalability of our methodology, we simplified the problem to a single-objective optimization (HPWL) and ran 100 DSE iterations for several large-scale designs, as shown in Table VIII. These new results demonstrate that our DSE methodology is highly effective on large designs, achieving significant wirelength improvements (up to 72%). Fig. 21 shows the cell density for "nvdla" before and after tuning. We note that the scalability of this multi-objective DSE is currently limited by the runtime of the underlying academic placer, which is why we focused on these smaller benchmarks for the comprehensive multi-objective evaluation. A key direction for future work is to integrate this DSE framework with a high-performance placer, such as DREAMPlace, to enable large-scale multi-objective optimization.

Beyond the downstream tasks previously discussed, our AiEDA framework provides broader foundational support for AI-driven EDA applications. It generates essential vectorized data, offers a versatile Python API, and manages the complete neural network training lifecycle. These capabilities enable a broad spectrum of AI-enhanced EDA applications, including DRC prediction [36], Steiner tree generation [37], 3D capacitance extraction [38], timing prediction [39], timing optimization [40], IR drop calculation [41], parameter optimization [42], and technology mapping [43]. Furthermore, its multi-engine capabilities extend its utility beyond the AI community to the broader EDA community for tasks such as tool metrics benchmarking.

VI. Conclusion

In this work, we introduce our open-source AI-aided design (AAD) library (AiEDA) and release the iDATA dataset. AiEDA integrates multiple design-to-vector techniques, which converts chip design data into structured representations compatible with neural network model input/output pipelines. AiEDA provides unified workflows from design execution to AI integration, while iDATA contains 600GB of multi-level structured data from 50 real designs. Through five representative downstream tasks, we demonstrated the effectiveness of our approach across design-level, net-level, graph/path-level, and patch-level representations. This work provides the first unified library that addresses complete data pipeline challenges in AAD, enabling researchers worldwide to advance AI-aided design automation more effectively. To foster collaboration, its open-source and modular architecture is designed to be highly extensible, welcoming community contributions to both the library's functionalities and the growing iDATA ecosystem. Future work will focus on expanding vectorization capabilities (e.g., layout-to-vector), improving dataset generation efficiency, integrating additional downstream tasks, and establishing tighter EDA flow coupling for optimization guidance. We believe this work will accelerate AI development and adoption in the EDA domain.

References

[1] Y. Du, Z. Guo, X. Jiang *et al.*, "PowPrediCT: Cross-Stage Power Prediction with Circuit-Transformation-Aware Learning," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2024, pp. 1–6.

[2] M. Wang, Y. Cheng, Y. Lin *et al.*, "MAUnet: Multiscale Attention U-Net for Effective IR Drop Prediction," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2024, pp. 1–6.

[3] H. Park, K. Baek, S. Kim *et al.*, "Pin Accessibility and Routing Congestion Aware DRC Hotspot Prediction for Designs in Advanced Technology Nodes With Consolidated Practical Applicability and Sustainability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 12, pp. 4786–4799, 2024.

[4] J. Ahn, K. Chang, K.-M. Choi *et al.*, "DTOC-P: Deep-Learning-Driven Timing Optimization Using Commercial EDA Tool With Practicality Enhancement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 8, pp. 2493–2506, 2024.

[5] J. Chen, J. Kuang, G. Zhao *et al.*, "PROS 2.0: A Plug-In for Routability Optimization and Routed Wirelength Estimation Using Deep Learning," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 1, pp. 164–177, 2023.

[6] H. Wu, Z. He, X. Zhang *et al.*, "ChatEDA: A Large Language Model Powered Autonomous Agent for EDA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 10, pp. 3184–3197, 2024.

[7] Y.-C. Lu, H. Ren, H.-H. Hsiao *et al.*, "GAN-Place: Advancing Open Source Placers to Commercial-quality Using Generative Adversarial Networks and Transfer Learning," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 2, pp. 32:1–32:17, 2024.

[8] L.-T. Chen, H.-R. Kuo *et al.*, "Arbitrary-size Multi-layer OARSMT RL Router Trained with Combinatorial Monte-Carlo Tree Search," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2024, pp. 1–6.

[9] M. Zhang, Z. Zhang, Y. Niu *et al.*, "Fast Constraints Tuning via Transfer Learning and Multiobjective Optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 9, pp. 2705–2718, 2024.

[10] B. Yang, Q. Xu, H. Geng *et al.*, "Miracle: Multi-Action Reinforcement Learning-Based Chip Floorplanning Reasoner," in *Proc. IEEE/ACM Design Autom. Test Europe (DATE)*, 2024, pp. 1–6.

[11] A. B. Kahng, "Solvers, Engines, Tools and Flows: The Next Wave for AI/ML in Physical Design," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2024, pp. 117–124.

