2024 25th International Symposium on Quality Electronic Design (ISQED) | 979-8-3503-0927-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISQED60706.2024.10528750

# FastPASE: An AI-Driven Fast PPA Speculation Engine for RTL Design Space Optimization

Akash Levy
*Dept. of Electrical Eng.*
*Stanford University*
Stanford, CA
akashl@stanford.edu

Joe Walston
*AI, Distinguished Arch.*
*Synopsys, Inc.*
Sunnyvale, CA
jwalston@synopsys.com

Sourav Samanta
*R&D Engineer*
*Synopsys, Inc.*
Sunnyvale, CA
sourav.samanta@synopsys.com

Priyanka Raina
*Dept. of Electrical Eng.*
*Stanford University*
Stanford, CA
praina@stanford.edu

Stelios Diamantidis
*AI, Distinguished Arch.*
*Synopsys, Inc.*
Sunnyvale, CA
stelix@synopsys.com

*Abstract*—**We develop a tool called FastPASE to rapidly predict design metrics such as power, performance, and area from RTL using graph convolutional networks. FastPASE encodes elaborated RTL netlists as dataflow graphs and applies fan-in/fan-out convolutions based on the connectivity of the nodes. We apply FastPASE to three design case studies: (1) a configurable single instruction multiple data (SIMD) IP, (2) SweRV, an open-source RISC-V core, and (3) randomly-generated combinational RTL. We find that depending on design size, FastPASE can produce predictions 16.7×-155× faster than a commercial physical design tool run up to placement, with average normalized mean absolute error (NMAE) of 13% across all metrics tested for SIMD unit and RISC-V core. On SweRV, we demonstrate that using FastPASE as a proxy metric engine during RTL design space optimization produces improved aggregate design scores in ∼13× less time than using full physical design runs. In this context, our tool enables RTL designers to quickly explore the RTL design space and determine the right parameter settings for a particular set of silicon design goals.**

*Index Terms*—**end-to-end machine learning, digital circuit design, electronic design automation, graph convolutional networks**

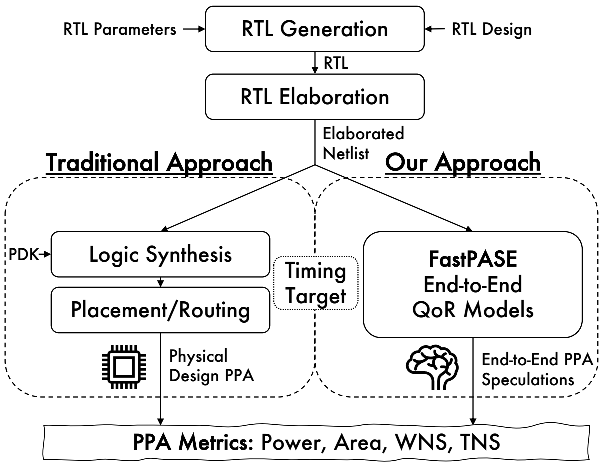


Fig. 1. A comparison between our proposed approach and the traditional approach to generating physical design PPA metrics.

## I. Introduction

We are entering an era where a single digital integrated circuit (IC) may contain 10s of billions of transistors, and a full physical design (PD) cycle for even a single block can take up to several hours (or even days). A big challenge that register transfer level (RTL) designers face today is rapidly evaluating how an RTL change (or a set of RTL changes) will affect the power, performance, and area (PPA) after physical design. Commercial electronic design automation (EDA) companies have developed tools to try to tackle this problem; Synopsys RTL Architect [1] allows RTL designers to estimate PPA ∼3× faster and within 5-10% of the final signoff PPA, and Cadence Joules RTL Power Solution [2] provides power back-annotation within 15% of the signoff value. These tools estimate PPA by emulating the physical design process at a coarse level of granularity, i.e., by developing rough floorplans and placement/routing estimates. However, calibrating these models to real signoff results can prove challenging, and the performance benefits may not be large enough (usually less than an order of magnitude) to enable deep design space exploration.

In this paper, we propose *Fast PPA Speculation Engine (FastPASE)* to make rapid end-to-end predictions about physical design PPA directly from RTL, through the use of specially-architected graph convolutional networks (GCNs) [3]. FastPASE trains on a few examples of an input design run through the physical design process on a particular technology, and thereafter makes speculations using only the elaborated netlist and clock period as inputs. Hence, during inference, technology-independent RTL elaboration (a relatively fast process) is the only prerequisite step for making predictions. The difference in our approach to making fast PPA predictions is illustrated in Fig. 1. In this work, we apply FastPASE to three test designs to showcase its efficacy: (1) a proprietary single instruction multiple data (SIMD) unit with several configuration knobs, (2) SweRV, Western Digital's open-source RISC-V core [4], and finally (3) randomly-generated combinational RTL from a regression suite, to evaluate model generalization.

The main contributions of this work are:

- We propose a graph representation for elaborated RTL netlists, where the feature vector in each node in the graph contains a one-hot encoding of its operation, concatenated with a graph-level vector to encode the target clock period, that is conducive to graph-based machine learning.
- We develop lightweight graph convolutional network models (∼40 kB model size per metric) that use *fan-in fan-out convolutions* to predict design PPA with high accuracy: normalized mean absolute error (NMAE) is between 1.65%-34.3% for power, 5.71%-18.6% for area, 2.54%-13.5% for total negative slack (TNS), and 4.21%-23.5% for worst negative slack (WNS). The average NMAE across these metrics is 13%. Furthermore, these PPA speculations are produced 16.7×-155× faster than Synopsys Fusion Compiler (run up to placement step), depending on the design [1]. We show that our models produce correlated speculations even on randomly-generated RTL.

- We present a real-world use case in which FastPASE is used "in the loop" as a proxy metric engine during design space optimization on the SweRV core. In this case study, FastPASE discovers designs with better aggregate design scores than those achieved by optimization using full physical design runs, and achieves $\sim 13\times$ speedup.

## II. Background & Related Work

With increasing size and complexity of digital chip designs, performing full physical design runs to evaluate PPA can become prohibitively slow. In this section, we briefly describe and summarize: (II-A) the physical design process today, (II-B) existing machine learning approaches for EDA, (II-C) graph machine learning techniques, and (II-D) design space optimization in EDA.

### A. Physical Design

Physical design of digital chips is typically performed using one or more commercial tools, state-of-the-art being Synopsys Fusion Compiler [1], Cadence Innovus [2], and Siemens Aprisa [5]. The physical design process is complex, involving many highly-tunable steps (e.g., synthesis, floorplanning, placement, clock tree synthesis, routing, etc.) and a large number of technology/design dependencies. Such complexity results in long run times and motivates the need for simpler/faster solutions to evaluate PPA.

### B. AI/ML-based EDA Techniques

There has recently been a trend towards incorporating machine learning (ML) and artificial intelligence (AI) techniques into various aspects of the EDA process to automate some components that have traditionally been handled by human designers [6], [7]. Core EDA algorithms, such as logic optimization, have been cast as problems that deep learning/reinforcement learning (RL) can target [8]–[10]. Google has developed a floorplanning technique that relies on deep RL [11], while NVIDIA is developing AI techniques in EDA for everything from predicting power/IR drop/parasitics to optimizing standard cell design [12]–[15]. The rapid influx of intelligent tools to aid chip designers across the EDA stack will continue to enable faster time-to-market for silicon products.

The most similar published work to our work presented here is SNS [16]. SNS has similar goals as this work, but uses different methods. SNS aims to predict the PPA of an arbitrary design after synthesis in order for designers to quickly understand the PPA impact of an RTL change. Meanwhile, FastPASE aims to predict physical PPA for the purpose of rapid design space optimization. PPA at the physical design stage is inherently harder to predict than synthesis PPA, since physical constraints must be considered. To deal with this, FastPASE employs *per-design model training* to effectively discover the quantitative relationship between the physical design process and PPA—SNS does not require any design-specific model training, meaning that an unseen design can be inferred on immediately. However, we believe that per-design training is an acceptable approach when targeting design space optimization (rather than simply evaluating an RTL change), since the number of examples required for training (typically 100-1,000) is relatively small compared to the total size of the design space (typically $>10^6$, often $>10^{20}$). SNS also differs from FastPASE in its prediction strategy, taking a path-based approach, rather than a graph-based one. It focuses on finding critical timing paths and aggregating randomly-sampled paths' characteristics to predict synthesis PPA. In this work, we show that explicit isolation of the critical timing path(s) may not be necessary when per-design training is employed, because timing dependencies can be learned from RTL design parameters and global graph structure.

There are several published works on PPA prediction for logic designs synthesized via high-level synthesis (HLS). High-level synthesis uses a high-level programming language (such as C or C++) to generate a gate-level hardware description. D-SAGE [17] utilizes graph neural networks (GNNs) to rapidly learn operation mapping patterns when applying HLS to modern field-programmable gate arrays (FPGAs). While the graph techniques described in D-SAGE are similar to the strategies we use in this work, the end goal is not to predict arbitrary PPA metrics, only delay. Two other works similarly examined HLS performance prediction using GNNs [18], [19], and these works also explore prediction of look-up table (LUT) count, digital signal processor (DSP) count, and flip-flop (FF) count.

We provide a comparison table summarizing the prior work and its relation to FastPASE in Section V-D.

### C. Machine Learning on Graphs

Automated machine learning on graphs is an active area of research across a multitude of disciplines [20], [21]. Graph-level tasks (such as the ones tackled in this work) may be performed with variants of graph neural networks (GNNs) [22]—these typically operate by constructing node-level features via matrix multiplication, then pooling/aggregating these node-level features to solve the task at hand. Graph convolutional networks (GCNs) are a variant in which node-level features are locally aggregated from neighboring nodes using a process that is analogous to a conventional convolution [3]. GCNs have already found numerous applications in EDA [23]—in this work, they are applied on elaborated netlists to serve as end-to-end physical design metric predictors.

### D. EDA Design Space Optimization (DSO)

The chip design process usually has a large number of "knobs" (tunable parameters), which is what makes it so difficult to find a good design configuration. The knobs start at the architecture level and go all the way down to the implementation/physical design level, as shown in Fig. 2. Examples of architectural design knobs include instruction set and memory/cache hierarchy while implementation/physical design knobs include frequency target and the chosen degree of design hierarchy flattening.

Computer architects have for decades been developing analytical techniques to find near-optimal solutions at the architectural level [24], [25]. Meanwhile, commercially available tools such as Cadence's Cerebrus [2] and Synopsys' DSO.ai [1] use reinforcement learning (RL) to search for optimization targets within the very large physical design solution space. However, there is a notable lack of solutions that target the microarchitectural/RTL implementation level. Knobs at this level include adder/multiplier architecture, clock gating strategies, and error correcting code (ECC) schemes. FastPASE mainly targets this level of abstraction in the chip design hierarchy, as shown in Fig. 2, though the approach we take allows accommodation of architectural/physical design parameters as well.

In this work, we include a case study in which we use a commercial RL-based optimization tool to search the RTL design space of a SweRV RISC-V core. Our tool, FastPASE, serves as a fast proxy metric generator and can be thought of as an

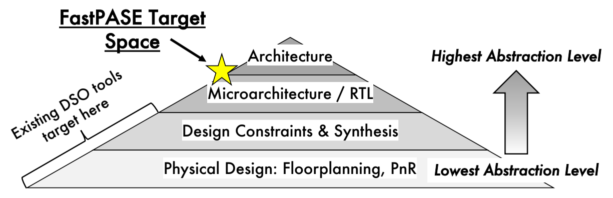

FastPASE Target Space
Architecture
Highest Abstraction Level
Existing DSO tools target here
Microarchitecture / RTL
Design Constraints & Synthesis
Physical Design: Floorplanning, PnR
Lowest Abstraction Level

Fig. 2. Chip design hierarchy. FastPASE targets exploration of the microarchitectural design space (i.e., the RTL implementation space), which is not often considered by existing DSO tools and computer architects.

AI-based "value function" for the reinforcement learning task of optimizing a design's RTL parameters.

## III. Datasets and Design Space Specifications

In this section, we describe the three datasets we use for testing and their respective RTL design spaces. We consider the design space for each design to be defined by its "permutons," i.e., the parameters/knobs controlling the RTL generation process (and also the clock period constraint). In order to keep the design space tractable, we constrain permutons to take on a finite number of discrete values. For categorical variables, this does not pose a challenge, but for parameters that usually exist over a continuous space, we can linearly or logarithmically discretize them between some finite bounds. For each design, we also vary the target clock period (`clk_period`) as a permuton between 2 ns and 10 ns to evaluate the design with low/high performance targets.

### A. `SIMDU`: Highly-Configurable Proprietary SIMD Unit

`SIMDU` is a highly-configurable SIMD unit consisting of 16 addressable arithmetic logic units (ALUs), in which several different FIFO, ALU function, and pipelining options exist for each ALU. After physical design on `TSMC5`, a single unit sampled from the design space has an average of ∼74K standard cells. A description of the design space is given in Table I.