[12] M. Shalan and T. Edwards, "Building OpenLANE: A 130nm OpenROAD-based Tapeout- Proven Flow," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2020, pp. 1–6.

[13] A. Olofsson, W. Ransohoff, and N. Moroze, "A distributed approach to silicon compilation: Invited," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2022, p. 1343–1346.

[14] T. Ajayi, V. A. Chhabria, M. Fogaça *et al.*, "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2019, pp. 1–4.

[15] X. Li, Z. Huang, S. Tao *et al.*, "iEDA: An Open-source Infrastructure of EDA," in *Proc. IEEE/ACM Asia South Pac. Design Autom. Conf. (ASPDAC)*, 2024, pp. 77–82.

[16] X. Li, S. Tao, S. Chen *et al.*, "iPD: An Open-source intelligent Physical Design Toolchain," in *Proc. of IEEE/ACM Asia South Pac. Design Autom. Conf. (ASP-DAC)*, 2024, pp. 83–88.

[17] J. Jung, A. B. Kahng, S. Kim *et al.*, "METRICS2.1 and Flow Tuning in the IEEE CEDA Robust Design Flow and OpenROAD ICCAD Special Session Paper," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2021, pp. 1–9.

[18] R. Liang, A. Agnesina, G. Pradipta *et al.*, "CircuitOps: An ML Infrastructure Enabling Generative AI for VLSI Circuit Optimization," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2023, pp. 1–6.

[19] V. A. Chhabria, W. Jiang, A. B. Kahng *et al.*, "OpenROAD and CircuitOps: Infrastructure for ML EDA Research and Education," in *Proc. IEEE VLSI Test Symp. (VTS)*, 2024, pp. 1–4.

[20] T. Fontana, R. Netto, V. Livramento, and et al., "How Game Engines Can Inspire EDA Tools Development: A use case for an open-source physical design library," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2017, pp. 25–31.

[21] H. Ren and J. Hu, Eds., *Machine Learning Applications in Electronic Design Automation*. Springer International Publishing, 2022.

[22] Y. Lin, Z. Jiang *et al.*, "DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 4, pp. 748–761, 2021.

[23] J. Liu, C.-W. Pui, F. Wang *et al.*, "CUGR: Detailed-Routability-Driven 3D Global Routing with Probabilistic Resource Model," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

[24] "Open Source Chip Project by University (OSCPU)." [Online]. Available: https://github.com/OSCPU

[25] "OSCC IP Project." [Online]. Available: https://github.com/oscc-ip

[26] "efabless/openlane2-ci-designs: Continuous Integration Designs for OpenLane 2.0.0 or higher." [Online]. Available: https://github.com/efabless/openlane2-ci-designs

[27] "ispras/hdl-benchmarks: Collection of Digital Hardware Modules & Projects (Benchmarks)." [Online]. Available: https://github.com/ispras/hdl-benchmarks

[28] "CHIPS Alliance." [Online]. Available: https://github.com/chipsalliance

[29] Z. Chai, Y. Zhao, W. Liu *et al.*, "CircuitNet: An Open-Source Dataset for Machine Learning in VLSI CAD Applications With Improved Domain-Specific Evaluation Metric and Learning Strategies," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 12, pp. 5034–5047, 2023.

[30] X. Jiang, Z. Chai, Y. Zhao *et al.*, "CircuitNet 2.0: An Advanced Dataset for Promoting Machine Learning Innovations in Realistic Chip Design Environment," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2023.

[31] P. Shrestha, A. Aversa, S. Phatharodom *et al.*, "EDA-schema: A Graph Datamodel Schema and Open Dataset for Digital Design Automation," in *Proc. ACM Great Lakes Symp. VLSI (GLSVLSI)*, 2024, pp. 69–77.

[32] Z. Wang, Z. Geng *et al.*, "Benchmarking End-To-End Performance of AI-Based Chip Placement Algorithms," *arXiv:2407.15026*, 2024.

[33] S. Ö. Arik and T. Pfister, "Tabnet: Attentive Interpretable Tabular Learning," in *Proc. AAAI Conf. Artif. Intell. (AAAI)*, vol. 35, no. 8, 2021, pp. 6679–6687.

[34] A. B. Kahng, C. Moyes *et al.*, "Wot the L: Analysis of Real versus Random Placed Nets, and Implications for Steiner Tree Heuristics," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2018, pp. 2–9.

[35] P. Spindler and F. M. Johannes, "Fast and Accurate Routing Demand Estimation for Efficient Routability-driven Placement," in *Proc. IEEE/ACM Design Autom. Test Europe (DATE)*, 2007, pp. 1–6.