TABLE I
PERMUTON DESCRIPTIONS FOR `SIMDU` DESIGN.

| Permuton | Type | Possible Values | Description | Size |
|---|---|---|---|---|
| *fifo_arch_{0-15}* | RTL | 0, 1, 2, 3 | Which FIFO architecture? | $4^{16}$ |
| *fn_arch_{0-15}* | RTL | 0, 1, 2 | Which ALU architecture? | $3^{16}$ |
| *fn_reg_{0-15}* | RTL | 0, 1, 2 | Whether ALU pipelined | $3^{16}$ |
| *clk_period* | PD | 2.0-10.0, steps of 0.2 | Target clock period (ns) | 41 |
| **Number of Design Space Dimensions** | | | | 49 |
| **Total Size of Design Space** | | | | 3.3E26 |

### B. `SweRV`: EH1 32-bit RISC-V Core

SweRV EH1 is a 32-bit CPU core which supports RISC-V's integer (I), compressed instruction (C), multiplication and division (M), and instruction-fetch fence and CSR instructions (Z) extensions. The core is a 9-stage, dual-issue, superscalar, mostly in-order pipeline with some out-of-order execution capability [4]. The core includes a branch predictor, optional ECC, a programmable interrupt controller (PIC), four system bus interfaces, and can reach up to 1GHz in 28nm technology. After physical design on `TSMC5`, a single core has on average ∼110K standard cells. The core presents a design of sufficient complexity and configurability that it becomes interesting to explore its vast design space. The core's permutons and their descriptions are given in Table II.

TABLE II
PERMUTON DESCRIPTIONS FOR `SweRV` RISC-V CORE DESIGN.

| Permuton | Type | Possible Values | Description | Size |
|---|---|---|---|---|
| *target* | RTL | default, default_ahb, hi_perf | Perf/power target | 3 |
| *btb_size* | RTL | 32, 48, 64, 128, 256, 512 | Size of BTB | 6 |
| *dccm_size* | RTL | 4, 8, 16, 32, 48, 64, 128, 256 | Size of DCCM | 6 |
| *icache_ecc* | RTL | 0, 1 | Use instr. cache ECC? | 2 |
| *icache_size* | RTL | 16, 32, 64, 128, 256 | Size of instr. cache | 5 |
| *pic_2cycle* | RTL | 0, 1 | Use 2-cycle PIC? | 2 |
| *pic_size* | RTL | 32, 64, 128, 256 | Size of PIC? | 4 |
| *mult_method* | RTL | 0, 1, 2, 3, 4 | Which multiply strategy? | 5 |
| *add_method* | RTL | 0, 1 | Which addition strategy? | 2 |
| *shift_method* | RTL | 0, 1, 2 | Which shifter strategy? | 3 |
| *clk_period* | PD | 2.0-10.0 in linear steps of 0.2 | Target clock period (ns) | 41 |
| **Number of Design Space Dimensions** | | | | 11 |
| **Total Size of Design Space** | | | | 1.1E7 |

### C. `RandRTL`: Random Combinational RTL

The final design we test consists of randomly-generated combinational RTL. The random RTL generator is highly configurable, generating a random set of procedural logic, including signed/unsigned arithmetic with varying bit widths, comparison, multiplexing, left/right shifting, type-casting, padding, and so forth. A random seed parameter is also used to arbitrarily increase the size of the design space by a factor of 10. `RandRTL` is implemented in `SAED32` technology, and the number of standard cells required to physically implement it varies widely from 1 all the way up to 190K, with an average of 8.2K. The descriptions of the design's permutons are given in Table III. The high-level purpose of this design is to demonstrate the predictive ability of FastPASE even with completely disparate (random) designs during training/inference.

TABLE III
PERMUTON DESCRIPTIONS FOR `RandRTL` DESIGN.

| Permuton | Type | Min | Max | Step | Description | Size |
|---|---|---|---|---|---|---|
| *num_stmts* | RTL | 1 | 10 | 1 | # of combinational statements | 10 |
| *num_terms* | RTL | 1 | 10 | 1 | Max number of terms on RHS of = | 10 |
| *num_temps* | RTL | 1 | 10 | 1 | Max number of temporaries | 10 |
| *case_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of a case statement | 21 |
| *if_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of an if statement | 21 |
| *temp_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of using temporary term | 21 |
| *const_prob* | RTL | 0 | 0.2 | 0.01 | Prob. term will be constant | 21 |
| *negconst_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of a negative constant | 21 |
| *negation_prob* | RTL | 0 | 0.2 | 0.01 | Prob. a term will be negated | 21 |
| *inp_width* | RTL | 3 | 16 | 1 | Maximum input operand width | 14 |
| *out_width* | RTL | 1 | 64 | 1 | Maximum output operand width | 64 |
| *mixed_prob* | RTL | 0 | 1 | 0.1 | Prob. of signed/unsigned mixing | 11 |
| *outtrunc_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of a bit-truncated output | 21 |
| *outlong_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of a bit-extended output | 21 |
| *reusesop_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of SOP reuse | 21 |
| *lowertrunc_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of lower-bit truncation | 21 |
| *uppertrunc_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of upper-bit truncation | 21 |
| *typecast_prob* | RTL | 0 | 0.5 | 0.05 | Prob. of a typecast | 11 |
| *compare_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of a compare operation | 21 |
| *conditional_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of a conditional operation | 21 |
| *shift_prob* | RTL | 0 | 0.2 | 0.01 | Prob. of a shift operation | 21 |
| *random_seed* | Seed | 1 | 10 | 1 | Random seed | 10 |
| *clk_period* | PD | 2 | 10 | 0.2 | Target clock period (ns) | 41 |
| **Number of Design Space Dimensions** | | | | | | 23 |
| **Total Size of Design Space** | | | | | | 1.4E29 |

## IV. FastPASE Approach

In this section, we describe the FastPASE network architecture, the techniques used to prepare the datasets for machine learning, and our evaluation methodology.

### A. Baseline: Synopsys Fusion Compiler

First, to procure a set of ground truth PPA metrics for each design, we instrument Synopsys Fusion Compiler. We test with two different PDKs: TSMC 5nm (`TSMC5`) and Synopsys 32nm Education (`SAED32`) technologies. In each case, we set the floorplan utilization to 0.7 and include an input delay and output delay of 1 ns. We then run Fusion Compiler's foundation flow with the clock period target set by the value of the design's `clk_period`. After each run, we extract the following PPA metrics: total standard cell area in $\mu m^2$ (`AREA`), total power in pW (`POWER`), total negative timing slack in ns (`TNS`), and worst negative timing slack in ns (`WNS`).

## *B. Graph Dataset Encoding: One-Hot Vectorization*

As mentioned in Section II-B, FastPASE trains on a per-design basis. To prepare training datasets, we perform {997, 500, 482} Fusion Compiler physical design runs for {`RandRTL`, `SIMDU`, `SweRV`}, respectively, sampled randomly across the design space with Latin hypercube sampling. In order to make predictions, we output the non-technology-mapped netlists from the elaboration step of the Fusion Compiler run and use Yosys [26] to obtain the netlist in an intermediate *GraphIR* dataflow representation. We ignore ports and net names and only consider the node operations and their fan-in/fan-out, as illustrated in Fig. 3. We enumerate all possible operations present in each training dataset to perform one-hot vectorization of the operation types. For each node, we end up with a node-level input feature vector that encodes the node's operation/type, e.g., AND, OR, ADDER, SELECT, INPUT, etc. The node input feature vector construction is illustrated in Fig. 3. As our final preprocessing step, we perform normalization of the PPA metric ground truth labels, which enables training of the FastPASE model in fewer epochs.

In addition to node type feature vector, we also prepare a graph-level feature vector that contains graph-level attributes. In this case, the only graph-level feature we want to include is clock period, but this vector could in principle include other physical implementation settings, e.g., the option of which floorplan to use. The graph-level vector (only one element here) is concatenated to each node's input feature vector to allow node-level representations to encode information about the timing requirement. In summary, the per-node input features to the model are (1) the clock period and (2) the components of the one-hot vector for operation type—no manual feature selection is performed to reduce input dimensionality.

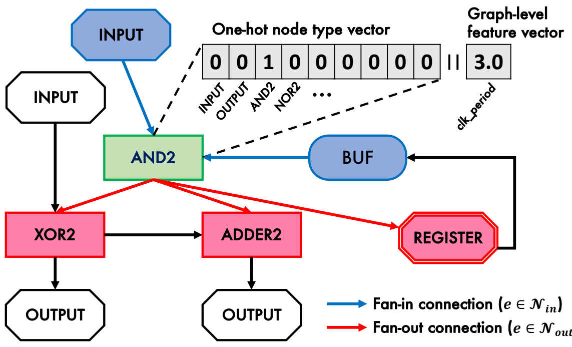


Fig. 3. Illustration of node input feature vector construction from a one-hot node type vector and a graph-level value vector (which embeds clock period). Also highlighted are the fan-in and fan-out connections for the AND2 cell.

## *C. FastPASE Architecture & Training*

The FastPASE predictive AI model is based on the concept of graph convolutions. Since the PPA of an individual cell depends on its fan-in and fan-out, we apply "fan-in fan-out convolutions" across our netlist graph to develop learned node representations that can be used for estimating PPA. Shallow node representations can be built upon to obtain deeper representations. We define a "fan-in fan-out convolution" operation as producing two hidden representation vectors $\{\vec{h}^l_{i,fanin}, \vec{h}^l_{i,fanout}\}$ for each node $i$ at layer $l$ as follows:

$$\vec{h}^{(l+1)}_{i,fanin} = \sigma\left(\vec{b}^{(l)}_{in} + \sum_{j\in\mathcal{N}_{in}(i)} \boldsymbol{W}^{(l)}_{in}\left(\frac{h^{(l)}_{j,fanin}}{c_{in,ji}} \middle|\middle| \frac{h^{(l)}_{j,fanout}}{c_{out,ji}}\right)\right) \tag{1}$$

$$\vec{h}^{(l+1)}_{i,fanout} = \sigma\left(\vec{b}^{(l)}_{out} + \sum_{j\in\mathcal{N}_{out}(i)} \boldsymbol{W}^{(l)}_{out}\left(\frac{h^{(l)}_{j,fanin}}{c_{in,ji}} \middle|\middle| \frac{h^{(l)}_{j,fanout}}{c_{out,ji}}\right)\right) \tag{2}$$

Above, $\{i,j\}$ are node indices, $\mathcal{N}_{in}(i)$ is the fan-in node set of node $i$, $\mathcal{N}_{out}(i)$ is the fan-out node set of node $i$, $\{\boldsymbol{W}^{(l)}_{in}, \vec{b}^{(l)}_{in}\}$ and $\{\boldsymbol{W}^{(l)}_{out}, \vec{b}^{(l)}_{out}\}$ are respectively the weights and biases of the fan-in/fan-out convolutions at layer $l$, $c_{in,ji}$ and $c_{out,ji}$ are respectively the product of the square root of the fan-in and fan-out degrees of nodes $i$ and $j$ (i.e., $c_{in,ji} = \sqrt{|\mathcal{N}_{in}(j)|}\sqrt{|\mathcal{N}_{in}(i)|}$, $c_{out,ji} = \sqrt{|\mathcal{N}_{out}(j)|}\sqrt{|\mathcal{N}_{out}(i)|}$). The $||$ symbol refers to a concatenation operation. Finally, $\sigma$ is a non-linear activation function (rectified linear unit, ReLU, is utilized in this work).

Our FastPASE model, shown in Fig. 4, consists of a very lightweight two-layer graph convolutional network (GCN). In the first layer, we generate 32 hidden features per node (16 fan-in features, 16 fan-out features), and at the second layer, we generate only two node-level features (one for fan-in, one for fan-out). Finally, we need to aggregate the node-level predictions to make predictions about the target metric. The appropriate aggregation operation to perform depends on the type of metric being predicted. For `AREA` and `POWER`, a summation operation is appropriate, as the effect of each cell on these metrics will be cumulative. For `FREQ` and `WNS`, we use the minimum and maximum reductions, respectively, to estimate the critical path length. Each metric's model tensors $\{\boldsymbol{W}, \vec{b}\}$ only consume 40 kB of space in total when encoded as doubles.

Model training is performed in a supervised learning fashion using the Adam optimizer (with a learning rate of 0.01) [27]. A smooth L1 loss function, defined below, is employed, which proves to be less sensitive to outliers than the commonly-used mean squared error (MSE) loss:

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}\begin{cases} 0.5(x_n - y_n)^2, & \text{if } |x_n - y_n| < 1 \\ |x_n - y_n| - 0.5, & \text{otherwise} \end{cases} \tag{3}$$

Above, $\{x_n, y_n\}$ are the prediction and ground truth for sample $n$, and $N$ is the total number of samples. For each design, we perform 30 epochs of training for each metric model (50 for the randomly-generated RTL) to bring the loss down to a reasonably low value, as shown in Fig. 5.

## *D. FastPASE-in-the-Loop: FastPASE as DSO Proxy Metric Engine*

During design space optimization (DSO), there is often little transparency into how a particular RTL parameter setting affects the target output metrics. Thus, Bayesian inference and/or reinforcement learning methods are often used for *black box optimization* to discover and reduce uncertainty about the design space in pursuit of good design solutions. Typically, DSO for chip design employs full Fusion Compiler runs to evaluate points in the design space. This can be extremely slow—evaluating a single design point may take hours to complete. One goal of developing