[36] Y. Li, R. Liu, Z. Zeng *et al.*, "AiDRC: Accelerating Detailed Routing by AI-Driven Design Rule Violation Prediction and Checking," *ACM Trans. Des. Autom. Electron. Syst.*, 2025.

[37] R. Liu, S. Ding, J. Sui *et al.*, "NeuralSteiner: Learning Steiner Tree for Overflow-avoiding Global Routing in Chip Design," *Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 37, pp. 127 346–127 368, 2024.

[38] Y. Cai, Y. Liang, Z. Luo, B. Xie, and X. Li, "PCT-Cap: Point Cloud Transformer for Accurate 3D Capacitance Extraction," in *Proc. of IEEE Int. Symp. EDA (ISEDA)*, 2024, pp. 1–6.

[39] H. Liu, Z. Zeng, S. Tao *et al.*, "AiTPO: KAN-UNet Heterogeneous Network for Timing Prediction and Optimization at Global Routing," *ACM Trans. Des. Autom. Electron. Syst.*, 2025.

[40] H. Wu, Z. Huang, X. Li, and W. Zhu, "AiTO: Simultaneous gate sizing and buffer insertion for timing optimization with GNNs and RL," *Integration, the VLSI Journal*, vol. 98, p. 102211, 2024.

[41] H. Liu, Y. Xu, S. Tao *et al.*, "Simultaneous Conjugate Gradient and iAFF-UNet for Accurate IR Drop Calculation," in *Proc. of IEEE Int. Conf. Comput. Design (ICCD)*. IEEE, 2024, pp. 665–672.

[42] X. Lai, M. Liu, X. Li *et al.*, "iPO: Constant Liar Parameter Optimization for Placement with Representation and Transfer Learning," *ACM Trans. Des. Autom. Electron. Syst.*, 2025.

[43] J. Liu, L. Ni, X. Li *et al.*, "AiMap: Learning to improve technology mapping for ASICs via delay prediction," in *Proc. of IEEE Int. Conf. Comput. Design (ICCD)*. IEEE, 2023, pp. 344–347.

Yihang Qiu received the B.E. and M.S. degrees from Guangdong University of Technology, Guangzhou, China, in 2020 and 2023, respectively. He is currently pursuing the Ph.D. degree with the University of Chinese Academy of Sciences, Beijing, China. He won the First Place Award at the ICCAD@CAD Contest in 2022. His research interests include physical design and AI for EDA.

Zengrong Huang received his M.S. degree from the School of Computer Science, Xidian University, Xi'an, China, in 2009. He is currently an Engineer at Pengcheng Laboratory. His research interests include physical design and AI for EDA.

Simin Tao received his M.S. degree from the School of Computer Science, Beijing Jiaotong University, Beijing, China, in 2010. He is currently an Engineer at Pengcheng Laboratory. His research interests include timing and power analysis.

Hongda Zhang received the B.E. degree from University of Electronic Science and Technology of China, Chengdu, China, in 2022. He received his M.S. degree from The University of New South Wales, Sydney, Australia, in 2024. He is currently pursuing the Ph.D. degree with the University of Chinese Academy of Sciences. His research interests include DSE and Large Language Model.

Weiguo Li received the M.S. degree in Mathematics and Applied Mathematics from Minnan Normal University, Zhangzhou, China, in 2024. He is currently pursuing the Ph.D. degree with South China University of Technology. His research interests include clock tree synthesis and AI for EDA.

Xinhua Lai is currently pursuing the Ph.D. degree with the University of Chinese Academy of Sciences. He has worked as a Research and Development Software Engineer with Huawei, Wuhan, Hubei, China. His research interests include design space exploration, machine learning, deep learning, large language model and optimization methods with applications in VLSI CAD.

Rui Wang received the B.E. degree in Mathematics and Applied Mathematics from Tianjin Normal University, Tianjin, China, in 2023. He is currently pursuing the M.S. degree in Computer Science and Technology at the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China. His research interests include logic synthesis and AI for EDA.

Weiqiang Wang (Member, IEEE) received the B.E. and M.S. degrees in computer science from Harbin Engineering University in 1995 and 1998, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences, China, in 2001. He is currently a professor with the School of Computer Science and Technology, University of Chinese Academy of Sciences. His research interests include AI for EDA, computer vision and machine learning.

Xingquan Li (Member, IEEE) received the B.E. and Ph.D. degree from Fuzhou University in 2013 and 2018, respectively. He is an Associate Professor at Pengcheng Laboratory. His research interests include EDA and AI for EDA. His team has developed an open-source iEDA infrastructure/toolchain. He has published over 70 papers, and received three First Place Awards from ICCAD@CAD Contest in 2017, 2018, and 2022. In 2020, he received the Application Award of Operations Research from the Operations Research Society of China, and the Best Paper Award from ISEDA 2023.