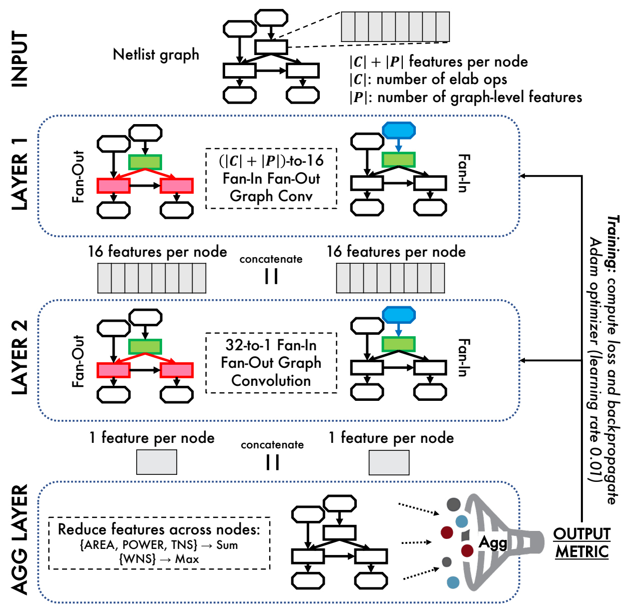

INPUT
Netlist graph
$|C| + |P|$ features per node
$|C|$: number of elab ops
$|P|$: number of graph-level features
LAYER 1
Fan-Out
($|C| + |P|$)-to-16 Fan-In Fan-Out Graph Conv
Fan-In
16 features per node
concatenate
16 features per node
LAYER 2
Fan-Out
32-to-1 Fan-In Fan-Out Graph Convolution
Fan-In
1 feature per node
concatenate
1 feature per node
AGG LAYER
Reduce features across nodes: {AREA, POWER, TNS} → Sum {WNS} → Max
Agg
OUTPUT METRIC
Training: compute loss and backpropagate
Adam optimizer (learning rate 0.01)

Fig. 4. The FastPASE model architecture. At the input, node feature vectors encode both the node's operation and the graph-level feature values. At layers 1 and 2, fan-in/fan-out convolutions are performed, and finally an aggregation layer reduces features across all nodes to produce the final metric prediction.

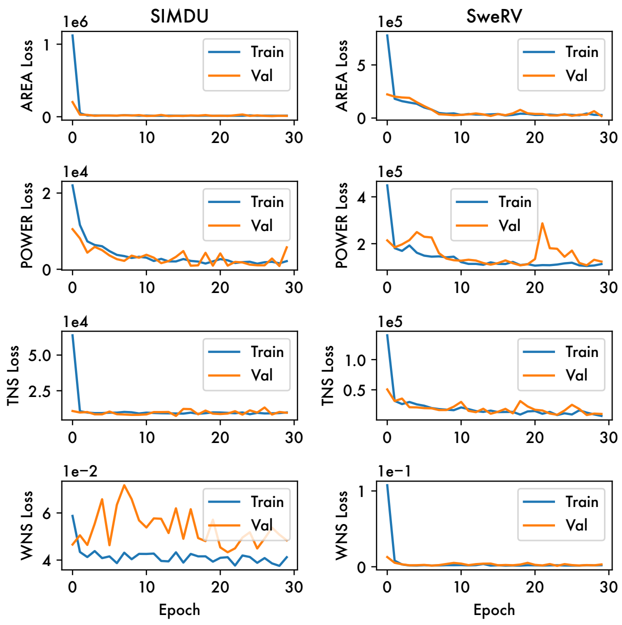


Fig. 5. Training and validation loss for SIMDU and SweRV across different PPA metrics. 10% of the training dataset is withheld during the training process for validation.

FastPASE is to have a fast proxy metric engine to drive RTL design space optimization without full Fusion Compiler evaluations. The resulting "FastPASE-in-the-loop pipeline" is depicted in Fig. 6. After training FastPASE on a few designs, it can be instrumented to speculate PPA without full physical design runs.

Another issue lies in the fact that DSO often involves optimizing across multiple objectives. There are two main strategies for dealing with multi-objective optimization—one is to identify a set of *pareto-optimal solutions* over the set

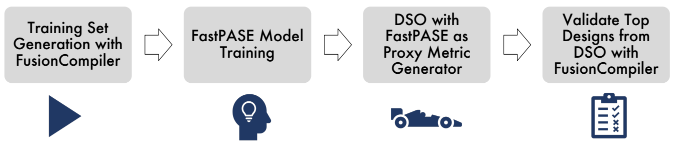

Training Set Generation with FusionCompiler
FastPASE Model Training
DSO with FastPASE as Proxy Metric Generator
Validate Top Designs from DSO with FusionCompiler

Fig. 6. FastPASE-in-the-loop pipeline, where FastPASE is used as a proxy metric generator for the commercial reinforcement-learning based design space optimization tool.

of metrics being considered, and the second is to define an aggregate design score, which combines several metric objectives into a single metric. Defining a design score that reflects one's optimization goals accurately has the advantage of enforcing a strict ordering on design "goodness" (pareto-optimality does not enforce such an ordering). To demonstrate FastPASE's efficacy in helping explore design spaces effectively, we apply a commercial reinforcement-learning based optimization tool, with FastPASE as a GCN-based value function. In our case study, our optimization goal is to minimize the aggregate design score defined by:

$$\text{Score}(D) = \sum_{m \in \{\text{metrics}\}} w_m \times \begin{cases} (D_m/D_{m,0})^2 & \text{if } D_m/D_{m,0} > 1 \\ D_m/D_{m,0} & \text{if } D_m/D_{m,0} \leq 1 \end{cases} \quad (4)$$

Above, $D$ represents a particular design, and $\{\text{metrics}\}$ is the set of metrics (e.g., `WNS`, `AREA`, `POWER`) composing the aggregate design score. $D_m$ is the value of metric $m$ in design $D$, and $D_{m,0}$ is the baseline value of metric $m$ in design $D$. The ratio $D_m/D_{m,0}$ is the fractional improvement of design $D$'s metric $m$ over the baseline metric $m$. It can be seen that design scores are penalized quadratically for metric $m$ being worse than its baseline ($D_m/D_{m,0} > 1$), and rewarded linearly for being better than the baseline ($D_m/D_{m,0} < 1$). Finally, $w_m$ is the "target multiplier" of metric $m$, specifying the user-defined weight/importance of improving a particular design metric $m$.

The target multipliers we use in our case study are given below (`CYCLES` represents the number of cycles obtained from RTL simulation of the design—it is not predicted by FastPASE, but rather taken directly from the ground truth simulation):

| **Metric** | **WNS** | **AREA** | **POWER** | **CYCLES** |
|---|---|---|---|---|
| **Target Multiplier** ($w_m$) | 0.5 | 0.9 | 0.8 | 0.9 |

To have a baseline for comparison in our case study, we run a session with full Fusion Compiler run for design score evaluation. This run explores the design space for 3 "learning generations" with 20 designs evaluated per generation. We allow FastPASE to search the solution space until it stops improving significantly. To compare against this baseline after FastPASE DSO is complete, we perform an evaluation of FastPASE's top 15 identified points by running Fusion Compiler on the same design points to evaluate the ground truth quality of top FastPASE-identified designs.

## V. Results and Discussion

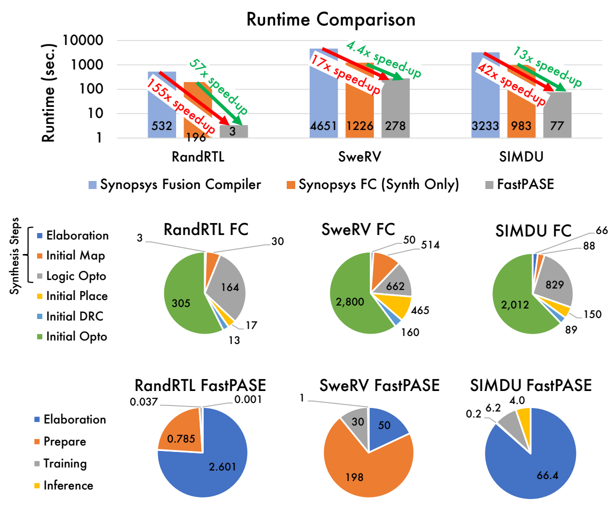
In this section, we present our results and discuss the implications. We show (1) the speedup obtained by FastPASE over Fusion Compiler, (2) FastPASE's goodness of fit, (3) the results of instrumenting DSO with FastPASE as a metric speculation engine.

### A. Training and Model Runtime

In Fig. 7, we compare the total runtime of FastPASE with Fusion Compiler (up to placement and also synthesis-only) for the three designs studied. Fusion Compiler's compilation process (up to placement) consists of six sub-steps: (1) *elaboration*, where RTL is converted to generic logic operators, (2) *initial mapping*, where these operators are mapped to standard cells, (3) *logic opto*, where logic minimization and optimization occurs, (4) *initial place*, where an initial standard cell placement is generated, (5) *initial DRC*, where initial design rules are checked, and finally (6) *initial opto*, where the initial placement is further optimized.

FastPASE's greatest speedup (155×) is realized for `RandRTL`, the smallest design, due to the fact that it can be elaborated and prepared quickly. FastPASE also achieves a large speedup on `SIMDU`—despite the design containing ∼74K cells after place and route, SIMDU involves mostly simple arithmetic logic that is represented in fewer operations with an elaboration-only netlist. Finally, on SweRV, FastPASE shows the least speed improvement, which is still more than an order of magnitude. A synthesis-only Fusion Compiler run (consisting of only elaboration, initial mapping, and logic opto) is roughly 3-4× faster than a run up to placement, but synthesis alone does not account for PPA perturbations that arise from physical implementation.

Fig. 7 also shows pie charts providing runtime breakdowns. The runtime breakdowns indicate that Fusion Compiler spends a majority of its runtime in the "Initial Opto" step, where it is focused on logic optimization after placement. FastPASE, on the other hand, spends a majority of its runtime on elaboration/preparation of the dataset—training and inference do not constitute a significant runtime component at all. (The elaboration/preparation stages were not implemented in the most efficient way and could be improved in a production environment.) It should be noted that the "effective training time" plotted in the pie chart is a per-sample metric, calculated by dividing the total time spent training (across all samples and all metrics, for 30 epochs) by the total number of inference samples. Overall, the 1-2 orders of magnitude speed gain from FastPASE positions it as a compelling tool for rapid PPA speculation.



```
Machine Specs:
Linux 3.10.0-1160.31.1.el7.x86_64 x86_64
OS: CentOS Linux release 7.3
CPU: 40x2400 MHz, Intel Xeon (2 socket, 20 core, No HT)
Memory: 754 GB RAM, 256 GB Swap
```

Fig. 7. Runtime comparison and breakdown for FastPASE vs. Fusion Compiler (FC). Host machine specs are also given.

## B. Goodness of Fit by Metric and Dataset

To evaluate the predictive capability of FastPASE, we show scatterplots of the predictions vs. the Fusion Compiler ground truth values for each design+metric combination in Fig. 8. We also compute various goodness of fit metrics: (1) $R^2$ is the correlation coefficient between the predictions and the ground truth, (2) RMSE is the root mean square error, (3) NRMSE is the normalized root mean square error, obtained by dividing RMSE by the standard deviation of the ground truth labels (can be represented in %), (4) MAE is the mean absolute error, and (5) NMAE is the normalized mean absolute error, obtained by dividing MAE by the standard deviation of the ground truth labels (can be represented in %). We can also compute "ordinal" versions of each of the five goodness of fit metrics above. Ordinal means that the predicted/actual ordered ranks are used in place of the metric values. Ordinal ranking is scale-free by default and useful for DSO, where we often care about being able to compare designs well against one another, rather than perfectly accurate predictions.

We believe that NMAE is the goodness of fit metric that best captures the relative goodness fairly across all metrics and designs, since (1) it is scaled by the standard deviation of the underlying ground truth distribution (as opposed to MAE and RMSE), (2) it is less sensitive to outliers (as opposed to $R^2$ and NRMSE), and (3) it has an intuitive and interpretable meaning. For `SweRV` and `SIMDU`, our models show NMAE between 1.65%-34.3% for power, 5.71%-18.6% for area, 2.54%-13.5% for total negative slack (TNS), and 4.21%-23.5% for worst negative slack (WNS). Even for `RandRTL`, which consists of totally unrelated designs, the NMAE remains at 28.0% for power, 26.0% for area, 30.9% for TNS, and 26.0% for WNS. The ordinal (ranked) correlations are similarly good.

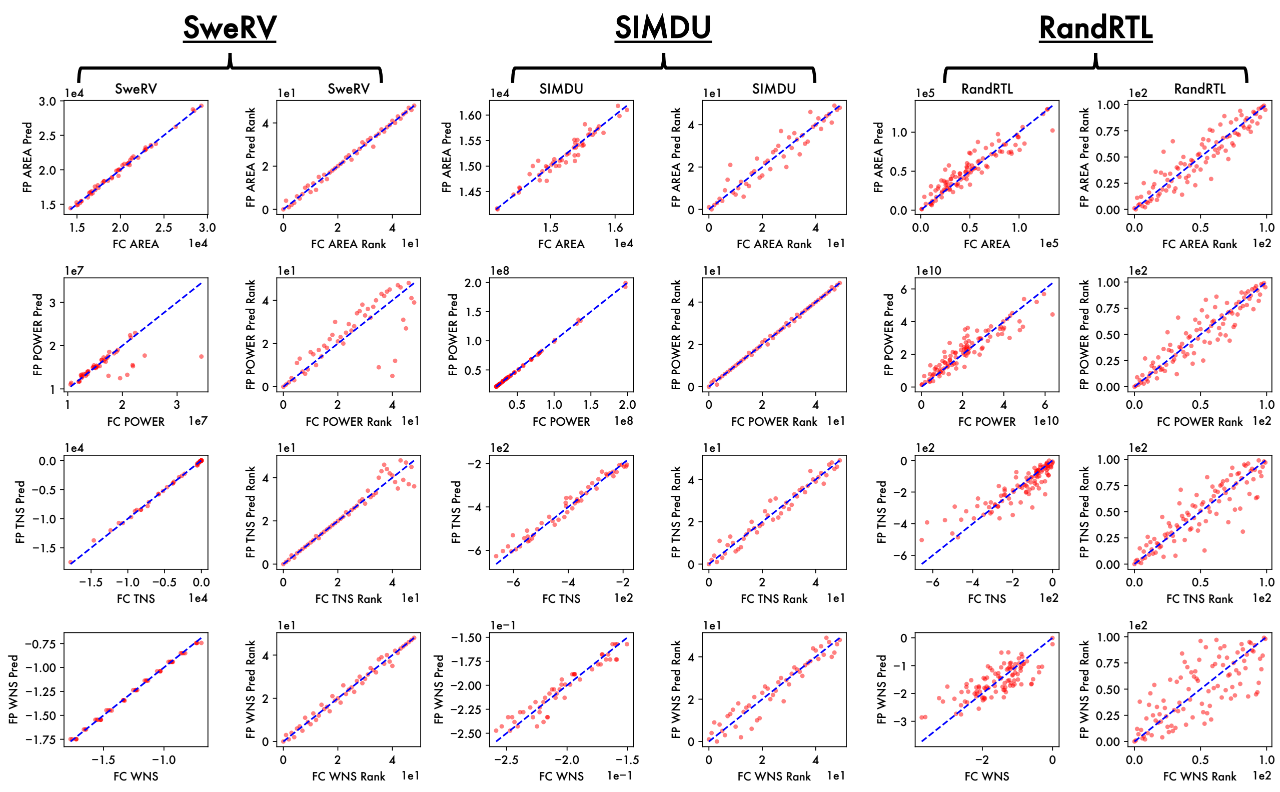
While the FastPASE results are generally well-correlated, it is important to understand the model limitations by analyzing outliers and scenarios in which the predictive capability is diminished. For example, SweRV shows several outliers in the power, leading to lower $R^2$ correlation and higher RMSE. Upon analysis, these outliers are caused by design permutations that are undersampled in the training set. In order for FastPASE to have the best predictive capability, the set of training designs must closely represent the inferred set.

In SIMDU, it is significantly more difficult to predict area than in SweRV. This is because SIMDU elaborates to relatively small dataflow graphs with mainly high-level ALU-type operations, e.g., ADD, MULT, etc. This means that Fusion Compiler requires more effort to synthesize the dataflow graph to standard cells and lay them out. Since FastPASE does not have visibility into this process, the predictive capability is slightly worse when the synthesis effort is higher.

In RandRTL, it is clear that WNS is difficult to predict across designs that are unrelated. This is because timing is based on a critical path, which can be difficult to isolate. Future work might be able to improve the timing results across unrelated designs by performing convolutions in a manner that replicates the behavior of traditional static timing analysis (STA).

## C. RISC-V Microarchitectural Exploration: PPA & Runtime

Once we have a trained set of FastPASE models for SweRV, we can run FastPASE-in-the-loop (described earlier in Section IV-D). Fig. 9 shows the top-5 and top-1 design scores vs. sample number



| Design | Metric | $R^2$ | RMSE | NRMSE (%) | MAE | NMAE (%) | Ordinal $R^2$ | Ordinal RMSE | Ordinal NRMSE (%) | Ordinal MAE | Ordinal NMAE (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SweRV | AREA | 0.995 | 253.536 | 6.947 | 208.352 | 5.709 | 0.991 | 1.309 | 9.163 | 0.939 | 6.570 |
| SweRV | POWER | 0.363 | 3.352e6 | 79.013 | 1.456e6 | 34.312 | 0.570 | 9.269 | 64.871 | 5.878 | 41.134 |
| SweRV | TNS | 0.998 | 208.685 | 4.552 | 116.629 | 2.544 | 0.942 | 3.399 | 23.786 | 1.878 | 13.140 |
| SweRV | WNS | 0.997 | 0.017 | 5.181 | 0.014 | 4.207 | 0.982 | 1.874 | 13.112 | 1.388 | 9.712 |
| SIMDU | AREA | 0.933 | 118.546 | 25.530 | 86.432 | 18.613 | 0.915 | 4.219 | 28.942 | 3.040 | 20.854 |
| SIMDU | POWER | 0.999 | 1.0645e6 | 2.537 | 6.919e5 | 1.648 | 0.998 | 0.693 | 4.753 | 0.400 | 2.744 |
| SIMDU | TNS | 0.970 | 23.394 | 17.058 | 18.548 | 13.525 | 0.959 | 2.912 | 19.976 | 2.240 | 15.366 |
| SIMDU | WNS | 0.916 | 0.008 | 28.686 | 0.007 | 23.486 | 0.911 | 4.299 | 29.490 | 3.440 | 23.598 |
| RandRTL | AREA | 0.874 | 11046.148 | 35.345 | 8111.264 | 25.954 | 0.861 | 10.758 | 37.083 | 8.160 | 28.127 |
| RandRTL | POWER | 0.851 | 5.405e9 | 38.352 | 3.950e9 | 28.025 | 0.855 | 10.999 | 37.913 | 8.020 | 27.644 |
| RandRTL | TNS | 0.790 | 67.245 | 45.614 | 45.509 | 30.870 | 0.783 | 13.441 | 46.330 | 9.600 | 33.090 |
| RandRTL | WNS | 0.570 | 0.478 | 65.262 | 0.382 | 52.182 | 0.482 | 20.785 | 71.643 | 16.340 | 56.323 |

Fig. 8. Results for FastPASE (denoted FP) vs. Synopsys Fusion Compiler (denoted FC).

for both FastPASE-in-the-loop and a baseline Fusion Compiler-driven run. Once FastPASE has explored the design space, Fusion Compiler can be run on the top 15 discovered design points to validate their goodness—the best validated point is plotted at the end of the FastPASE run with a star. FC-based DSO in our setup takes ∼6.5 hours to generate 60 runs, versus ∼0.5 hours to generate 600 runs in our FastPASE-based DSO setup. Validation of the top 15 FastPASE points with Fusion Compiler takes 0.73 hours. It can be seen that the best design score achieved by FastPASE is 0.062, which is lower than the best design score of 0.151 achieved by Fusion Compiler—this improved design point is discovered in 13× less time than the traditional approach.

### *D. Comparison with Similar Prior Work*

Table IV shows a comparison between FastPASE and similar prior work. SNS and FastPASE metrics in the table correspond to those reported on their synthetically-generated design inputs (for FastPASE, this is the `RandRTL` dataset, and for SNS, this is the GAN-generated dataset). FastPASE's timing predictions are different from prior work in that target clock period is a

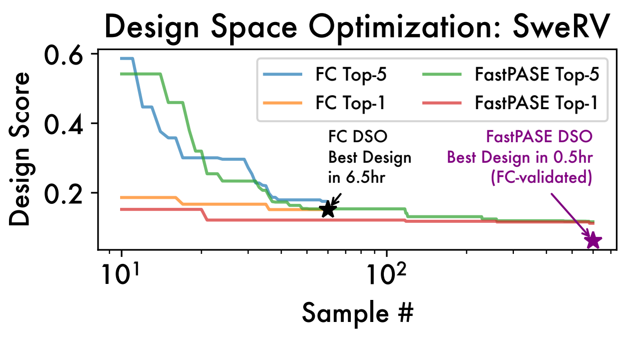


Fig. 9. Design score (lower is better) plotted as a function of sample number during DSO for FastPASE and Fusion Compiler (FC) runs. In ∼6.5 hours, FC-based DSO is able to run 3 generations of samples (with 20 samples per generation). In ∼0.5 hours, FastPASE-based DSO is able to run 30 generations of samples (also with 20 samples per generation). It can thus be seen that FastPASE-based DSO is able to find an RTL design permutation with significantly better aggregate design score in a fraction of the time.

design input and WNS/TNS are predicted, while in prior work, critical path (CP) length is predicted directly from a netlist or high-level description. Also note that since datasets used across these studies are different, the numbers in the table do not constitute an apples-to-apples comparison.

TABLE IV
COMPARISON BETWEEN FASTPASE AND PRIOR WORK.

| **Reference** | **FastPASE (this work)** | **SNS [16]** | **D-SAGE [17]** | **HLSPP [18]** | **IM-P [19]** |
|---|---|---|---|---|---|
| **Technique** | GCN | DNN | GraphSAGE | GNN | GNN |
| **Design Type** | RTL | RTL | HLS | HLS | HLS |
| **DSO** | Yes | Yes | No | No | Yes |
| **Datasets** | CPU SIMD Random | CPU DNN acc. GAN-gen. | HLS bench | HLS bench | HLS bench |
| **Predicted Metrics** | Cell Area Power WNS/TNS | Cell Area Power CP Len | CP Len | LUT Count DSP Count CP Len | LUT Count DSP Count CP Len |
| **Platform** | ASIC | ASIC | FPGA | FPGA | FPGA |
| **Ground Truth** | Placement | Synthesis | Synthesis | Synthesis | Synthesis |
| **Clk NRMSE** | 0.65 (WNS) 0.46 (TNS) | 0.67 (CP) | 0.82 (CP) | - | - |
| **Pow NRMSE** | 0.38 | 0.6 | - | - | - |
| **Cell Area NRMSE** | 0.35 | 0.22 | - | - | - |
| **Clk NMAE** | 0.52 (WNS) 0.31 (TNS) | 0.38 (CP) | - | 0.06 (CP) | 0.04 (CP) |
| **Pow NMAE** | 0.28 | 0.49 | - | - | - |
| **Cell Area NMAE** | 0.26 | 0.55 | - | - | - |
| **LUT Count NMAE** | - | - | - | 0.22 | 0.092 |

*A "-" indicates that the metric was not reported.*
*CP indicates that critical path length was modeled rather than TNS or WNS.*
*For NRMSE and NMAE, lower is better.*

## VI. CONCLUSION

We have presented FastPASE, a fast PPA speculation engine based on lightweight graph convolutional networks. We demonstrate that our encoding approach, involving one-hot node representations and graph-level feature embedding combined with fan-in fan-out convolution, enables FastPASE to make rapid, well-correlated predictions about PPA from only an elaborated netlist and the target clock period. We have shown that FastPASE can be applied to complex hardware such as SIMD units and RISC-V cores, and a trained model can be used to make predictions 1-2 orders of magnitude faster than a physical design run. Finally, we have demonstrated that using FastPASE "in the loop" enables rapid design space optimization over large RTL design spaces to achieve superior user-defined aggregate design scores in significantly reduced time. In conclusion, we hope that the FastPASE approach to PPA speculation will encourage the further development of fast end-to-end speculation models in the electronic design automation space, which will in turn enable rapid/large-scale design space optimization.

## REFERENCES

[1] "Synopsys Chip Implementation and Signoff." https://www.synopsys.com/implementation-and-signoff.html, 2022. Synopsys, Inc.

[2] "Cadence Design Tools." https://www.cadence.com/en_US/home/tools/tools-a-z.html, 2022. Cadence, Inc.

[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[4] T. Marena, "RISC-V: high performance embedded SweRV™ core microarchitecture, performance and CHIPS Alliance," *Western Digital Corporation*, 2019.

[5] "Siemens Aprisa." https://eda.sw.siemens.com/en-US/ic/aprisa/, 2022.

[6] A. Venkatachar, "Confluence of AI/ML with EDA and Software Engineering," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pp. 13–15, IEEE, 2021.

[7] A. F. Budak, Z. Jiang, K. Zhu, A. Mirhoseini, A. Goldie, and D. Z. Pan, "Reinforcement learning for electronic design automation: Case studies and perspectives," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 500–505, IEEE, 2022.

[8] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk, and G. De Micheli, "Deep learning for logic optimization algorithms," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, IEEE, 2018.

[9] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "Drills: Deep reinforcement learning for logic synthesis," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 581–586, IEEE, 2020.

[10] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pp. 145–150, 2020.

[11] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, *et al.*, "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[12] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph neural network inference for transferable power estimation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[13] V. A. Chhabria, Y. Zhang, H. Ren, B. Keller, B. Khailany, and S. S. Sapatnekar, "MAVIREC: ML-aided vectored IR-drop estimation and classification," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1825–1828, IEEE, 2021.

[14] H. Ren, G. F. Kokai, W. J. Turner, and T.-S. Ku, "ParaGraph: Layout parasitics and device parameter prediction using graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[15] H. Ren and M. Fojtik, "Nvcell: Standard cell layout in advanced technology nodes with reinforcement learning," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 1291–1294, IEEE, 2021.

[16] C. Xu, C. Kjellqvist, and L. W. Wills, "SNS's not a synthesizer: a deep-learning-based synthesis predictor," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 847–859, 2022.

[17] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate operation delay prediction for FPGA HLS using graph neural networks," in *Proceedings of the 39th International Conference on Computer-Aided Design*, pp. 1–9, 2020.

[18] N. Wu, H. Yang, Y. Xie, P. Li, and C. Hao, "High-level synthesis performance prediction using GNNs: Benchmarking, modeling, and advancing," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 49–54, 2022.

[19] N. Wu, Y. Xie, and C. Hao, "IronMan-Pro: Multiobjective design space exploration in HLS via reinforcement learning and graph neural network-based modeling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 3, pp. 900–913, 2022.

[20] Z. Zhang, X. Wang, and W. Zhu, "Automated machine learning on graphs: A survey," *arXiv preprint arXiv:2103.00742*, 2021.

[21] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A gentle introduction to graph neural networks," *Distill*, vol. 6, no. 9, p. e33, 2021.

[22] A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *IEEE Transactions Neural Networks*, vol. 8, no. 3, pp. 714–735, 1997.

[23] D. S. Lopera, L. Servadei, G. N. Kiprit, S. Hazra, R. Wille, and W. Ecker, "A survey of graph neural networks for electronic design automation," in *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pp. 1–6, IEEE, 2021.

[24] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78120–78145, 2019.

[25] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 97–108, 2014.

[26] C. Wolf, J. Glaser, and J. Kepler, "Yosys: A free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.